

High Performance Computing

Why it is important

Introduction

HPC =



?

NO





Introduction

Towards a Breakthrough in Software for Advanced Computing Systems

Report from a Workshop
organised by the European Commission
in preparation for HORIZON 2020

held in July 2012 in Brussels, Belgium

Software development has not evolved as fast as hardware capability and network capacity.

Nominal and sustained performance of computing systems is further diverging, unless they are manually optimised which limits portability to other systems.

Development and maintenance of software for advanced computing systems is becoming increasingly effort-intensive requiring dual expertise, both on the application side and on the system side.

...

**In order to program the next generation of computing systems,
everyone must become a parallel programmer!**



STORAGE



PERFORMANCE

1970

```
PROGRAM HELLO  
C  
REAL A(10,10)  
DO 50 I=1,10  
    CALL DGEMM(N,10,I,J,A)  
CONTINUE  
50
```



2018

```
PROGRAM HELLO  
C  
REAL A(10,10)  
DO 50 I=1,10  
    CALL DGEMM(N,10,I,J,A)  
CONTINUE  
50
```

SOFTWARE

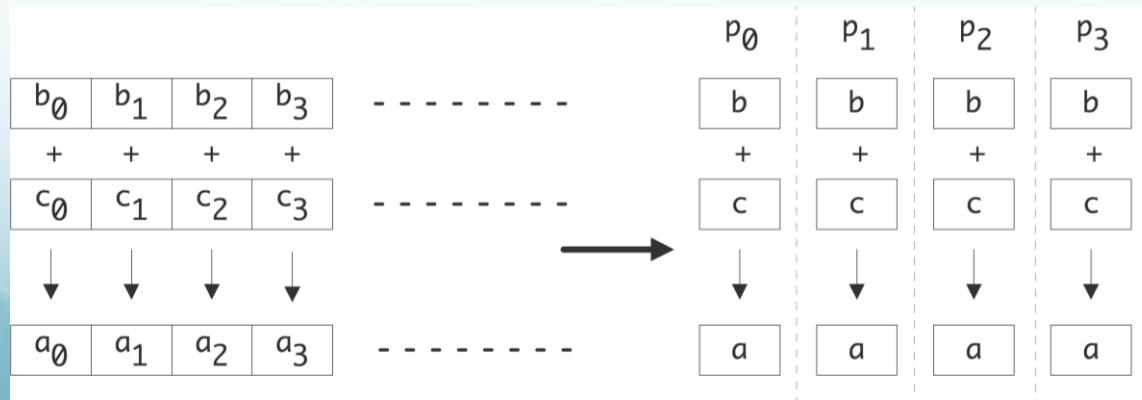
What is Parallel Computing?

Simultaneous use of multiple resources to solve a computational problem.

In scientific codes, there is often a large amount of work to be done, and it is often regular to some extent, with the same operation being performed on many data.

```
for (i=0; i<n; i++)
```

$$a[i] = b[i] + c[i];$$

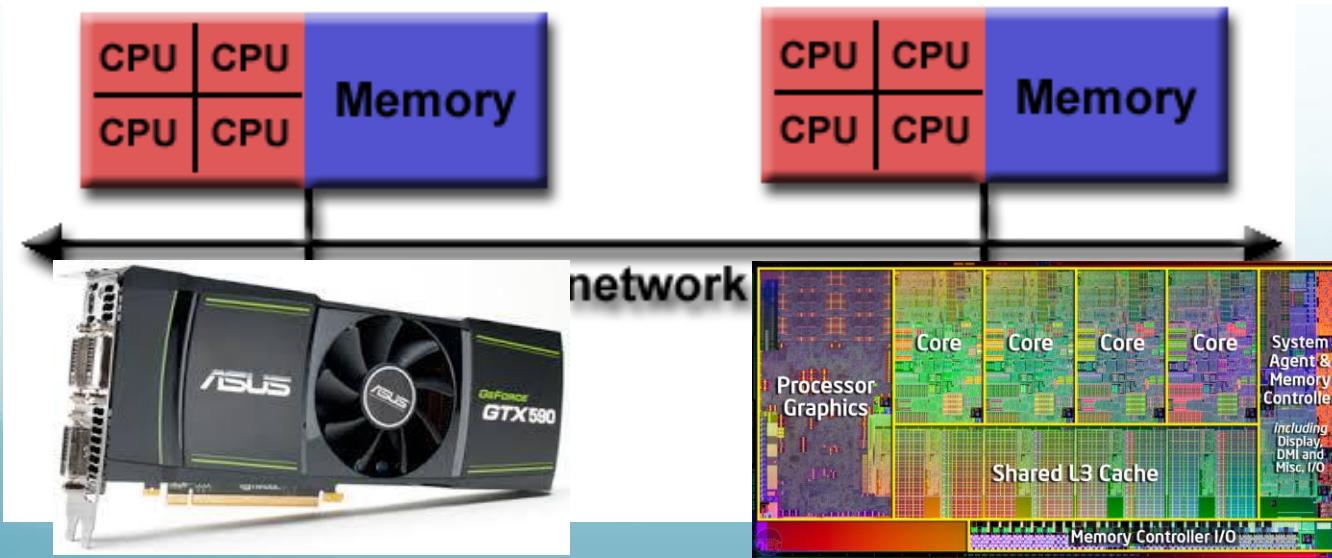


Why use Parallel Computing?

- Save time
- Solve larger problems
 - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer
 - More memory available
 - Higgs discovery “only possible because of the extraordinary achievements of Grid computing”
— Rolf Heuer, CERN DG
- Use of non-local resources
 - Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.
E.g. SETI@home.

Actual CPUs and GPUs

- Modern CPUs have vector instructions that can perform multiple instances of an operation simultaneously. On Intel processors this is known as SIMD Streaming Extensions (SSE) or Advanced Vector Extensions (AVX). GPUs are vector processors.
- By far the most common parallel computer architecture are MIMD (also known as Single Program Multiple Data, SPMD): the processors execute multiple, possibly differing instructions, each on their own data. There is a great variety in MIMD computers. Some of the aspects concern the way memory is organized, and the network that connects the processors.



Parallel Programming

The simplest case is the **embarrassingly parallel problem**, where little or no effort is required to speedup the process

- No dependency/ communication between the parallel tasks

Examples :

- Distributed relational database queries
- Rendering of computer graphics
- Event simulation and reconstruction in particle physics
- Brute-force searches in cryptography
- Ensemble calculations of numerical weather prediction

BUT normally you have to work (hard) to parallelize an application!

To summarize

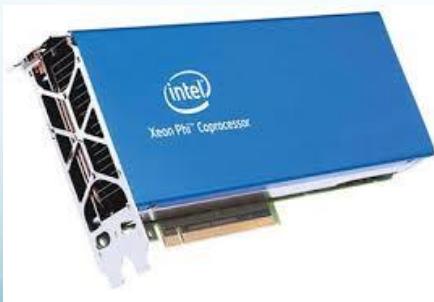
In order to write efficient scientific codes, **it is important to understand the resource architecture.**

The difference in speed between two codes that compute the same result can range from a few percent to orders of magnitude, depending only on factors relating to how well the algorithms are coded for the specific architecture.

Clearly, it is not enough to have an algorithm and “put it on the computer”: some knowledge of computer architecture is useful, sometimes crucial.

JUNE 2013 - NOVEMBER 2015

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890



88% of the cores



46% of the cores



Homogeneous cores

62%

65%

85%



NOVEMBER 2016

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2					74%	
3						
4						

The diagram illustrates the internal structure of the ShenWei 26010 chip. It features a 2D grid of 'CPE cluster' units, each containing a 4x4 grid of processing elements. These clusters are interconnected by a 'NoC' (Network-on-Chip). Each cluster is connected to a 'SI' (System Interface) unit. Additionally, each cluster is connected to a 'Main memory' block, which further connects to 'MC' (Memory Controller) and 'MPE' (Memory Protection Element) units. The entire system is organized into 'Group' units, which are interconnected via the NoC.

The ShenWei 26010 is a 260-core, 64-bit RISC chip that exceeds 3 teraflops at maximum tilt, putting it on par with Intel's Knight's Landing Xeon Phi.



JUNE 2019



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	Sierra - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRPCP	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Texas Advanced Computing Center/Univ. of Texas United States	Frontera - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR Dell EMC	448,448	23,516.4	38,745.9	
6	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	387,872	21,230.0	27,154.3	2,384
7	DOE/NNSA/LANL/SNL United States	Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	979,072	20,158.7	41,461.2	7,578

74%

Today

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
...					
119	davinci-1 - BullSequana X, AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR Infiniband, Atos Leonardo SpA Italy	38,400	3,449.0	6,412.0	

- <https://www.arm.com/blogs/blueprint/fujitsu-a64fx-arm>
- <https://www.top500.org/news/fugaku-holds-top-spot-exascale-remains-elusive/>

Cloud and High Performance Computing



Rank System

101

Amazon EC2 Cluster, Intel Xeon E5-2680v2 10C 2.800GHz,
10G Ethernet

Vendor	Total Cores	Rmax (TFlops)	Rpeak (TFlops)
Self-made	26,496	484.2	593.5

18 hours, \$33K, and 156,314 cores: Amazon cloud HPC hits a “petaflop”

1.21 petaflops? Great scott!

To get all those cores, Cycle's cluster ran simultaneously in Amazon data centers across the world, in Virginia, Oregon, Northern California, Ireland, Singapore, Tokyo, Sydney, and São Paulo. The bill from Amazon ended up being \$33,000.

\$4,829-per-hour supercomputer built on Amazon cloud to fuel cancer research

A 50,000-core supercomputer deployed on Amazon shows the cloud's potential

- The advantage of pay-as-you-go computing has been an industry goal for many years.
- The Globus Project has shown the power of Grid computing.
- Cloud computing takes Grid computing to a whole new level by using virtualization to encapsulate an operating system (OS) instance

The cluster used a mix of 10 Gigabit Ethernet and 1 Gigabit Ethernet interconnects. However, the workload was what's often known as "embarrassingly parallel," meaning that the calculations are independent of each other. As such, the speed of the interconnect didn't really matter.

Operational costs 1: energy costs

Total cost of ownership: total cost of acquisition and operating costs



June 2014

MFLOPS/W

KW

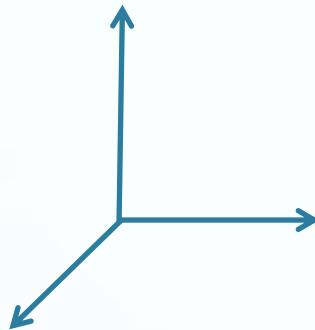
38	2,176.58	DOE/NNSA/LLNL	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom	7,890.00
43	2,143.03	DOE/SC/Oak Ridge National Laboratory	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	8,208.00
49	1,901.54	National Super Computer Center in Guangzhou	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P	17,808.00

Supercomputer's lifelong energy costs almost equal the investment costs

Tianhe-2: $\approx 20\text{MW} \approx \20 million/year for electricity

Operational costs 2: human resources and tools

HARDWARE



DEVELOPMENT
TOOLS

DEVELOPERS

- Development tools
 - GNU gcc
 - Intel Parallel Studio XE (oneAPI)
 - PGI Accelerator Workstation
 - NVIDIA HPC SDK
- free
- ~~from 699 US\$~~ NOW FREE
- ~~from 759 US\$ again~~ NOW FREE
- free

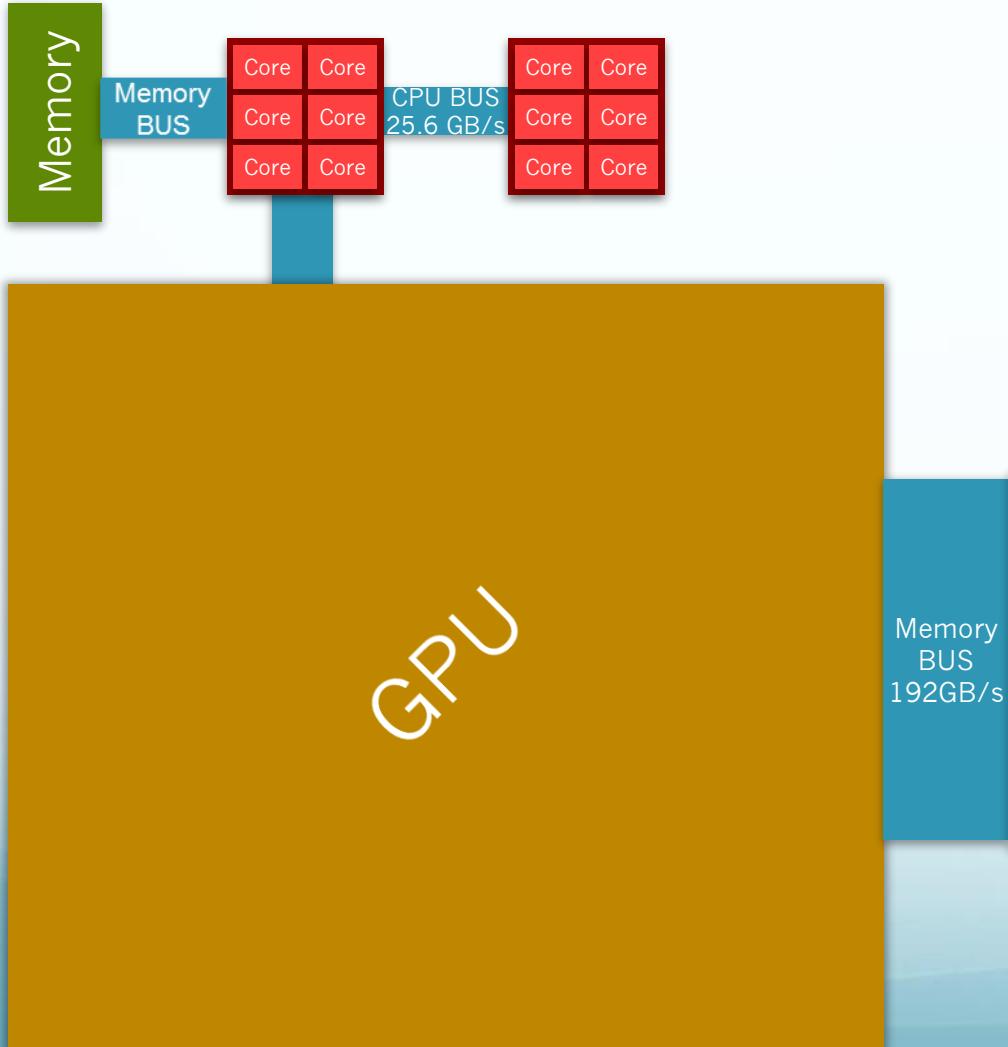


Raw numbers or real performances ?

- A workstation with dual 12-cores CPUs and 4 GPUs
 - 139 GFLOPs SP / 69 GFLOPs DP per CPU
 - 3.5 TFLOPs SP / 1.2 TFLOPs DP per GPU
 - = 14.3 TFLOPs SP / 4.9 TFLOPS DP at about 12,000 US\$
 - 1.6 KWatt
- First position in June 2001,
at the bottom of the list in June 2008
- How can programmers exploit such performance ?
 - programming paradigms and languages
 - development tools

Raw numbers or real performances ?

Architecture of test workstation (fp32)



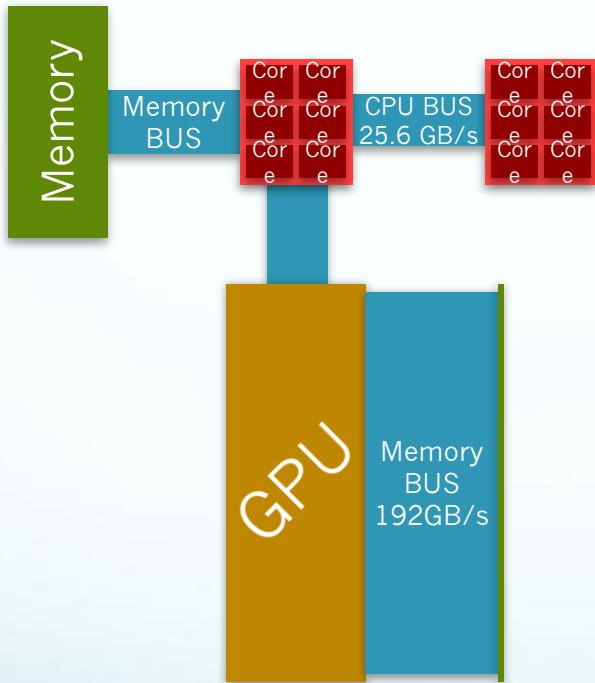
2 Intel Xeon E5645
(115 Gflops, 9.6 per core)

64 GB main memory

nVidia GTX 580
(1581 Gflops)

1.5 GB GPU memory

Architecture of test workstation (fp64)



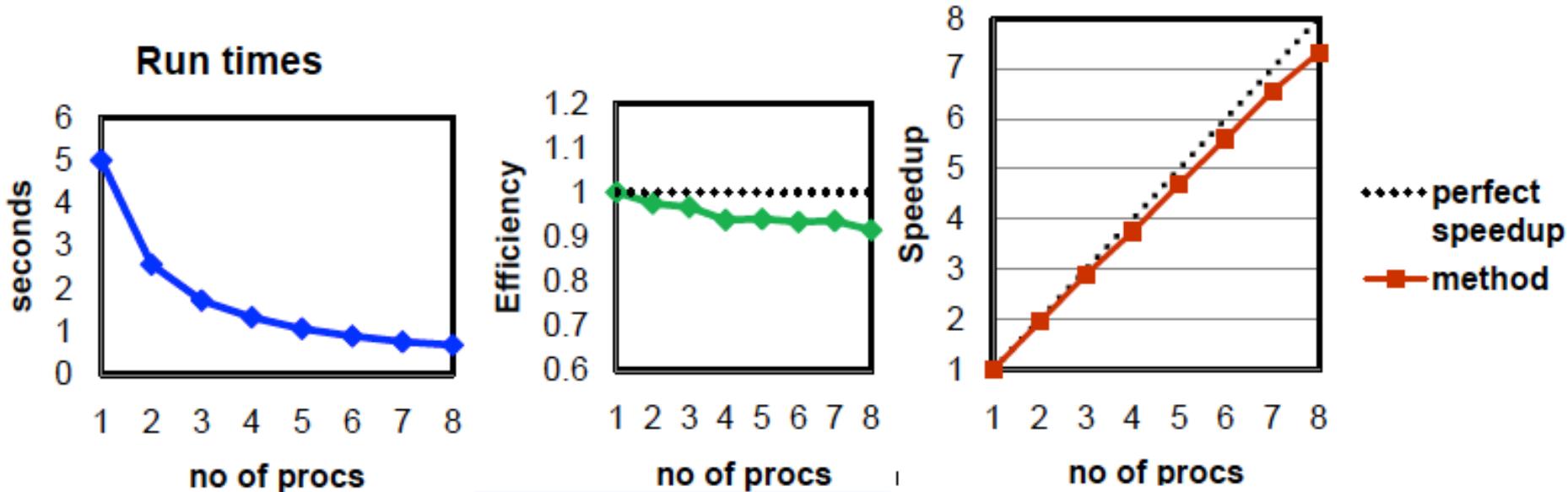
2 Intel Xeon E5645
(58 Gflops)
64 GB main memory

nVidia GTX 580
(198 Gflops)
1.5 GB GPU memory

Parallel Computing Performance Metrics

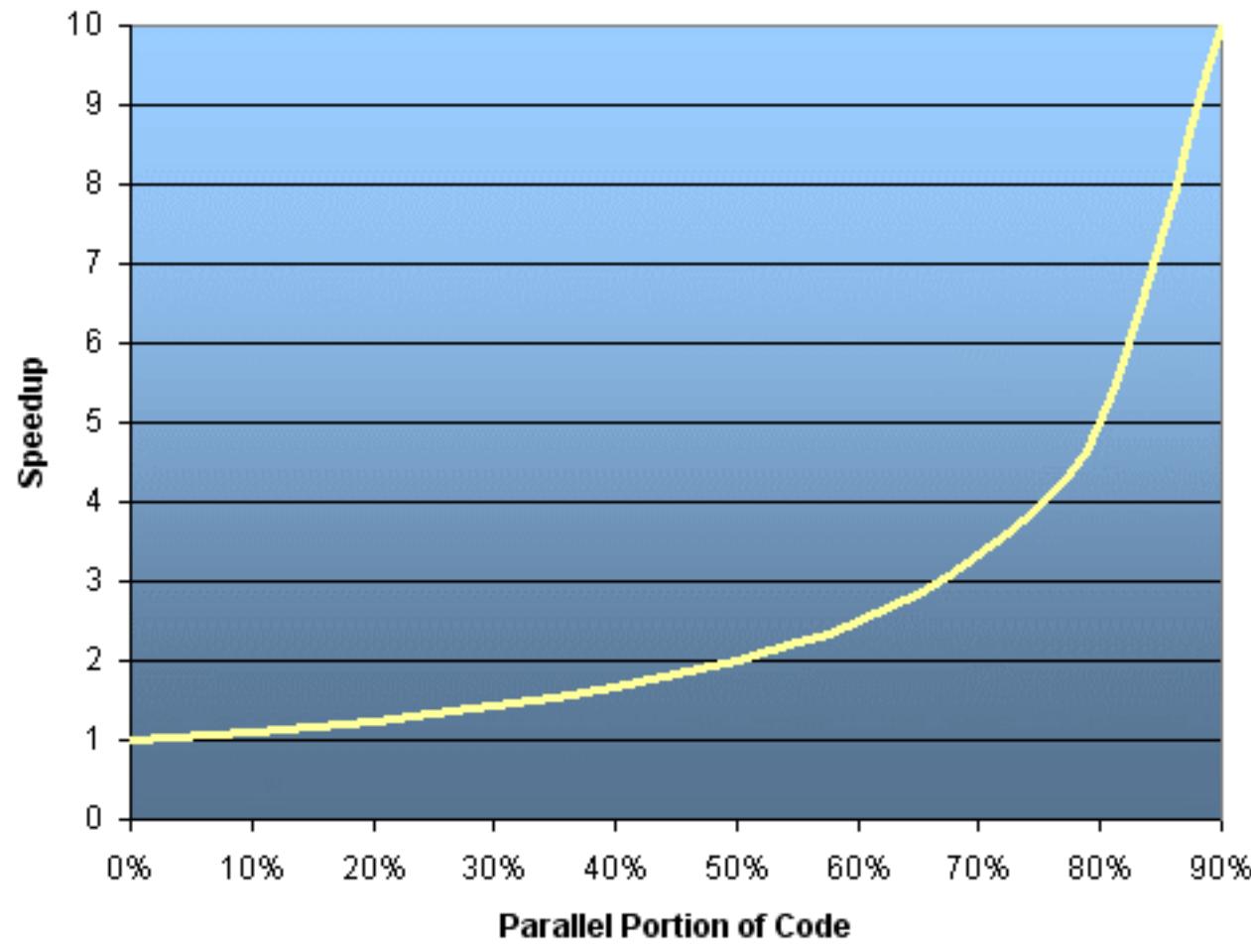
Let $T(n,p)$ be the time to solve a problem of size n using p processors

- Speedup: $S(n,p) = T(n,1)/T(n,p)$
- Efficiency: $E(n,p) = S(n,p)/p$



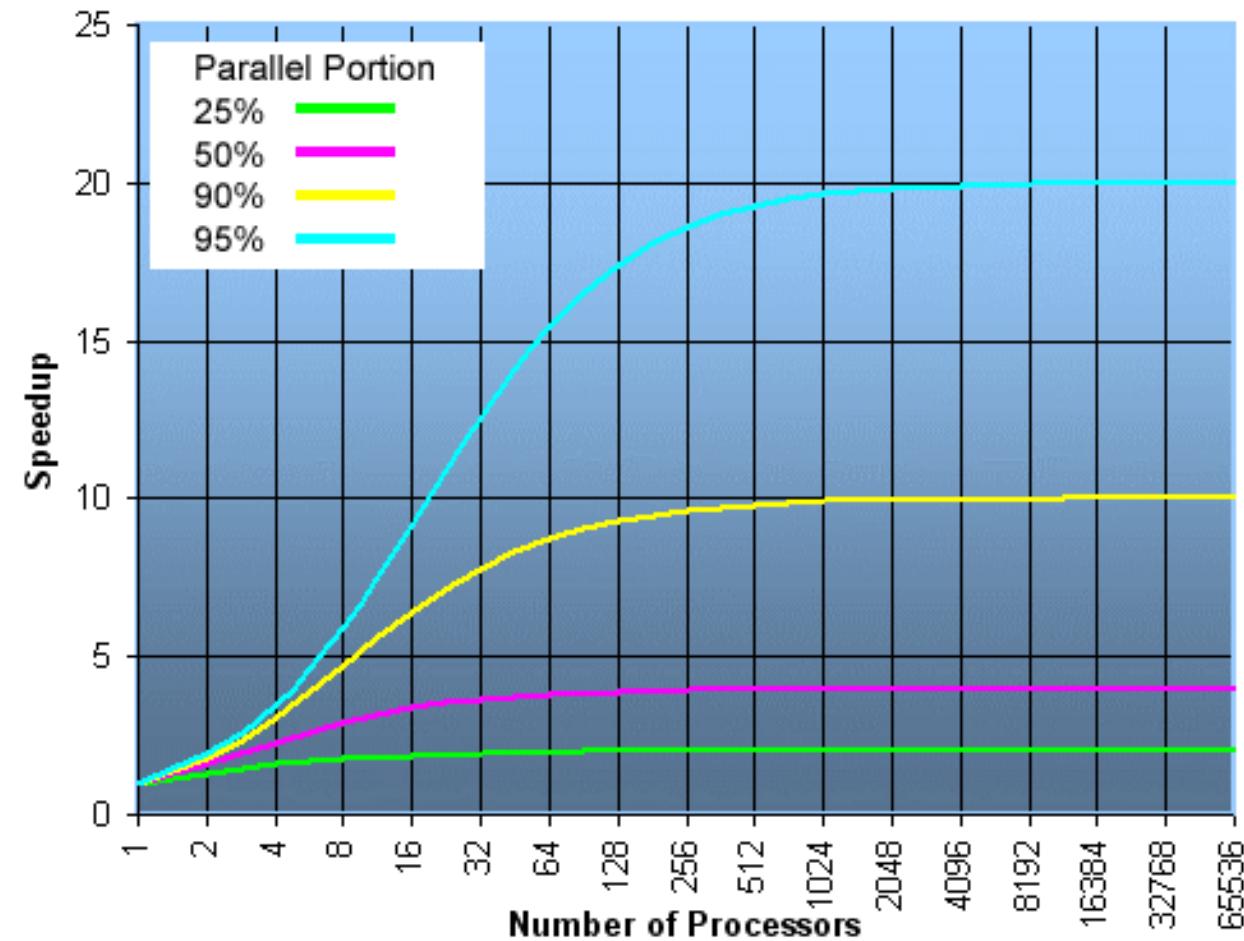
Amdahl's Law

Maximal Speedup = $1/(1-P)$, P parallel portion of code



Amdahl's Law

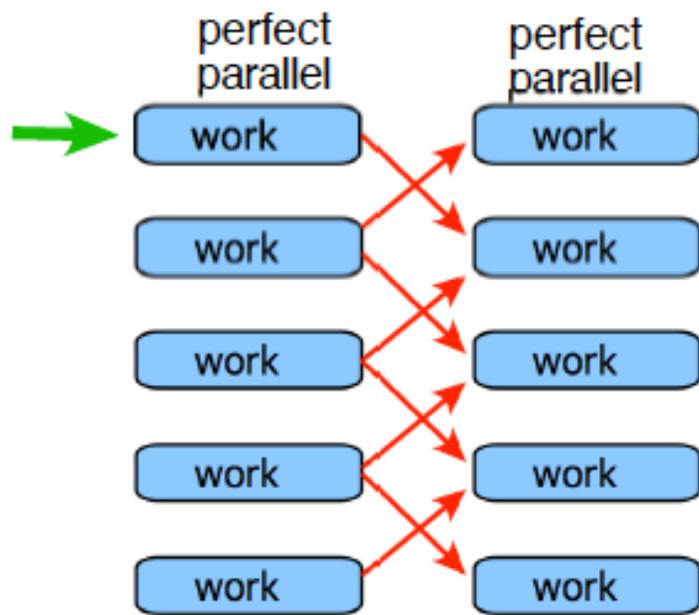
Speedup = $1/((P/N)+S)$,
N no. of processors, S serial portion of code



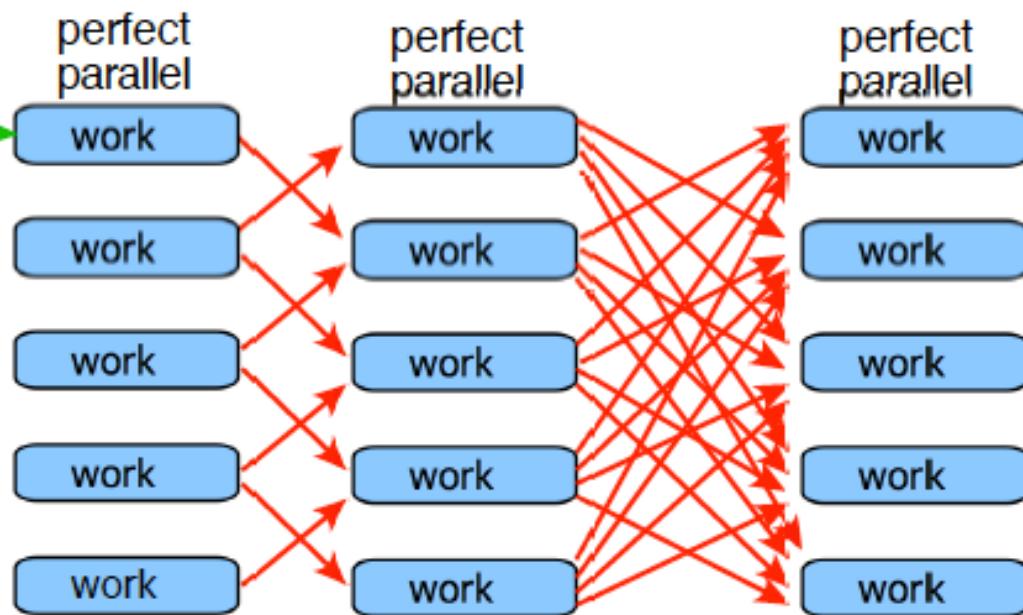
Amdahl Was an Optimist

Parallelization usually adds communications/overheads

serial



serial



neighbor
comm

neighbor
comm

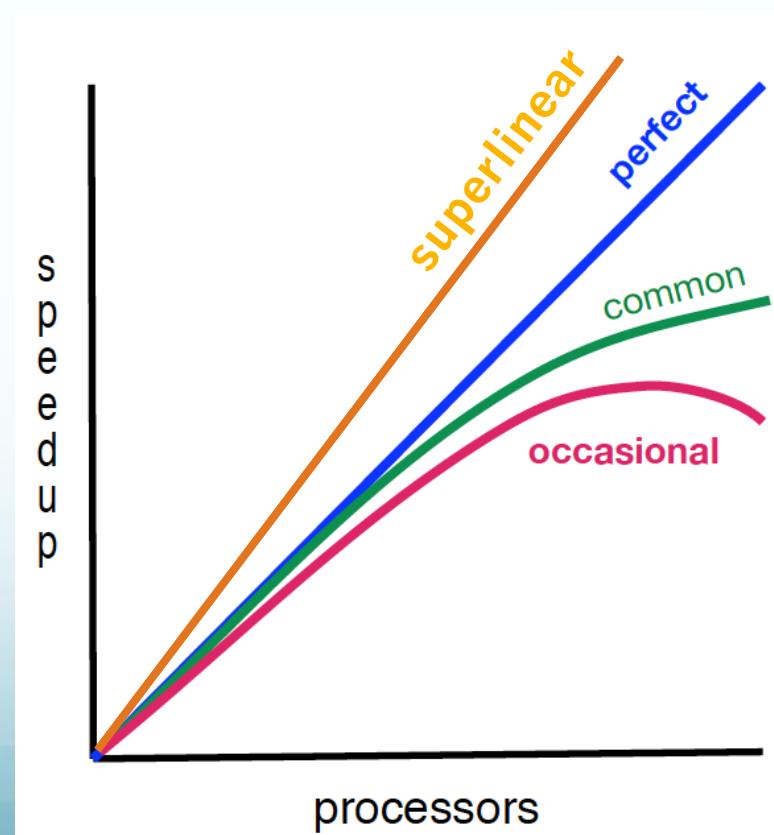
global
comm

To summarize

$$0 < \text{Speedup} \leq p$$

$$0 < \text{Efficiency} \leq 1$$

Linear speedup : speedup = p.



Amdahl was a Pessimist

Superlinear speedup is very rare. Some reasons for
speedup > p (efficiency > 1)

- Parallel computer has p times as much RAM so higher fraction of program memory in RAM instead of disk.

An important reason for using parallel computers

- In developing parallel program a better algorithm was discovered, older serial algorithm was not best possible.

A useful side-effect of parallelization

- In general, the time spent in serial portion of code is a decreasing fraction of the total time as problem size increases.

The lesson is

- Linear speedup is rare, due to communication overhead, load imbalance, algorithm/architecture mismatch, etc.
- Further, essentially nothing scales to arbitrarily many processors.
- However, for most users, the important question is:

**Have I achieved acceptable performance on
my software for a suitable range of data
and the resources I'm using?**

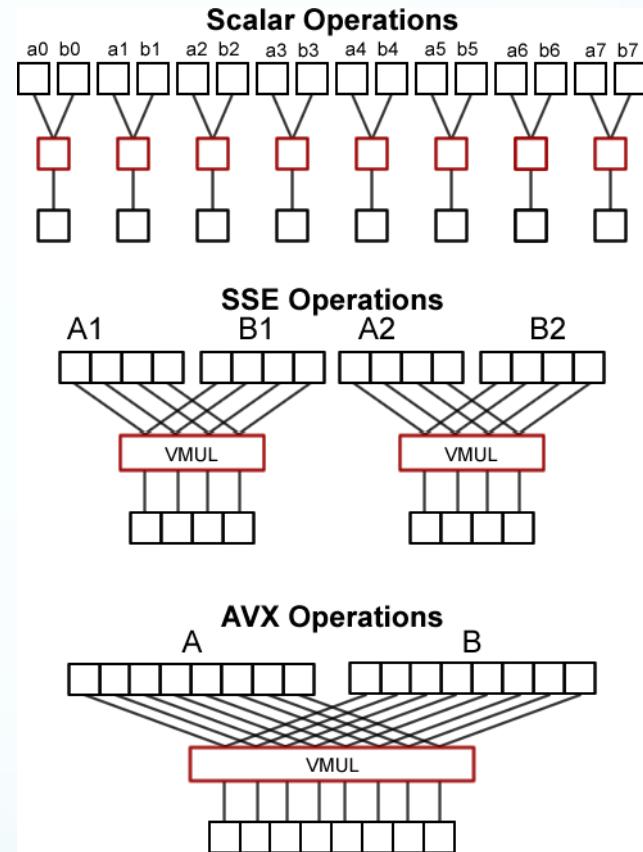
The N-Body simulation

```
for (k=0; k<timesteps; k++) {
    swap(oldbodies, newbodies);
    for (i=0; i<N; i++) {
        tot_force_i[X] = tot_force_i[Y] = tot_force_i[Z] = 0.0;
        for (j=0; j<N; j++) {
            if (j==i) continue;
            //20 floating point operations
            r[X] = oldbodies[j].pos[X] - oldbodies[i].pos[X];
            // analogous for r[Y] and r[Z]
            distSqr = r[X]*r[X] + r[Y]*r[Y] + r[Z]*r[Z] + EPSILON2;
            distSixth = distSqr * distSqr * distSqr;
            invDistCube = 1.0/sqrts(distSixth);
            s = oldbodies[j].mass * invDistCube;
            tot_force_i[X] += s * r[X];
            // analogous for Y and Z
        }
        //24 flops
        dv[X] = dt * tot_force_i[X] / oldbodies[i].mass;
        newbodies[i].pos[X] += dt * ( oldbodies[i].vel[X] + dv[X]/2 );
        newbodies[i].vel[X] = oldbodies[i].vel[X] + dt * dv[X];
        // analogous for Y and Z
    }
}
```

N-Body – sequential algorithm

- Maximum theoretical performances:
9.6 GFLOPs (single core – **SIMD** instructions
– 551\$ CPU, 700\$ icc)
- All-pairs algorithm $O(N^2) \times$ timesteps
- We consider timestep=100 and N=1K, 10K

```
void multiply(float *A, float *B, float *C, int size) {  
    for (int i = 0; i < size; i++) {  
        C[i] = A[i] * B[i];  
    }  
}
```



Compiler	GFLOPs		Efficiency		MFLOPs/US\$	
	1K	10K	1K	10K	1K	10K
gcc (5.1)	1.59	1.67	16.6%	17.4%	3	2
Intel	2.70	5.71	28.1%	59.5%	2	5

N-Body – OpenMP and MPI

```
#pragma omp parallel for private(j, r, distSqr,
    distSixth, invDistCube, s, tot_force_i, dv)

for (i=0; i<N; i++) {
    tot_force_i[X] = tot_force_i[Y] = tot_force_i[Z] = 0.0;
    for (j=0; j<N; j++) {
        if (j==i) continue;
        //20 floating point operations
        r[X] = oldbodies[j].pos[X] - oldbodies[i].pos[X];
        // analogous for r[Y] and r[Z]
        distSqr = r[X]*r[X] + r[Y]*r[Y] + r[Z]*r[Z] + EPSILON2;
        distSixth = distSqr * distSqr * distSqr;
        invDistCube = 1.0/sqrta(distSixth);
        s = oldbodies[j].mass * invDistCube;
        tot_force_i[X] += s*r[X];
        // analogous for tot_force_i[Y] and tot_force_i[Z]
    }
    bnum = N % numprocs;
    bstart=(N/numprocs*id)      + ((id>=bnum) ? bnum : id);
    bstop =(N/numprocs*(id+1)) + ((id>=bnum) ? bnum-1 : id);
    dv[X] = d[newbodies];
    newbodies++;
    MPI_Barrier(MPI_COMM_WORLD); //for timing purpose
    for (k=0; k<num_steps; k++) {
        swap(oldbodies, newbodies);
        for (i=bstart; i<=bstop; i++) { ... }
        MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, newbodies, ...);
    }
}
```

N-Body – OpenMP and MPI

- Maximum theoretical performances:
57.6 GFLOPs (1 CPU – 6 cores)
115 GFLOPs (2 CPUs – 12 cores)
- Intel compiler only (cost disregarded)
- OpenMP is easier
- Think parallel

Compiler	GFLOPs		Efficiency		MFLOPs/US\$	
	1K	10K	1K	10K	1K	10K
OpenMP - 1 cpu	25.49	33.1	44.3%	57.5%	23	30
OpenMP - 2 cpus	35.50	63.24	30.8%	54.9%	32	57
MPI - 1 cpu	27.21	29.12	47.2%	50.6%	25	26
MPI - 2 cpus	50.27	59.93	43.6%	52.0%	46	54

N-Body – CUDA and OpenACC

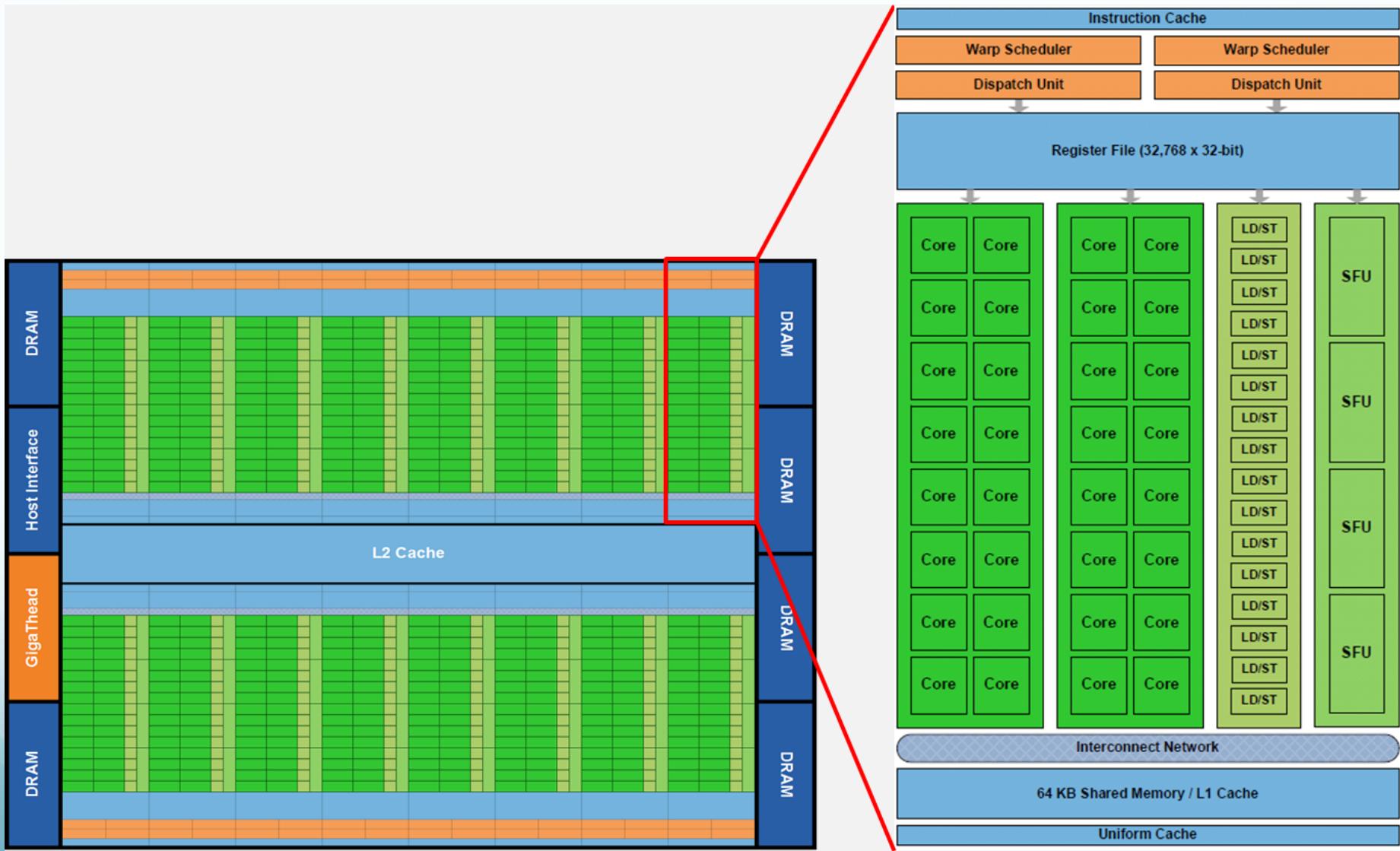
```
for (k=0; k<timesteps; k++) {
    swap(oldbodies ,newbodies );
    for (i=0; i<N; i++) {
        tot_force_i[X] = tot_force_i[Y] = tot_force_i[Z] = 0.0;
        for (j=0; j<N; j++) {
            if (j==i) continue;
            //20 floating point operations
            distSixth = rsqrtf(distSqr);
            invDistCube = distSixth * distSixth * distSixth;
            distSixth = distSqr * distSqr * distSqr;
            invDistCube = 1.0/sqrtf(distSixth);
            s = oldbodies[j].mass * invDistCube;
            tot_force_i[X] += s * r[X];
            // analogous for Y and Z
        }
        //24
        #pragma acc data copy(bodies1[0:N], bodies2[0:N])
        dv[X]           copyin(bodiesvel[0:N], bodiesm[0:N], dt)
        newbo
        for (k=0; k<num_steps; k++) {
            newbo
            #pragma acc kernels loop independent
            for (i=0; i<num_bodies; i++) {
                tot_force_x = 0.0; ...
                #pragma acc loop independent
                reduction(:tot_force_x ,tot_force_y ,tot_force_z)
        }
    }
}
```

N-Body – CUDA and OpenACC

- Maximum theoretical performances:
1581 GFLOPs (1 GPU – 512 cores – 499 US\$)
- OpenACC is much much easier
- Performances with and w/o *fastmath* (rsqrtf function), N=10K

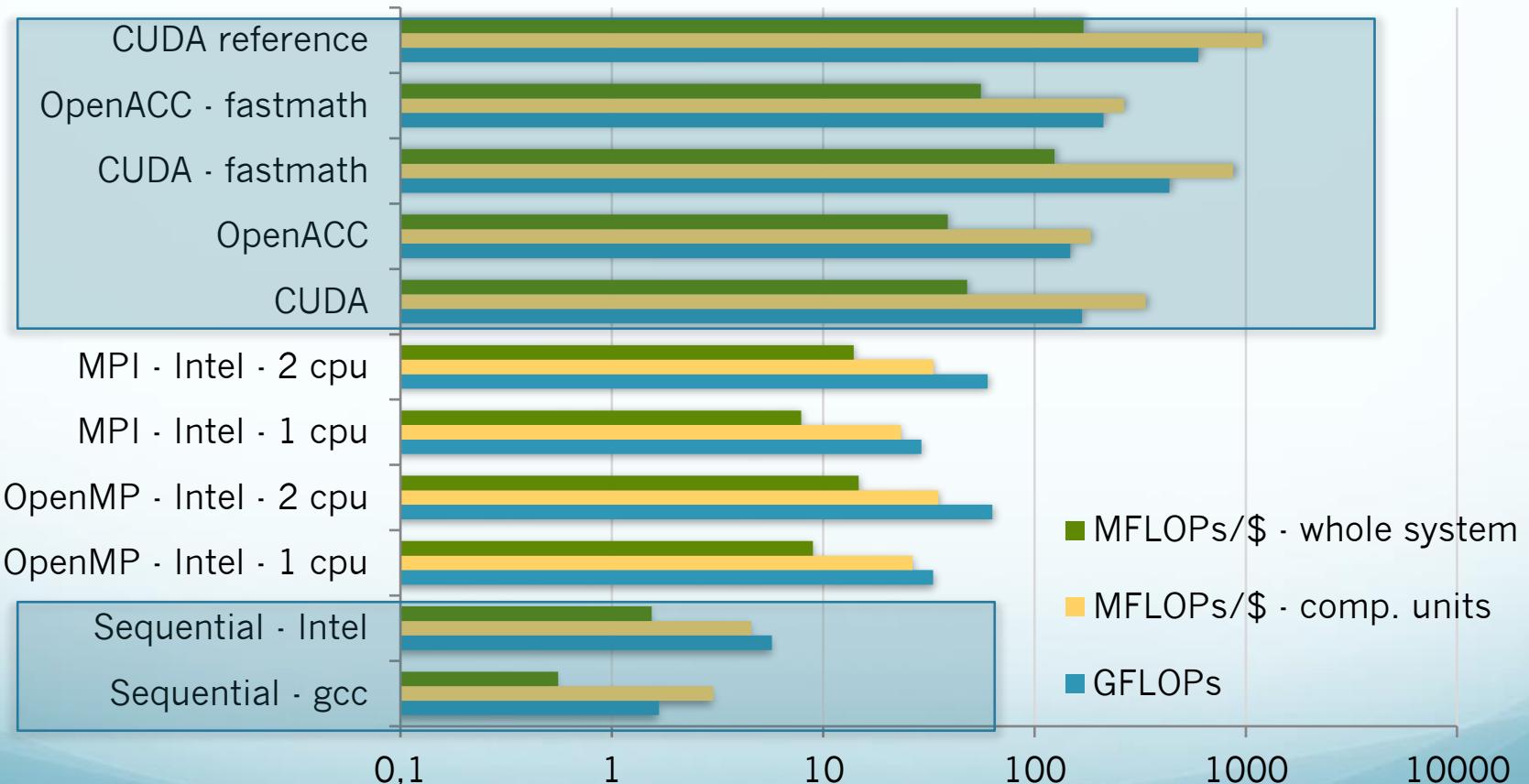
Algorithm	GFLOPs	Efficiency	MFLOPs/US\$
CUDA	167.71	10.6%	336
OpenACC	147.57	9.3%	185
CUDA - fastmath	434.46	27.5%	871
OpenACC - fastmath	211.80	13.4%	265
CUDA reference	597.11	37.8%	1197

N-Body – CUDA and OpenACC



Performance/Price (10K)

Programmers and tools make the difference!



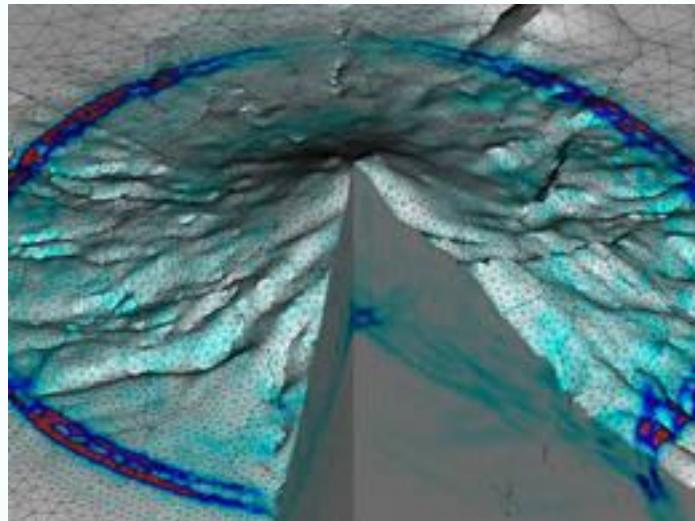
A real-world example



TOP 10 Sites for June 2012

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	16,324.8	20,132.7	7,890
2	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
3	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,162.4	10,066.3	3,945
4	Leibniz Rechenzentrum Germany	SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147,456	2,897.0	3,185.1	3,423

SeisSol Earthquake simulation SW



The extensive optimization and the complete parallelization of the 70,000 lines of SeisSol code results in a peak performance of up to 1.42 petaflops.

This corresponds to 44.5 percent of Super MUC's theoretically available capacity

Name	MPI	# cores	Description	TFlop/s/island	TFlop/s max
Linpack	IBM	★ 128000	TOP500	161	2560
Vertex	IBM	★ 128000	Plasma Physics	15	245
GROMACS	IBM, Intel	★ 64000	Molecular Modelling	40	110
SeisSol	IBM	★ 64000	Geophysics	31	95
waLBerla	IBM	★ 128000	Lattice Boltzmann	5.6	90
LAMMPS	IBM	★ 128000	Molecular Modelling	5.6	90
APES	IBM	★ 64000	CFD	6	47
BQCD	Intel	★ 128000	Quantum Physics	10	27

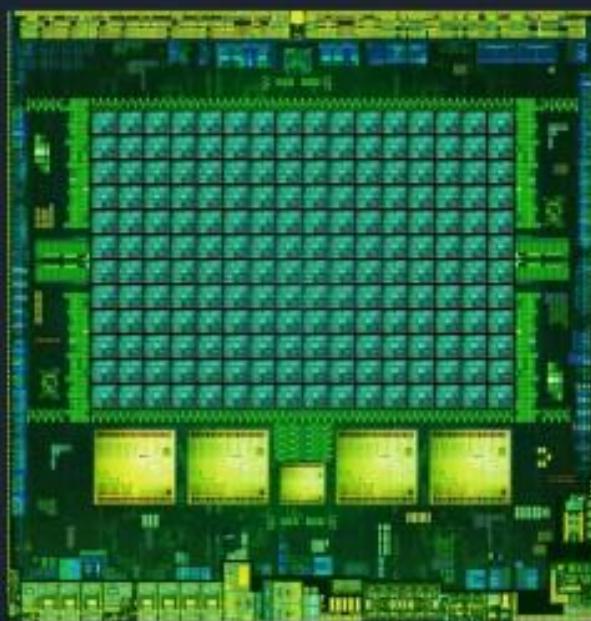
Aspects we will not talk about

- We are not green: 1 week of molecular dynamics simulation on 512 cores = 3200 kWh, corresponding to

- 1600 CO₂ kg
- 340 € energy bill
- 13000 km by car



- A national supercomputing facility has a yearly CO₂ footprint comparable to a takeoff of SATURN V



NVIDIA TEGRA K1

CPU: 2.2GHz 4+1 A15

GPU: 192 核 Kepler

CPU
4+1
A15

GPU
192
Kepler

TEGRA X1 MAXWELL GPU

- ▶ 2x performance vs Tegra K1
- ▶ 2x perf/watt vs Tegra K1
- ▶ 2 SM
- ▶ 256 CUDA Cores

	Tesla K40 + CPU	Nvidia Tegra K1	
Single Precision Peak	4.2 TeraFlops	326 GFlops	13
Single Precision SGEMM	3.8 TeraFlops	290 GFlops	
Memory	12GB @ 288GB/s	2GB @ 14.9GB/s	
Power (CPU + GPU)	~ 385Watt	<11Watts	
Performance Per Watt	10SP GFlops Per Watt	26SP GFlops Per Watt	35

\$ 4,000

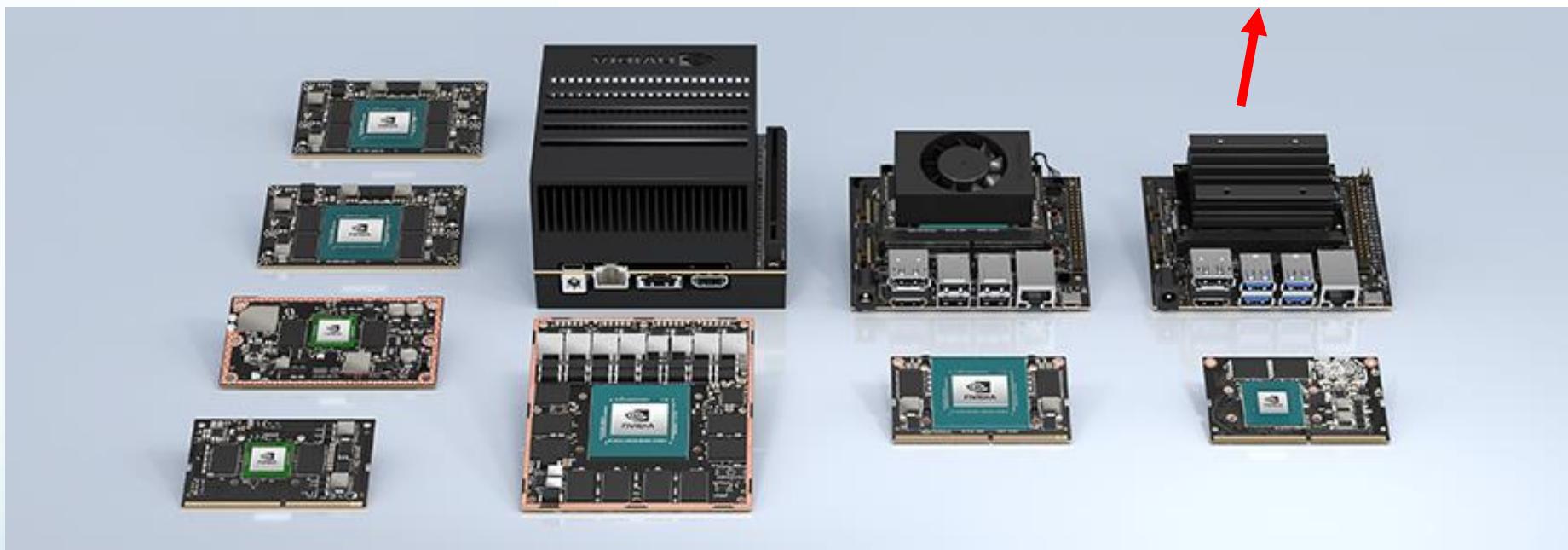
\$ 200

20

Today

[VIEW TECHNICAL SPECIFICATIONS >](#)

GPU	128-core NVIDIA Maxwell™
CPU	Quad-core ARM® A57 @ 1.43 GHz
Memory	2 GB 64-bit LPDDR4 25.6 GB/s



Nvidia Jetson Dev Kit: 59 \$, 472 GFLOPs FP16, 10 W, 45x70 mm

<https://developer.nvidia.com/embedded/jetson-modules>

Example: Functional Magnetic Resonance Imaging

- The science on that is pretty well established. They knew how to take the data that was coming from the MRI, and they could compute on it and create a model of what's going on inside the brain. But in 2012, when we started the project, they estimated it would take 44 years on their cluster
- they parallelized their code and saw huge increases in performance
- But they also looked at it algorithmically with machine learning and AI,
- They put it all together and ended up with a 10,000X increase in performance. They went from something requiring a supercomputing project at a national lab to something that could be done clinically inside a hospital in a couple of minutes.

<https://www.princeton.edu/news/2017/02/23/princeton-intel-collaboration-breaks-new-ground-studies-brain>

<https://www.hpcwire.com/2017/06/08/code-modernization-bringing-codes-parallel-age/>

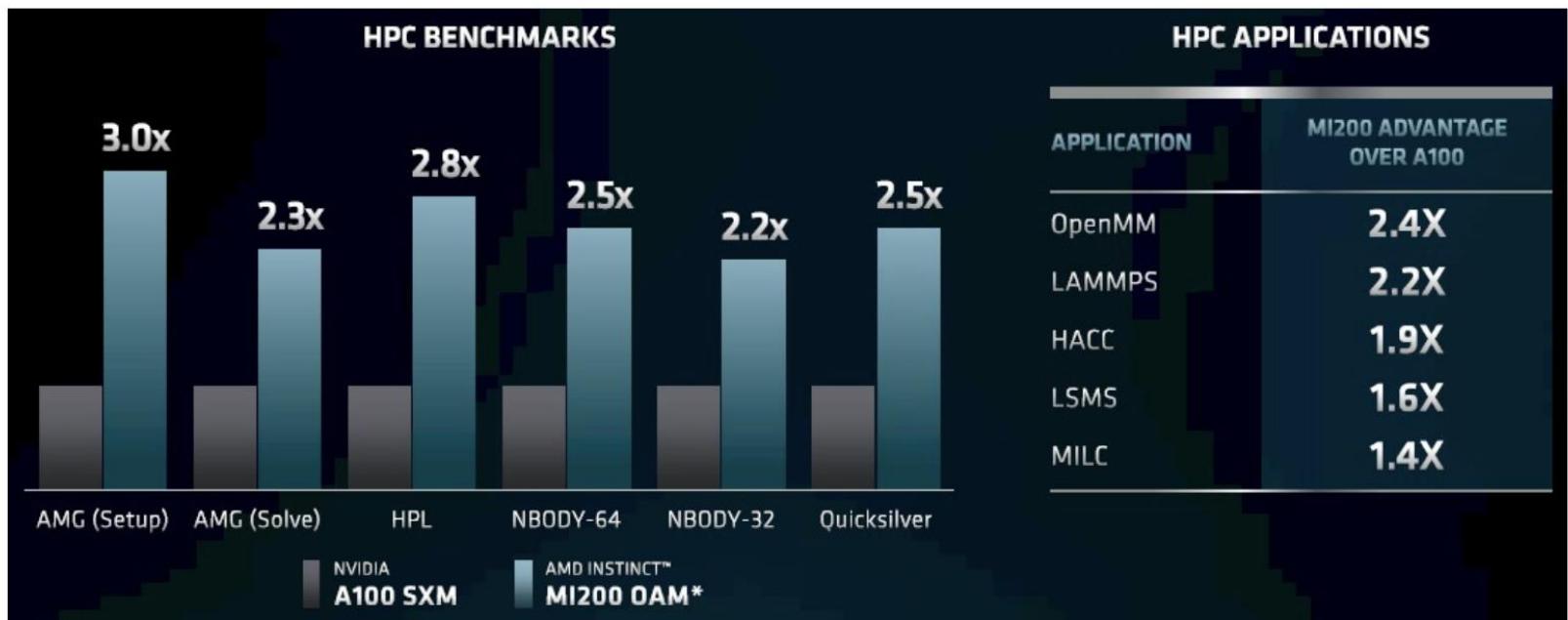
Again on SW

https://www.nextplatform.com/2021/12/06/stacking-up-amd-mi200-versus-nvidia-a100-compute-engines/?mc_cid=11aeb90192&mc_eid=e50c89e962

PEAK PERFORMANCE	A100	MI200*	INSTINCT™ ADVANTAGE
FP64 VECTOR	9.7 TF	47.9 TF	4.9X
FP32 VECTOR	19.5 TF	47.9 TF	2.5X
FP64 MATRIX	19.5 TF	95.7 TF	4.9X
FP32 MATRIX	N/A	95.7 TF	N/A
FP16, BF16 MATRIX	312 TF	383 TF	1.2X
MEMORY SIZE	80 GB	128 GB	1.6X
MEMORY BANDWIDTH	2.0 TB/s	3.2 TB/s	1.6X

Again on SW

What matters, of course, is the performance of the MI200 versus the A100 on HPC benchmarks and real HPC applications. McCredie offered up this chart as food for thought:

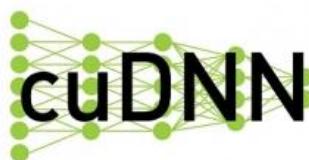
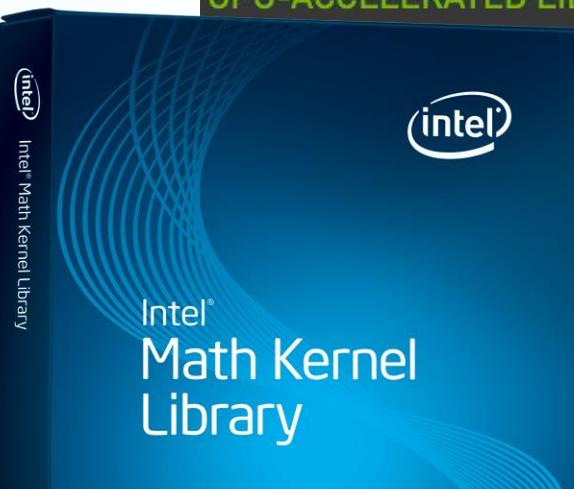


As you can see, the performance numbers are definitely in favor of the Aldebaran GPUs, each half of which has a little bit more performance of the whole A100 on common HPC benchmarks and a little less than that on the HPC applications shown on the right part of this table. The gap is not as big as raw feeds and speeds show, and we think that has to do with the maturity of the compilers and math libraries in the ROCm stack from AMD for its own GPUs versus the CUDA stack from Nvidia for its own GPUs. There is \$100 million in non-recurring engineering funds in the Frontier system alone to try to close some of that ROCm-CUDA gap.

Libraries

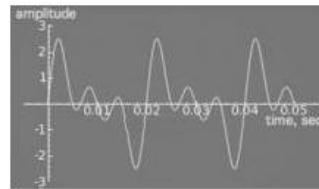
GPU-ACCELERATED LIBRARIES

Your application can be as easy as simply calling a library function. Intel® provides GPU-accelerated, high performance libraries available today.



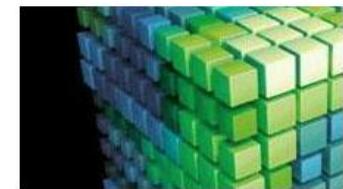
cuDNN

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks, it is designed to be integrated into higher-level machine



cuFFT

NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop

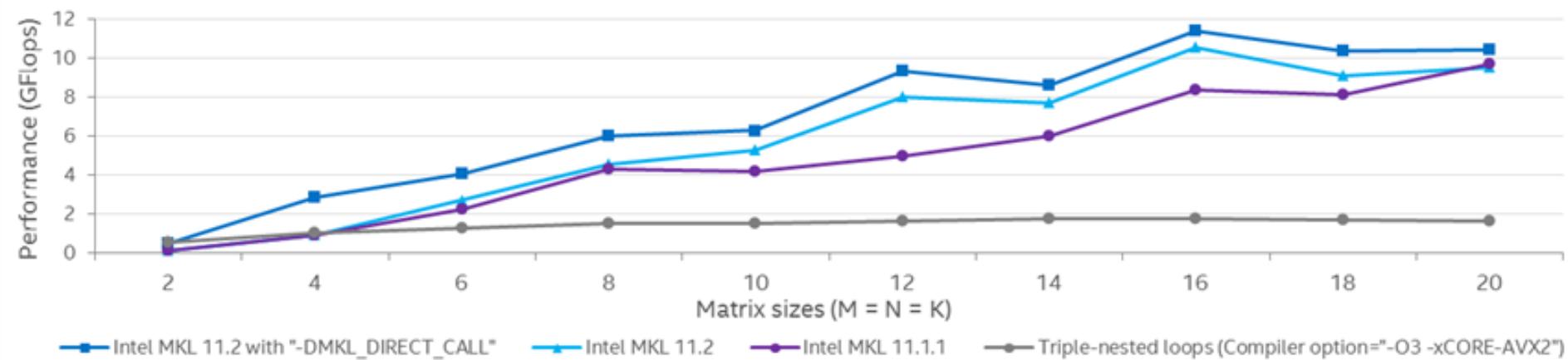


cuBLAS-XT

cuBLAS-XT is a set of routines which accelerate Level 3 BLAS [Basic Linear Algebra Subroutine] calls by spreading work across more than one

Faster Small Matrix Multiplication using Intel® MKL

For 4x4 to 20x20 matrices, S/DGEMM, Single thread, Intel® Xeon® Processor E5-2697v3



— Intel MKL 11.2 with "-DMKL_DIRECT_CALL"

— Intel MKL 11.2

— Intel MKL 11.1

— Triple-nested loops (Compiler option="-O3 -xCORE-AVX2")

Conclusions

The efficient exploitation of current heterogeneous HPC solutions require good understanding of HW and SW features (architectures, instructions sets, sdk, ...)

- Not only HW
- Skilled developers
- State-of-the-art software libraries and programming tools.

Good tools and developers are worth the money

The course aims at presenting the basics to let you became good HPC developers!

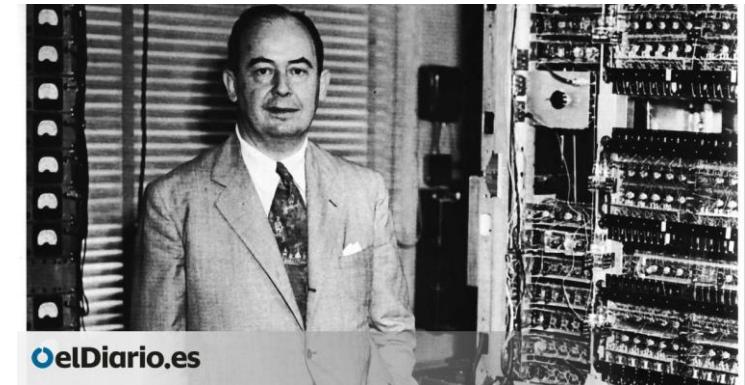
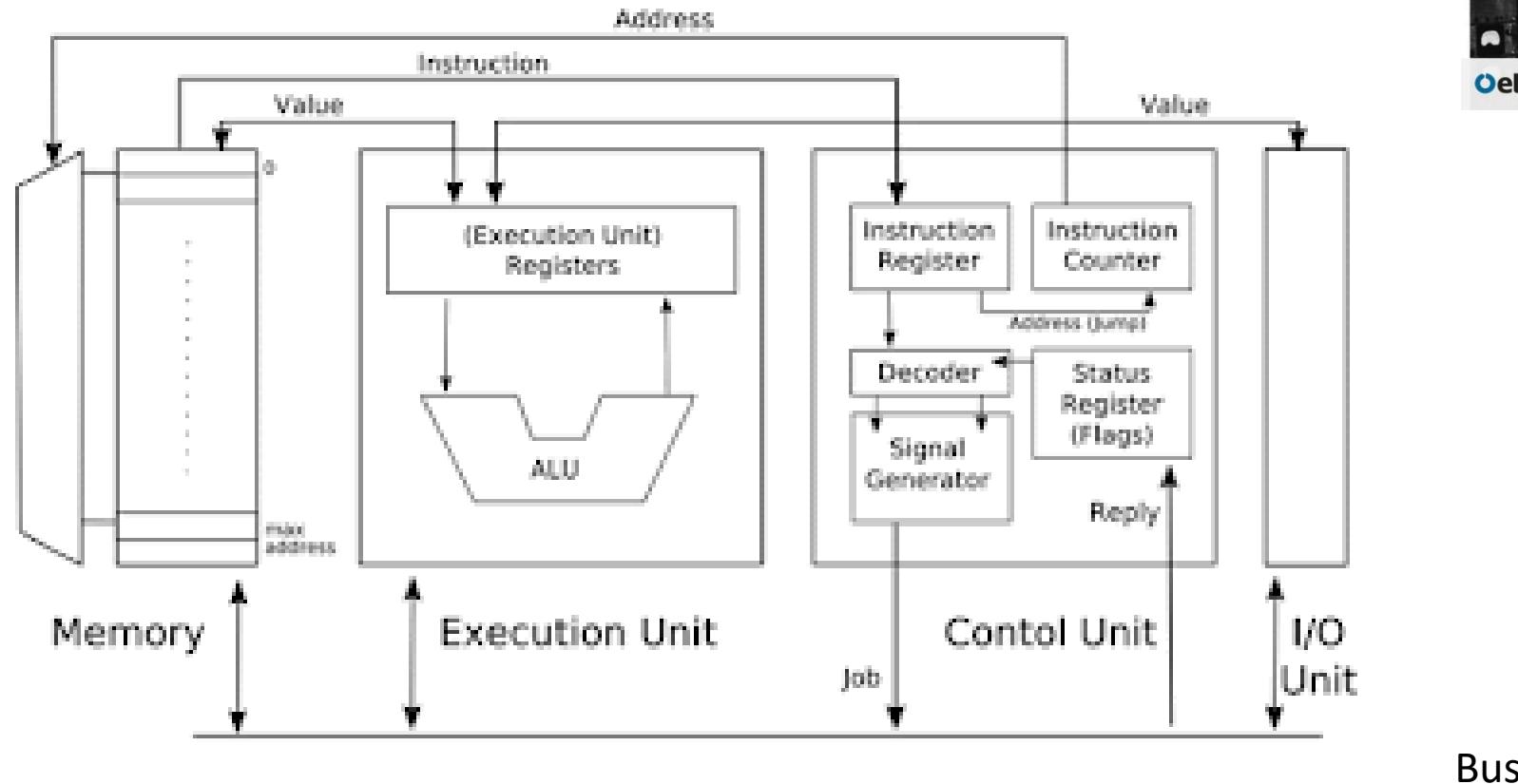
Single Processor Computing

And a first set of basic concepts every computer scientists should know

Introduction

- While to some scientists the abstract fact of the existence of a solution is enough, in HPC we actually want that solution, and preferably yesterday
 - efficiency of both algorithms and hardware
 - arithmetic precision
 - the difference between two algorithmic approaches can enable hardware-specific optimizations
- Therefore, in order to write efficient scientific codes, it is important to understand your computer architecture.
- The difference in speed between two codes that compute the same result can range from a few percent to orders of magnitude, depending only on factors relating to how well the algorithms are coded for the processor architecture.

The Von Neumann Architecture



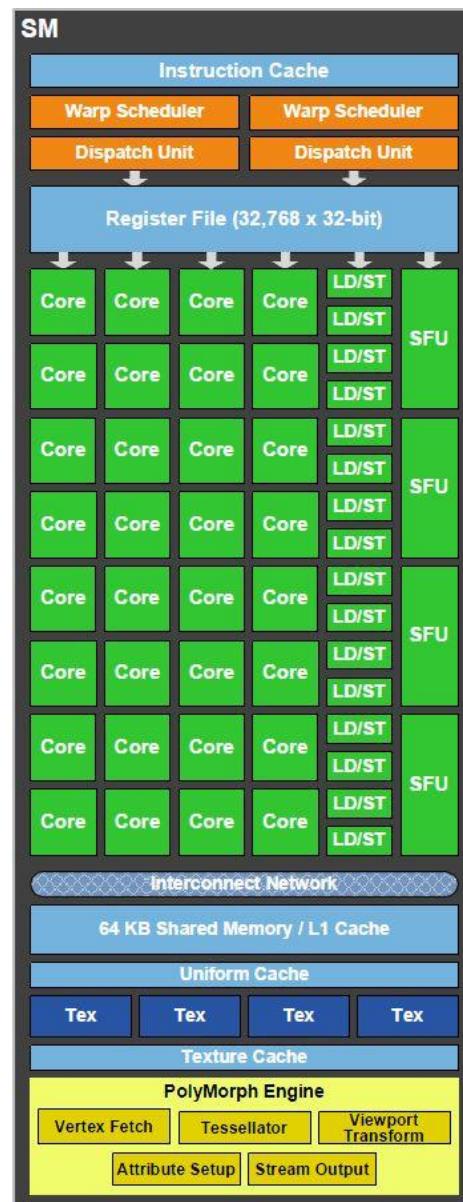
<http://abelgo.cn/cs101/papers/Neumann.pdf>

Example

- Many NVIDIA GPUs are equipped with Special Function Units (SFUs) that execute transcendental/intrinsics instructions such as sin, cosine, reciprocal, and square root
- PROS: each SFU executes one transcendental instruction per thread, per clock, wrt to the traditional instructions that require several FMAD instructions
- CONS: limited precision

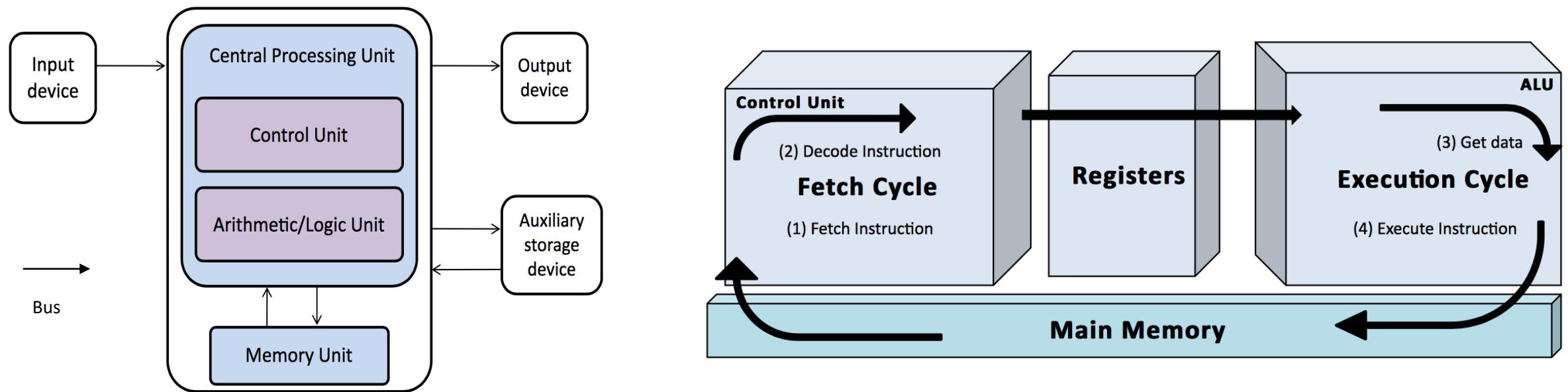
Algorithm	GFLOPs
CUDA	167.71
OpenACC	147.57
CUDA - fastmath	434.46
OpenACC - fastmath	211.80
CUDA reference	597.11

Function	Error bounds
<code>_sinf(x)</code>	For x in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
<code>_cosf(x)</code>	For x in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and larger otherwise.
<code>_sincosf(x, sptr, cptr)</code>	Same as <code>_sinf(x)</code> and <code>_cosf(x)</code> .
<code>_tanf(x)</code>	Derived from its implementation as <code>_sinf(x) * (1/_cosf(x))</code> .
<code>_powf(x, y)</code>	Derived from its implementation as <code>exp2f(y * log2f(x))</code> .



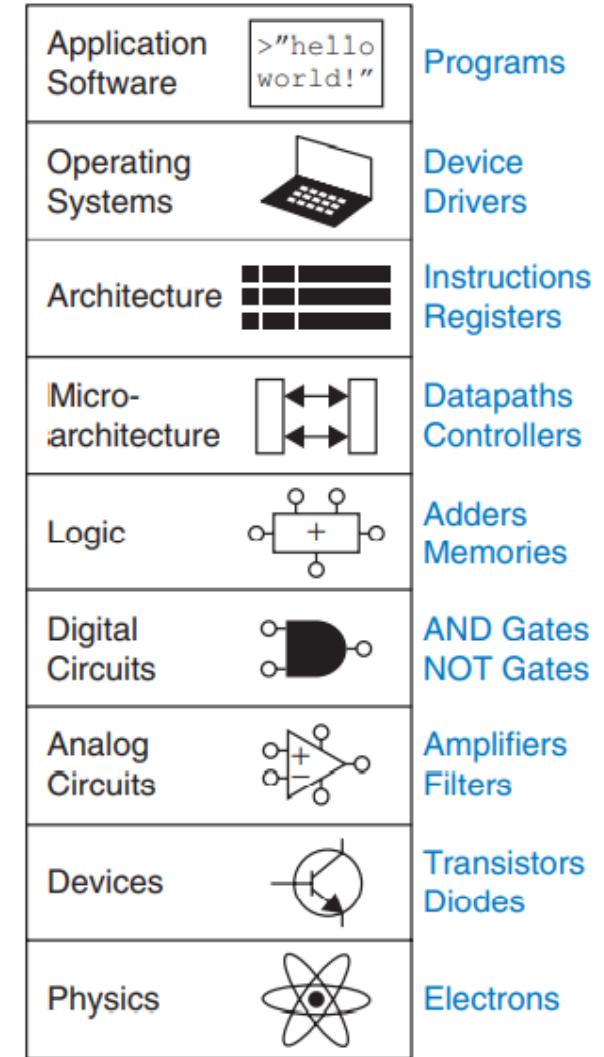
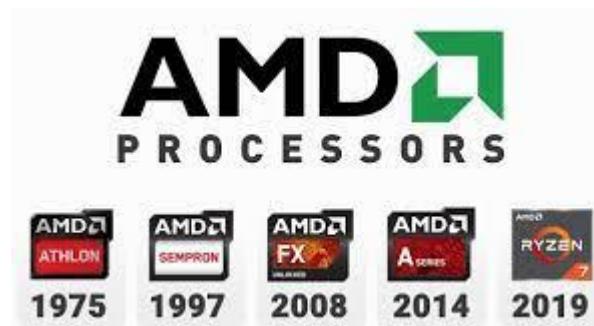
von Neumann Architecture

- Present computers have many aspects in common.
- Their architecture is based on an undivided memory that stores both program and data ('stored program'), and a processing unit that executes the instructions, operating on the data in 'fetch, execute, store cycle'
- We can say that a computer is the HW implementation of that cycle, and it represents an universal executor of algorithms



Architectures

- But, actually, a computer architecture (e.g. x86) is defined by its
 - instruction set
 - architectural state, i.e. the information necessary to define what a computer is doing (PC + registers).
- A microarchitecture involves combining logic elements to execute the instructions defined by the architecture



Processing instructions

```
void store(double *a, double *b, double *c) {  
    *c = *a + *b;  
}
```

```
gcc -O2 -S -o - store.c or, better  
gcc -O2 -g -Wa,-adhln -fverbose-asm store.c > store.s
```

```
store:  
    movsd (%rdi), %xmm0 # Load *a to %xmm0  
    addsd (%rsi), %xmm0 # Load *b and add to %xmm0  
    movsd %xmm0, (%rdx) # Store to *c  
    ret
```

Each instruction is processed as follows:

- Fetch instruction from memory.
- Decode the instruction and read registers
- Execute the operation OR calculate an address
- Access an operand in data memory (if necessary)
- write the result into a register (if necessary)

Slide on x86 Assembler

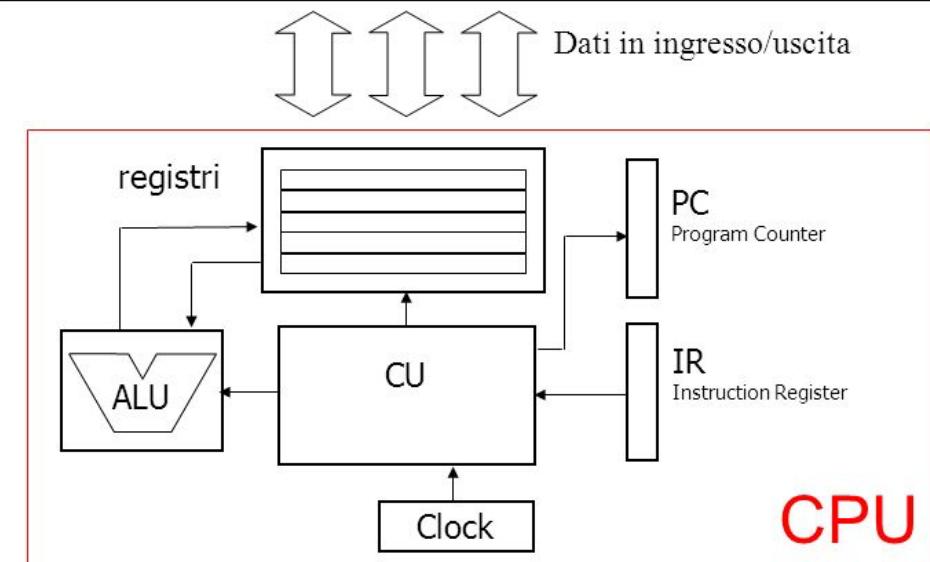
<https://brown-cs0330.github.io/website/index.html>

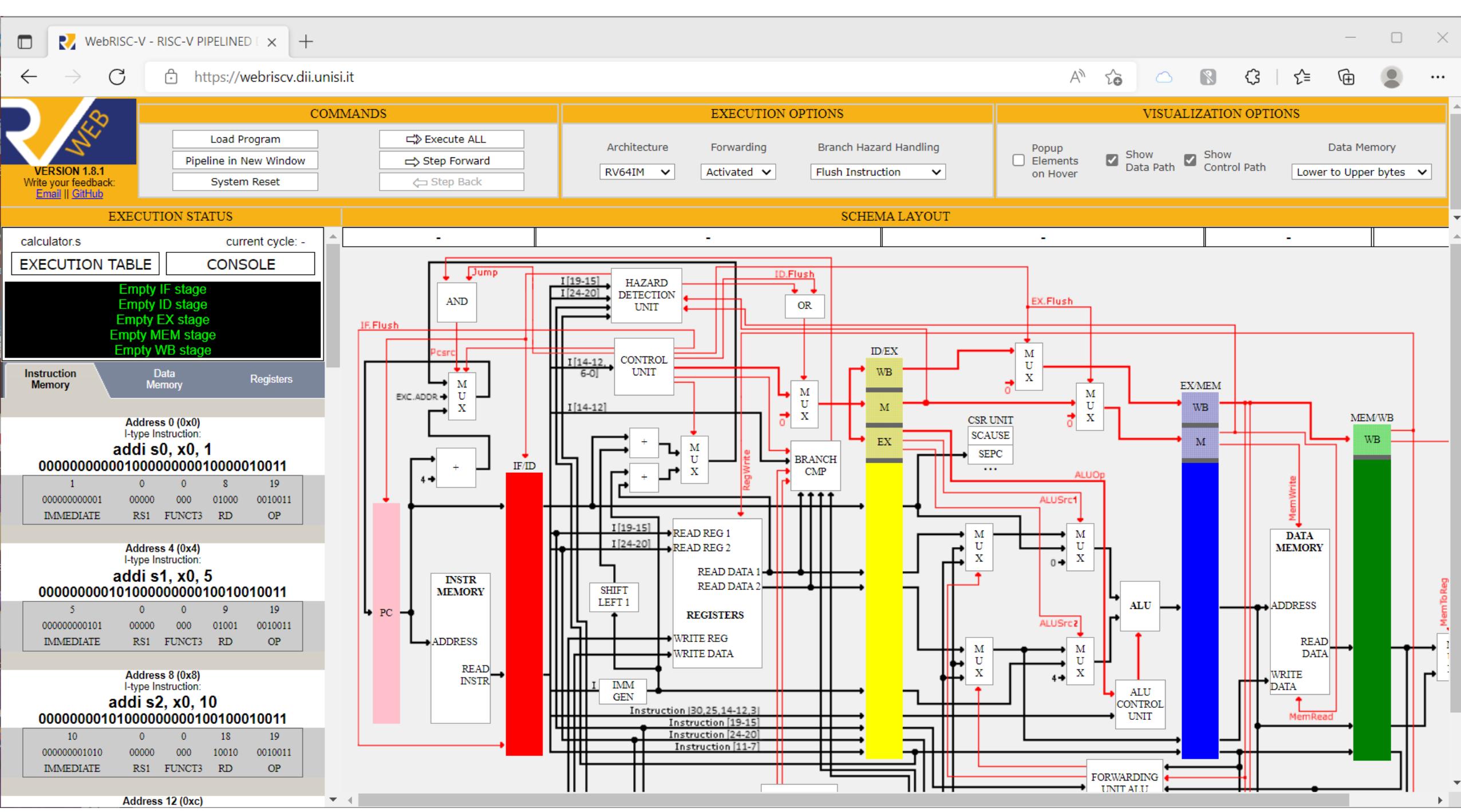
Compiler explorer

<https://godbolt.org>

A bit of assembler <https://stackoverflow.com/questions/5325326/what-is-the-meaning-of-each-line-of-the-assembly-output-of-a-c-hello-world/38290148>

BUS

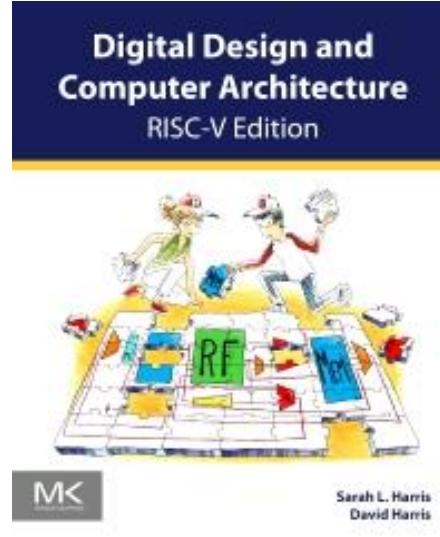




Risc-V Architecture

Table 6.1 RISC-V register set

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments/Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporary registers



Code Example 6.21 USING A FOR LOOP TO ACCESS AN ARRAY

High-Level Code

```
int i;
int scores[200];

for (i = 0; i < 200; i = i + 1)
    scores[i] = scores[i] + 10;
```

RISC-V Assembly Code

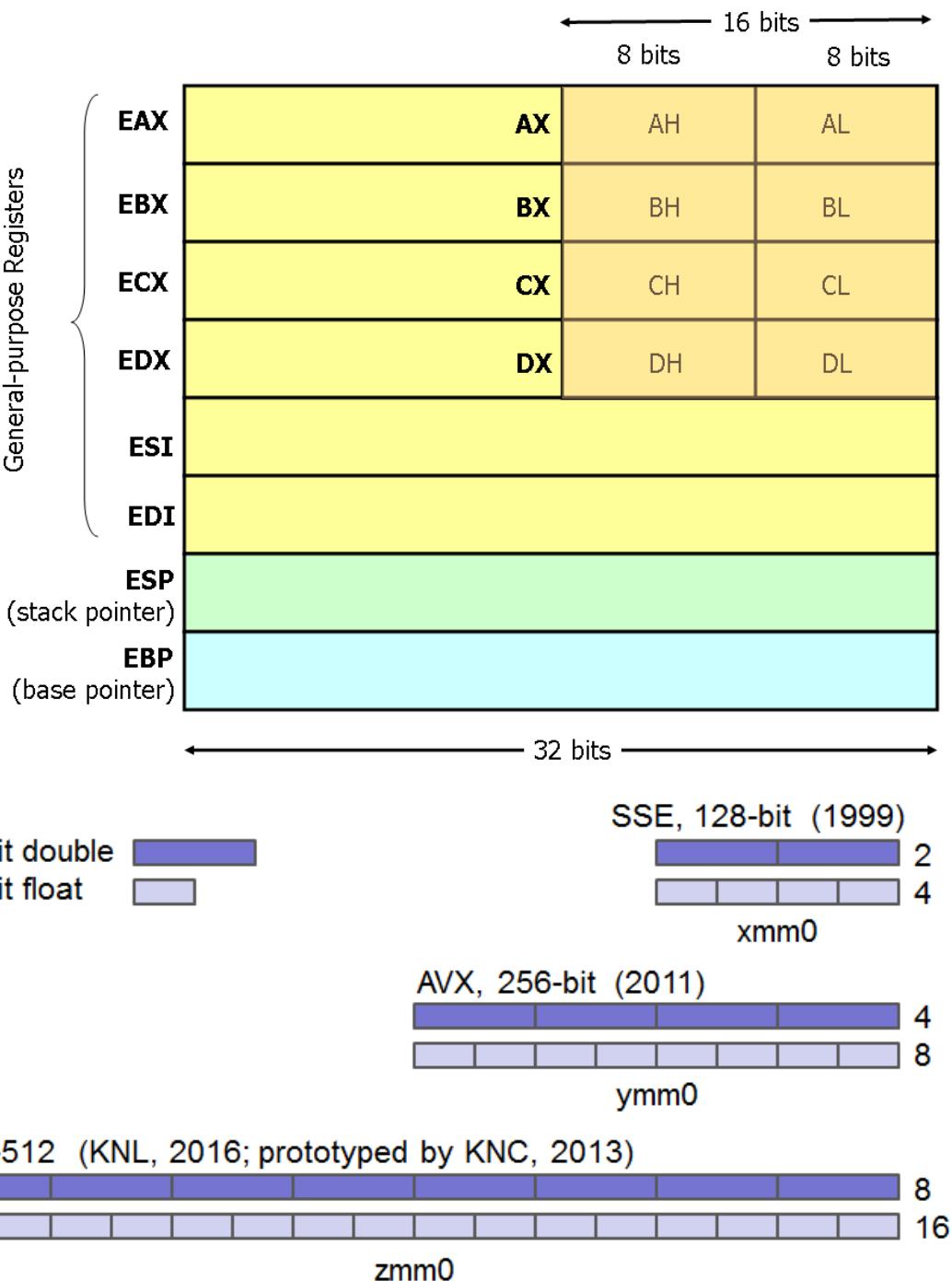
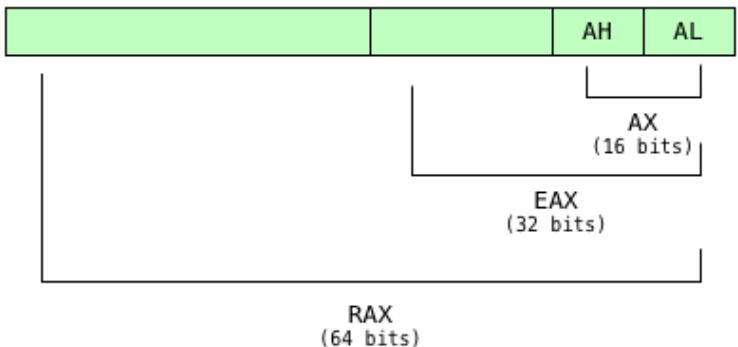
```
# s0 = scores base address, s1 = i

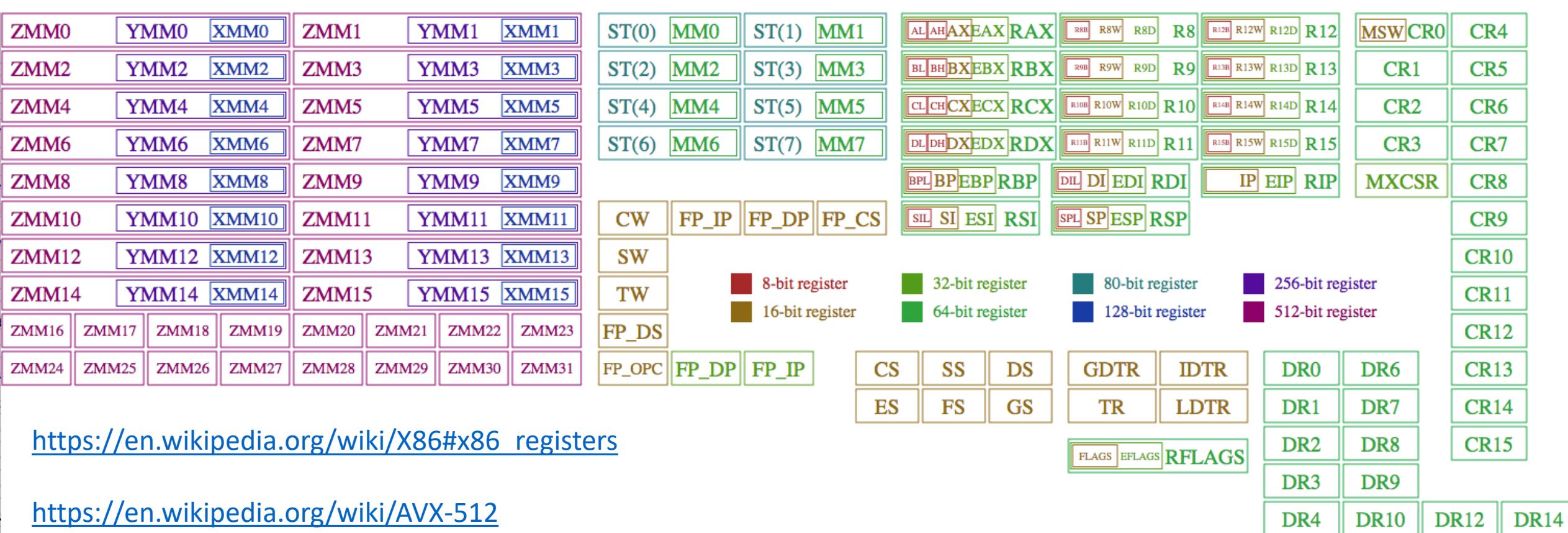
addi s1, zero, 0    # i = 0
addi t2, zero, 200 # t2 = 200

for:
    bge s1, t2, done # if i >= 200 then done
    slli t0, s1, 2    # t0 = i * 4
    add t0, t0, s0    # address of scores[i]
    lw t1, 0(t0)      # t1 = scores[i]
    addi t1, t1, 10   # t1 = scores[i] + 10
    sw t1, 0(t0)      # scores[i] = t1
    addi s1, s1, 1    # i = i + 1
    j for             # repeat
done:
```

X86 Registers

- Small amount of memory internal to a processor
- The registers are what the processor actually operates on
- An operation like $a = b + c$ involves three registers
- Registers have specific names, sizes, a high bandwidth and low latency because they are part of the processor.





https://en.wikipedia.org/wiki/X86_registers

<https://en.wikipedia.org/wiki/AVX-512>

Name	Extension sets	Registers	Types
Legacy SSE	SSE-SSE4.2	xmm0-xmm15	single floats. From SSE2: bytes, words, doublewords, quadwords and double floats.
AVX-128 (VEX)	AVX, AVX2	xmm0-xmm15	bytes, words, doublewords, quadwords, single floats and double floats.
AVX-256 (VEX)	AVX, AVX2	ymm0-ymm15	single float and double float. From AVX2: bytes, words, doublewords, quadwords
AVX-128 (EVEX)	AVX-512VL	xmm0-xmm31 (k1-k7)	doublewords, quadwords, single float and double float. With AVX512BW: bytes and words
AVX-256 (EVEX)	AVX-512VL	ymm0-ymm31 (k1-k7)	doublewords, quadwords, single float and double float. With AVX512BW: bytes and words
AVX-512 (EVEX)	AVX-512F	zmm0-zmm31 (k1-k7)	doublewords, quadwords, single float and double float. With AVX512BW: bytes and words

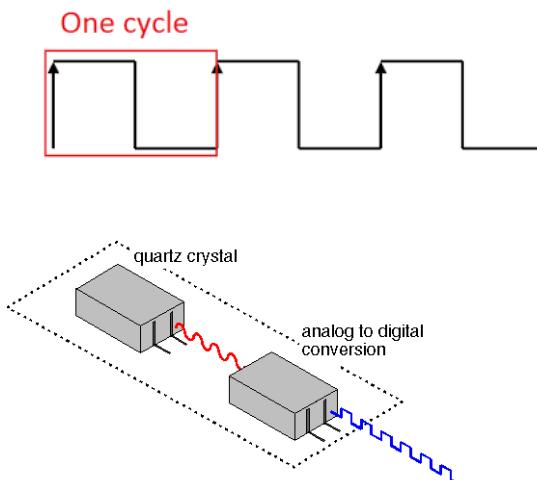
The Clock

The instructions of all modern processors need to be *synchronised* with a timer or *clock*.

The *clock cycle* τ is defined as the time between two adjacent pulses of oscillator that sets the time of the processor.

The number of these pulses per second is known as *clock speed* or *clock frequency*, generally measured in GHz (gigahertz, or billions of pulses per second).

The clock cycle controls the synchronization of operations in a computer: All the operations inside the processor last a multiple of τ .



The quartz crystal generates continuous waves, which are converted into digital pulses.

Processor	τ (ns)	freq (MHz)
CDC 6600	100	10
Cyber 76	27.5	36
IBM ES 9000	9	111
Cray Y-MP C90	4.1	244
Intel i860	20	50
PC Pentium	< 0.5	> 2 GHz
Power PC	1.17	850
IBM Power 5	0.52	1.9 GHz
IBM Power 6	0.21	4.7 GHz

Increasing the clock frequency:

The *speed of light* sets an upper limit to the speed with which electronic components can operate.

Propagation velocity of a signal in a vacuum:
300,000 Km/s = 30 cm/ns

Heat dissipation problems inside the processor. Power consumption varies as the square or cube of the clock frequency.

edge-triggered clocking

- Any values stored in a logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa
- The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

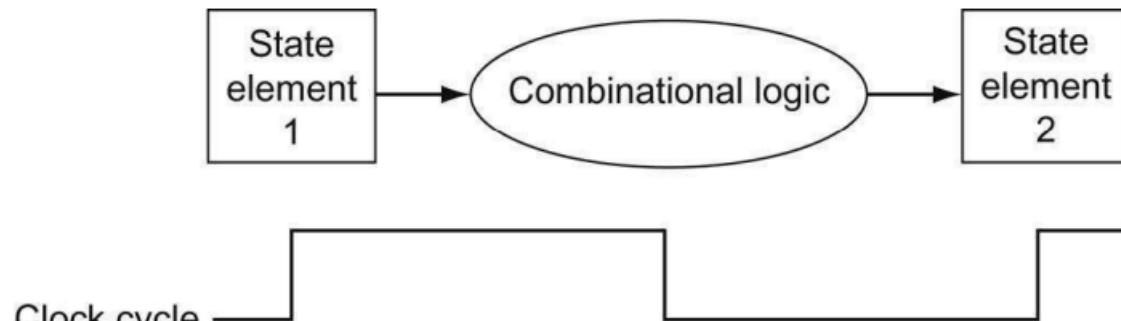


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related.

CPU performance

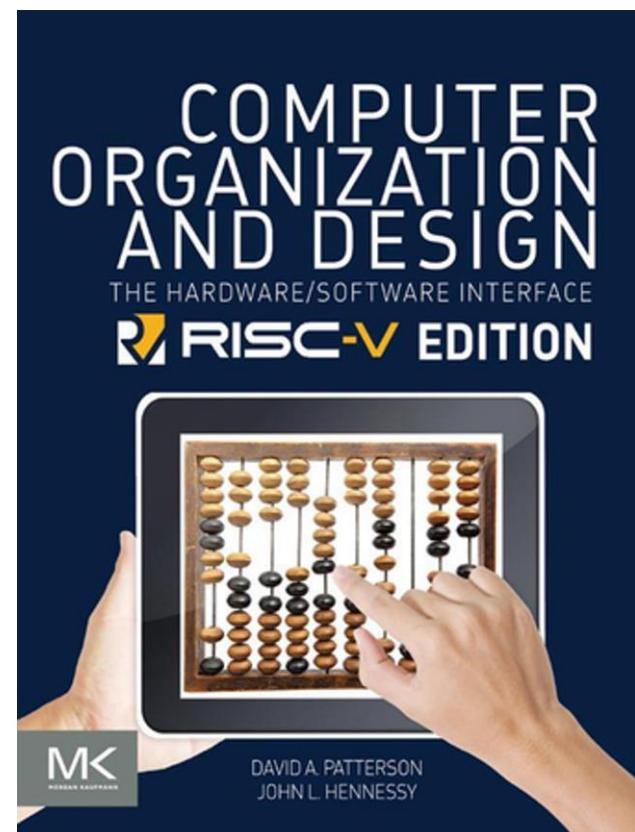
- Execution time: sum of the actual cycles required for each specific instruction
https://www.agner.org/optimize/instruction_tables.pdf
 - Execution time: (number of instructions x average CPI) / clock frequency
 - A program is converted in
 - 1) 4 billion MIPS instructions. The MIPS processor requires, on average, 1.5 Cycles Per Instruction and the clock speed is 1 GHz
 - 2) 2 billion x86 instructions. The x86 processor requires, on average, 6 CPI and the clock speed is 1.5 GHz
- Which is better?

<https://www.anandtech.com/show/3593>

Understanding Program Performance

The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in varied ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.



Moore's Law

- In 1965, Gordon Moore proposed that the number of transistors on a silicon chip would double every year.
- Moore's Law, as it is now known, proved prophetic about the exponential growth of computing power that made much of the modern world possible.
- A transistor produces, amplifies, and directs an electrical signal using three leads, a source, a gate, and a drain.
- When voltage is applied to the gate lead, an incoming current at the source lead will be allowed to pass through to the drain lead. Take the voltage away from the gate lead and the current cannot pass through.
- What this does is produce a way to compute logical values, 1 and 0 in computer terms, based on the whether there is voltage applied to the gate and the source leads.
- Connect the drain lead of a transistor to the source lead or the gate lead of another transistor and suddenly you can start producing incredibly complex logic systems. **And the more transistors you can fit on a chip, the more computationally powerful this network becomes.**

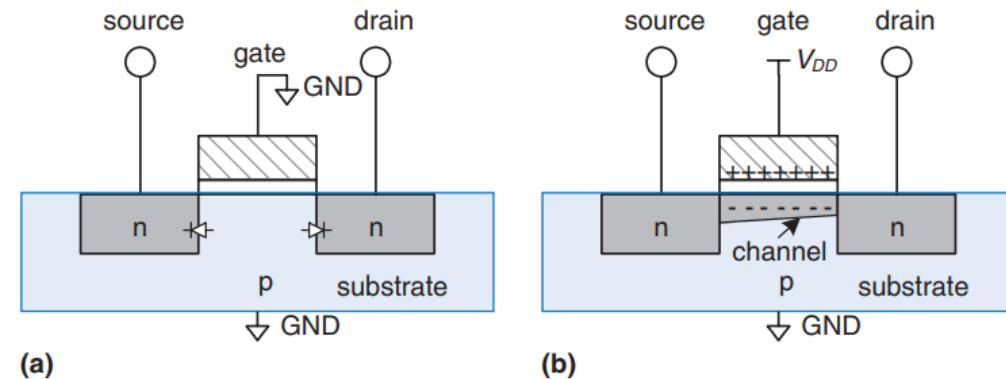
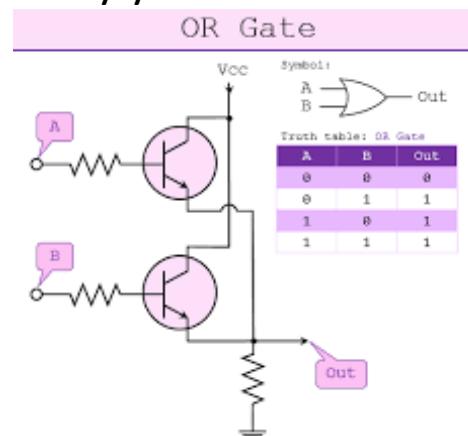


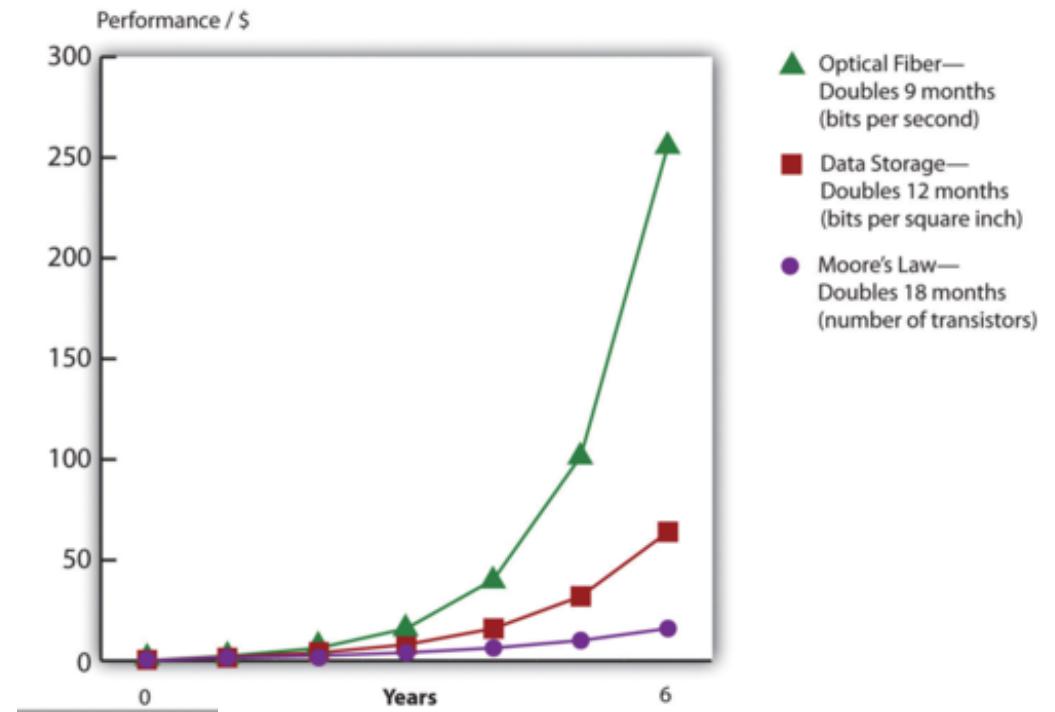
Figure 1.30 nMOS transistor operation

<http://www.moorelaw.org> How a CPU is made <https://www.youtube.com/watch?v=qm67wbB5Gml>
<https://interestingengineering.com/no-more-transistors-the-end-of-moores-law>
<https://www.101computing.net/from-transistors-to-micro-processors/>



Other formulations and atoms

- Butters' law says that the amount of data coming out of an optical fiber is doubling every nine months.
- Kryder's Law says that «Inside of a decade and a half, hard disks had increased their capacity 1,000-fold.



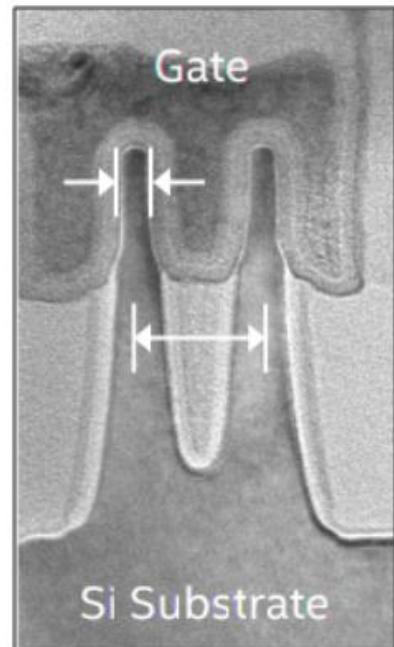
The **atoms** used in silicon chip fabrication are around 0.2nm.

CPU Vendors measures their technology in nm, but that's not the size of individual transistors; that's actually a measure of the distance *between* discrete components on a chip.

<https://www.extremetech.com/computing/97469-is-14nm-the-end-of-the-road-for-silicon-lithography>

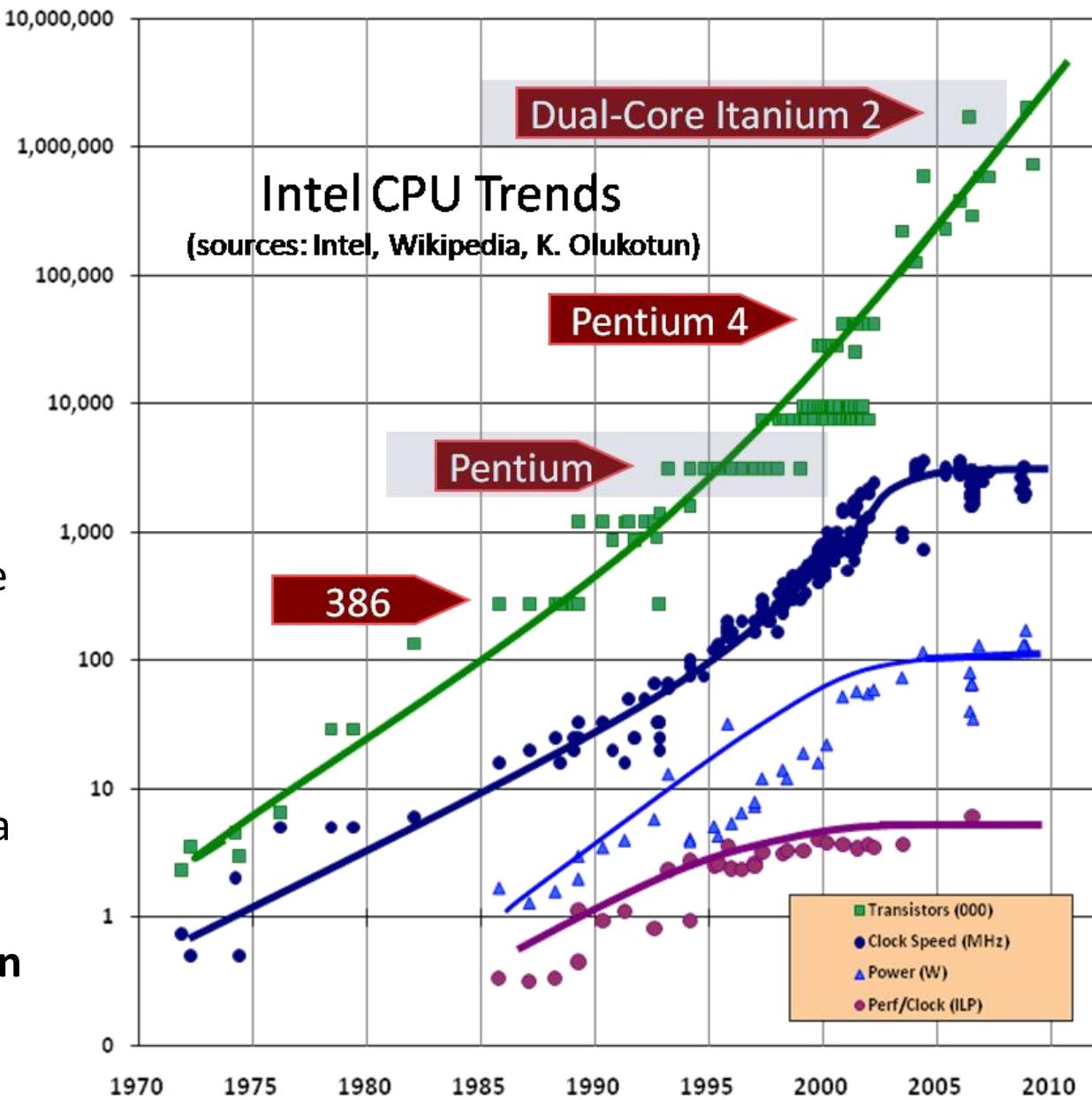
8 nm Fin Width

42 nm Fin Pitch



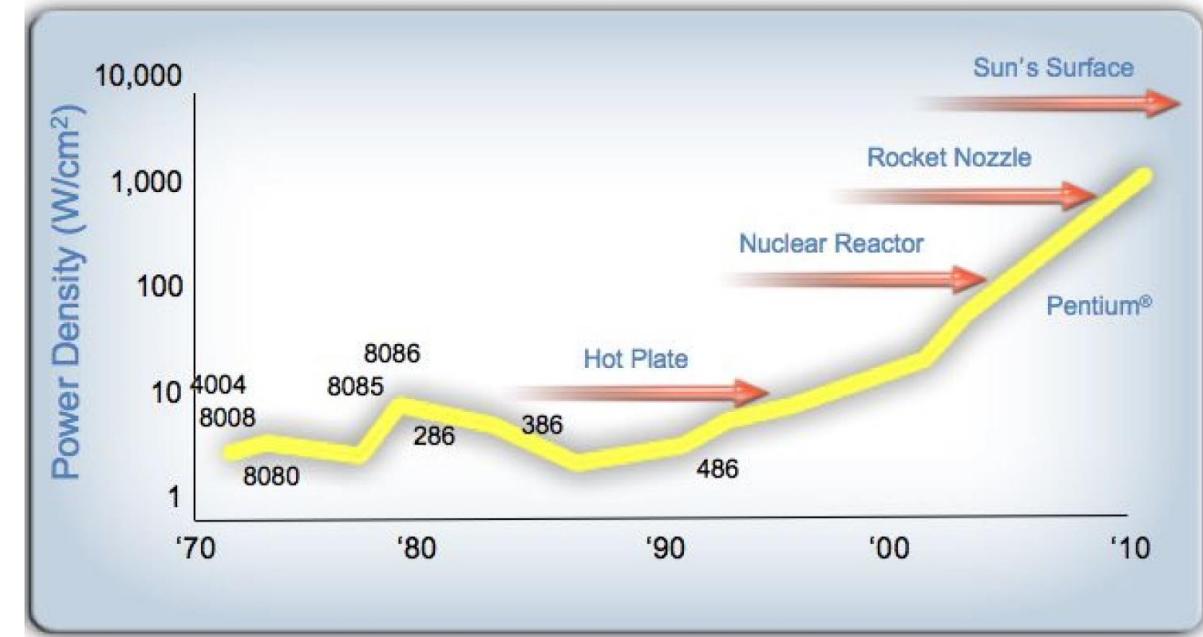
Moore's Law

- The speeds have barely doubled within a 10 year span (i.e. 2000-2010).
- This is because we are looking at the speeds and not the **number of transistors**.
- The speeds over a large number of years for example went between 1.3 and 2.8 GHz, which is barely double but what needs to be kept in mind is that the 2.8 is a QUAD CORE while the 1.3 is a single CORE.
- This means that the actual power of the 2.8 would be found if you **multiply by four** – which would give you a whopping 11.2 which is a far cry from 1.3.
- **The limitation which exists is that once transistors can be created as small as atomic particles, then there will be no more room for growth in the CPU market where speeds are concerned**



Moore's Law is Dead. Long Live Moore's Law!

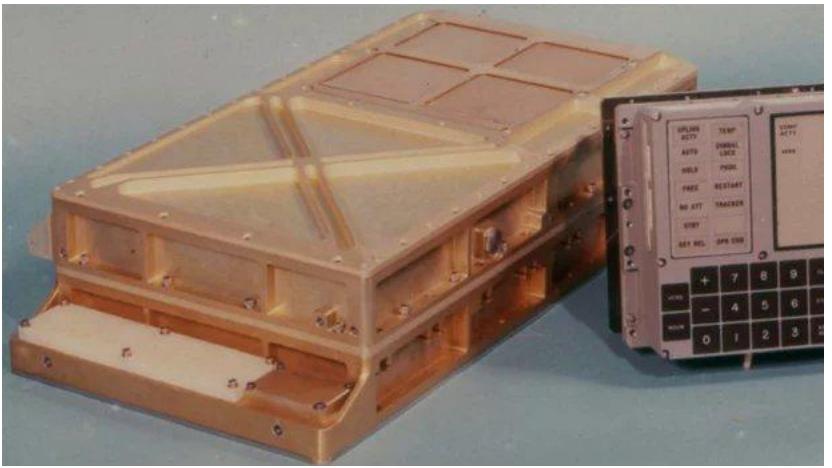
- The frequency of a single core can not be scaled up since this would raise the heat production of the chip too far.
 - They have gotten so small and the channel that carries the electrical current through the transistor cannot always contain it. This generates heat.
 - When the number of transistors doubles, so does the amount of heat they can generate.
 - The cost of cooling is becoming too big.
- The graph shows the heat that a chip would give off, if single-processor trends had continued.
- The computer design run into a power wall , where the sophistication of a single core can not be increased any further (we will see it, e.g. ILP) and **the only way to increase performance is to increase the amount of explicitly visible parallelism**
 - Each core is less powerful than the previous generation's single core design but their number increases



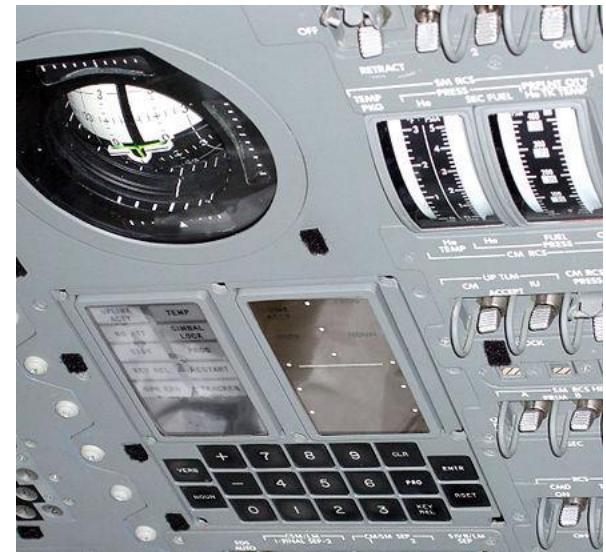
The Free Lunch Is Over

A Fundamental Turn Toward Concurrency in Software

- The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency. (March 2005)
<http://www.gotw.ca/publications/concurrency-ddj.htm>
- Make a CPU ten times as fast, and software will usually find ten times as much to do (*or, in some cases, will feel at liberty to do it ten times less efficiently – see next slide*).
- Most classes of applications have enjoyed free and regular performance gains for several decades, even without releasing new versions or doing anything special, because the CPU manufacturers (primarily) and memory and disk manufacturers (secondarily) have reliably enabled ever-newer and ever-faster mainstream systems.
- Right enough, in the past. But dead wrong for the foreseeable future. **So?**



Apollo Guidance computer, 2 MHz, 72 KB
RAM, 32 KG and about 1 MB di ROM.



The Free Lunch Is Over

Over the past 30 years, CPU designers have achieved performance gains in three main areas:

- 1. Clock speed**

Increasing clock speed is about getting more cycles. Running the CPU faster more or less directly means doing the same work faster.

- 2. Execution optimization**

Optimizing execution flow is about doing more work per cycle. Today's CPUs sport some more powerful instructions, and they perform optimizations that range from the pedestrian to the exotic, including pipelining, branch prediction, executing multiple instructions in the same clock cycle(s), and even reordering the instruction stream for out-of-order execution. These techniques are all designed to make the instructions flow better and/or execute faster, and to squeeze the most work out of each clock cycle by reducing latency and maximizing the work accomplished per clock cycle.

- 3. Cache**

the memory is too slow to load data into the process at the rate the processor can absorb it. Specifically, a single load can take 1000 cycles, while a processor can perform several operations per cycle, therefore huge efforts are paid to reduce the so-called memory wall

This set of slides focuses on 2-3

The Free Lunch Is Over

Present performance growth drivers are:

- **Hyperthreading**

Hyperthreading is about running two or more threads in parallel inside a single CPU by better exploiting it when some threads wait the completion of I/O operations . A limiting factor, however, is that although a hyper-threaded CPU has some extra hardware including extra registers, it still has just one cache, one integer math unit, one FPU, and in general just one each of most basic CPU features. Hyperthreading is sometimes cited as offering a 5% to 15% performance boost for reasonably well-written multi-threaded applications, or even as much as 40% under ideal conditions for carefully written multi-threaded applications. That's good, but it's hardly double, and it doesn't help single-threaded applications.

- **Multicore**

Multicore is about running two or more actual CPUs on one chip. The performance gains ideally are about the same as having a true dual-CPU system (only the system will be cheaper because the motherboard doesn't have to have two sockets and associated "glue" chipery) for reasonably well-written multi-threaded applications. Not single-threaded ones.

- **Cache**

On-die cache sizes can be expected to continue to grow. Of these three areas, only this one will broadly benefit most existing applications. The continuing growth in on-die cache sizes is an incredibly important and highly applicable benefit for many applications, simply because space is speed.

- **Multicore(s) and related parallel programming techniques represent the main topic of this course**

Myths and Realities: 2 x 3GHz < 6 GHz

- So a dual-core CPU that combines two 3GHz cores practically offers 6GHz of processing power. Right? **Wrong**.
- **FIRST: memory**

Even having two threads running on two physical processors doesn't mean getting two times the performance. Similarly, most multi-threaded applications won't run twice as fast on a dual-core box. They should run faster than on a single-core CPU; the performance gain just isn't linear, that's all.
- Why not? First, there is coordination overhead between the cores to ensure cache coherency (a consistent view of cache, and of main memory) and to perform other handshaking. Today, a two- or four-processor machine isn't really two or four times as fast as a single CPU even for multi-threaded applications. The problem remains essentially the same even when the CPUs in question sit on the same die.
- **Second: coding**

Unless the two cores are running different processes, or different threads of a single process that are well-written to run independently and almost never wait for each other, they won't be well utilized.
- (Despite this, I will speculate that today's single-threaded applications as actually used in the field could actually see a performance boost for most users by going to a dual-core chip, not because the extra core is actually doing anything useful, but because it is running the adware and spyware that infest many users' systems and are otherwise slowing down the single CPU that user has today. I leave it up to you to decide whether adding a CPU to run your spyware is the best solution to that problem.)
- If you're running a single-threaded application, then the application can only make use of one core. There should be some speedup as the operating system and the application can run on separate cores, but typically the OS isn't going to be maxing out the CPU anyway so one of the cores will be mostly idle. (Again, the spyware can share the OS's core most of the time.)

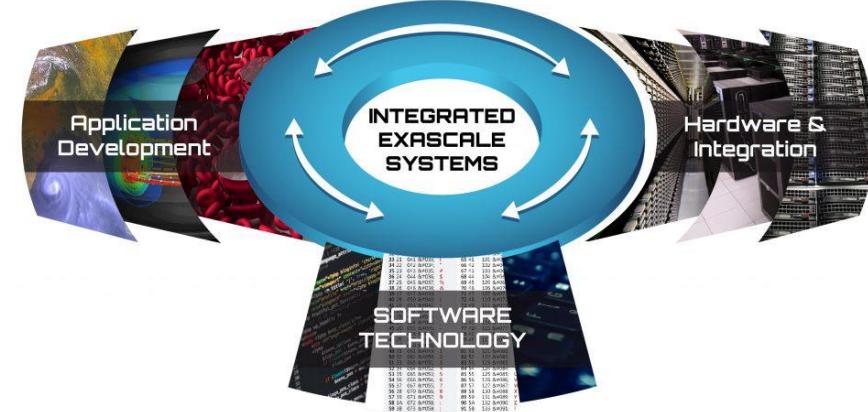
Concurrency is not new

- “Concurrency? That’s not news! People are already writing concurrent applications.” That’s true. Of a small fraction of developers.
- People have been doing object-oriented programming since at least the days of Simula in the late 1960s. But OO didn’t become a revolution, and dominant in the mainstream, until the 1990s. Why then? The reason the revolution happened was primarily that our industry was driven by requirements to write larger and larger systems that solved larger and larger problems and exploited the greater and greater CPU and storage resources that were becoming available. OOP’s strengths in abstraction and dependency management made it a necessity for achieving large-scale software development that is economical, reliable, and repeatable.
- A similar situation occurs today. Applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains that have now started becoming available and will continue to materialize over the next several years.
- Probably the key issue today is that **concurrency really is hard**: The programming model, meaning the model in the programmer’s head that he needs to reason reliably about his program, is much harder than it is for sequential control flow.

Conclusions

- Now is the time to take a hard look at the design of your application, determine what operations are CPU-sensitive now or are likely to become so soon, and identify how those places could benefit from concurrency.
- A few rare classes of applications are naturally parallelizable, but most aren't. Even when you know exactly where you're CPU-bound, you may well find it difficult to figure out how to parallelize those operations; all the more reason to start thinking about it now. Implicitly parallelizing compilers can help a little, but don't expect much; they can't do nearly as good a job of parallelizing your sequential program as you could do by turning it into an explicitly parallel and threaded version.
- Not all applications (or, more precisely, important operations of an application) are amenable to parallelization. The usual example here is that just because it takes one woman nine months to produce a baby doesn't imply that nine women could produce one baby in one month.
- But did you notice the problem with leaving the analogy at that? Here's the trick question to ask the next person who uses it on you: Can you conclude from this that the Human Baby Problem is inherently not amenable to parallelization? Usually people relating this analogy err in quickly concluding that it demonstrates an inherently nonparallel problem, but that's actually not necessarily correct at all.
- **It is indeed an inherently nonparallel problem if the goal is to produce one child. It is actually an ideally parallelizable problem if the goal is to produce many children!**
- Knowing the real goals can make all the difference. This basic goal-oriented principle is something to keep in mind when considering whether and how to parallelize your software.

Moore's law is dead – long live AI

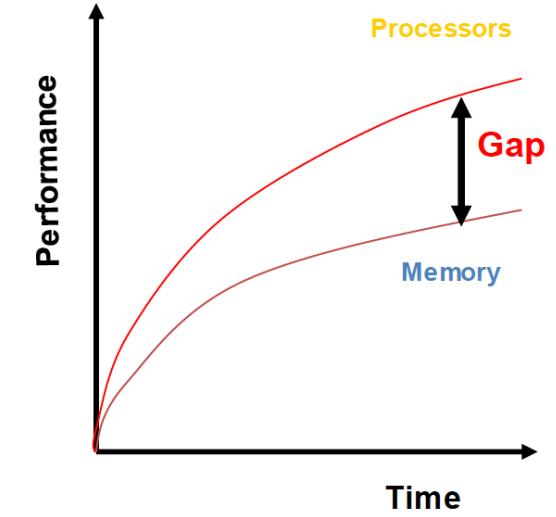


- Nvidia founder and CEO, Jensen Huang
Accelerated computing looks at the entire stack, algorithms, software and processor,” he said. “We can study where bottlenecks are. New software systems make the application go faster, not just the chip”.
- “Let’s say you have an airplane that has to deliver a package. It takes 12 hours to deliver it. Instead of making the plane go faster, concentrate on how to deliver the package faster, look at 3D printing at the destination,” he explained. Applications acceleration is not just to make the chip go faster, it is to deliver the goal faster.”
- <https://www.electronicsweekly.com/news/products/micros/jensen-huang-moores-law-dead-long-live-ai-2018-10/>
- <https://siliconangle.com/2021/04/10/new-era-innovation-moores-law-not-dead-ai-ready-explode/>
- *But in this course we'll focus on traditional HPC techniques.*

von Neumann Architecture

It is a too high level and simple abstraction for present computers.

- Memories
 - The real/main limitation for a CPU is the performance difference between data to/from memory.
 - while memory looks randomly addressable, in practice there is a concept of **locality**: once a data item has been loaded, nearby items are more efficient to load
 - moreover, there are different **memory levels** to deal with the memory wall
- Instruction Level Parallelism (ILP)
 - contemporary CPUs operate on several instructions **simultaneously** and, possibly, they are handled in a **different order** with respect to the sequential code
 - while each instruction can take several clock cycles to complete, a processor can complete **one instruction per cycle** in favourable circumstances (pipelining)
 - their inputs and outputs are also being moved in an **overlapping** manner



Pipelining

- A single instruction takes several clock cycles to complete
- Subdivide an instruction into stages (or segments):
 1. Instruction decode (including finding the location of operands)
 2. Copying the operands into registers ('data fetch').
 3. Aligning the exponents; the addition $.35 \times 10^{-1} + .6 \times 10^{-2}$ becomes $.35 \times 10^{-1} + .06 \times 10^{-1}$
 4. Actual operation
 5. Normalize the result if needed ($.41 \times 10^{-1}$ does not require it)
 6. Storing the result
- Pipeline: separate piece of hardware for each subdivision

Pipelining

- If every stage is designed to finish in 1 clock cycle, the whole instruction takes 5 cycles. However, if each stage has **its own hardware**, we can execute two operations in much less than 5 cycles:
 - Execute the IF for the first operation;
 - Do the ID for the first operation, and at the same time the IF for the second.
 - Execute the third stage for the first operation and the second stage of the second operation simultaneously.
 - ...
- You see that the first operation still takes 5 clock cycles, but the second one is finished a mere 1 cycle later, so both in 6 cycles.
- On a **traditional EU**, producing n results takes $t(n) = nst$ where s is the number of stages, and t the clock cycle time.
- On a **pipelined EU** the time is $t(n) = [s+(n-1)]$ where s becomes a setup cost: the first operation still must go through the same stages as before, but after that one result will be produced each cycle.

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction x	IF	ID	EXE	MEM	WB				
Instruction x+1		IF	ID	EXE	MEM	WB			
Instruction x+2			IF	ID	EXE	MEM	WB		
Instruction x+3				IF	ID	EXE	MEM	WB	
Instruction x+4					IF	ID	EXE	MEM	WB

Figure 7: Simple 5-stage pipeline diagram.

Pipelining

add x1, x2, x3

ID Two registers, x2 and x3, are read from the register file

WB The result is written into x1 in the register file

Id x1, offset(x2)

ID the x2 register value is read from the register file.

EXE The ALU computes the sum of X2 with the offset

MEM The result is used as the address for the data memory.

WB The data from the memory unit is written into the register file

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.24 Total time for each instruction calculated from the time for each component.

Pipelining

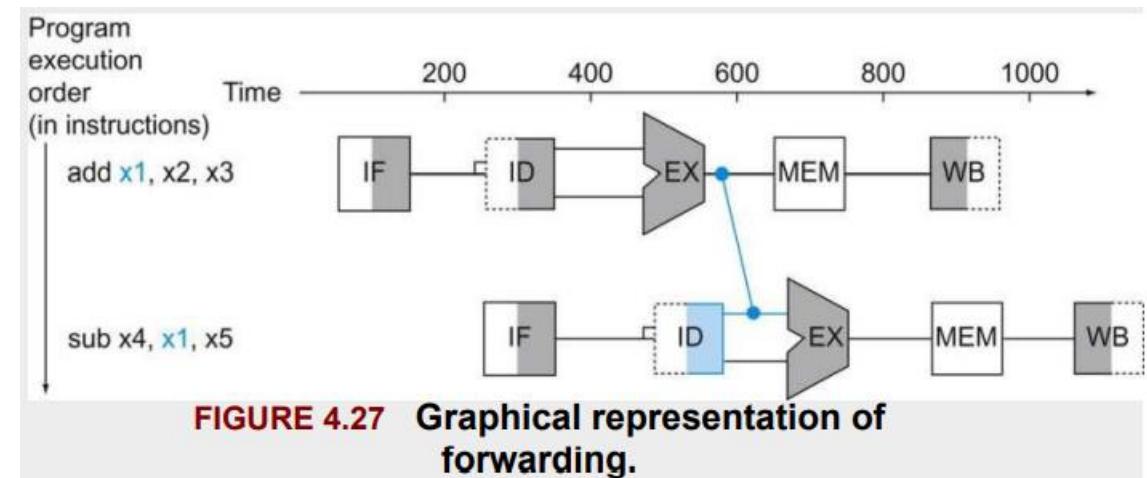
- Most modern CPUs are deeply pipelined, aka super pipelined
- Throughput: the number of instructions that complete and exit the pipeline per unit of time
- *In theory* the time required to move an instruction from one stage to the other can define the basic machine cycle or clock for the CPU.
 - Actually, some stages can require large execution times (e.g. a division can take 10 to 20 clock cycles), so faster cycles to speedup the most common cases (sum vs sqrt, cached data vs non cached ones)
- Hazards limit the performance.

Hazards

- **Structural** hazards are due to resource conflicts.
Hardware resources replication.
- **Control** hazards are caused due to the changes in the program flow.
Branch prediction and speculative execution.
- **Data** hazards are caused by data dependencies.
Bypassing/forwarding or
Out-of-order execution

Data Hazards

- It does not solve all the possible issues, but **mitigates** them.
- E.g. if the first instruction is LD x1 we'll have the value at the fourth stage.



- out-of-order (OOO) execution: sequential instructions can enter the execution pipeline stage in any **arbitrary order** only limited by their dependencies.
- BUT OOO execution CPUs must still give **the same result** as if all instructions were executed in the program order.
- An instruction is called **retired** when its results are correct and visible in the architectural state.

Instruction	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
Instruction x	IF	ID	EXE	MEM	WB					
Instruction x+1		IF	ID			EXE	MEM	WB		
Instruction x+2			IF	ID	EXE	MEM			WB	
Instruction x+3				IF	ID		EXE	MEM		WB

Figure 8: The concept of Out-Of-Order execution.

Structural hazards

- Traditionally: one instruction at a time.
 - Now multiple Eus
- Superscalar CPUs: they can issue 2-6 instructions at a time
 - Scheduling via hardware or compiler with the Very Long Instruction Word

Instruction	Clock cycle					
	1	2	3	4	5	6
Instruction x	IF	ID	EXE	MEM	WB	
Instruction x+1	IF	ID	EXE	MEM	WB	
Instruction x+2		IF	ID	EXE	MEM	WB
Instruction x+3		IF	ID	EXE	MEM	WB

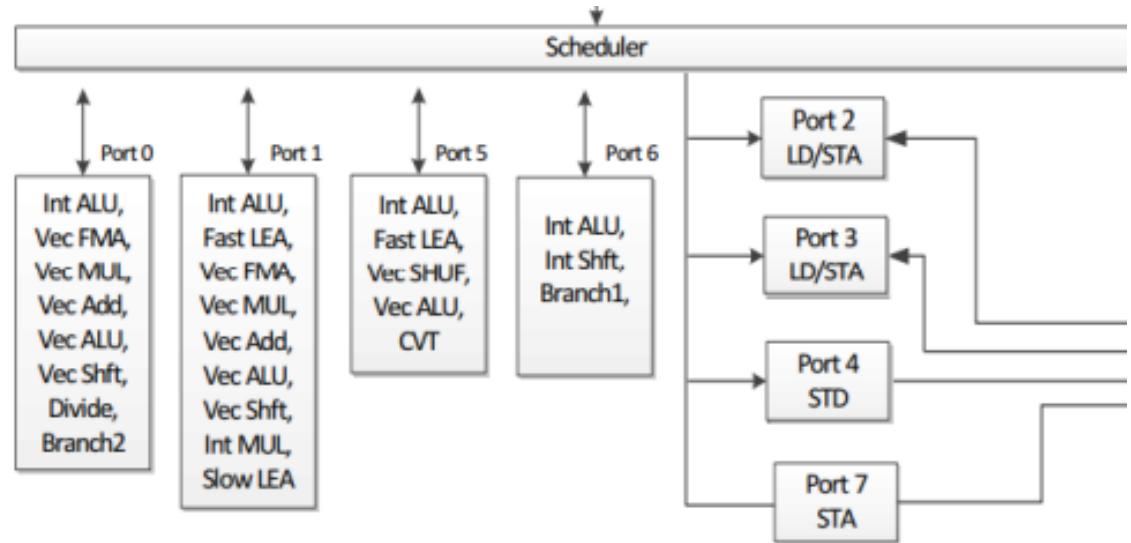


Figure 9: The pipeline diagram for a simple 2-way superscalar CPU.

Intel skylane

Branch Prediction

- Branches are points in the instruction stream where the execution may jump to another location, instead of executing the next instruction.
- Processors, instead of waiting for the loop to branch route outcome, try to **predict** the action that will be taken, in order to keep the pipeline full
- First possibility: at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for branches that jump to an earlier address.
- Otherwise, dynamic hardware predictors make their guesses depending on the behaviour of each branch and may change predictions for a branch over the life of a program
- One popular approach to dynamic prediction of branches is *keeping a history* for each branch as taken or untaken, and then using the recent past behaviour to predict the future. The amount and type of history kept have become extensive, with the result being that dynamic branch predictors can correctly predict branches with more than 90% accuracy.
- In every case when the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and **must restart the pipeline from the proper branch address**. Longer pipelines exacerbate the problem, in this case by raising the cost of misprediction

Speculative execution

- The CPU takes a guess on an outcome of the branch and initiates processing instructions from the chosen path
- State changes to the machine cannot be committed until the condition is resolved to ensure that the architecture state of the machine is never impacted by speculatively executing instructions.
 - A branch instruction can be dependent on a value loaded from memory, which can take hundreds of cycles

Listing 3 Speculative execution

```
if (a < b)
    foo();
else
    bar();
```

Instruction	Clock cycle							
	1	2	3	4	5	6	7	8
BRANCH (a < b)	IF	ID	EXE	MEM	WB			
CALL foo				IF	ID	EXE	MEM	WB
// INSTR from foo					IF	ID	EXE	MEM

(a) No speculation

Instruction	Clock cycle						
	1	2	3	4	5	6	7
BRANCH (a < b)	IF	ID	EXE	MEM	WB		
CALL foo			IF*	ID*	EXE	MEM	WB
// INSTR from foo				IF*	ID	EXE	MEM

(b) Speculative execution

To summarize

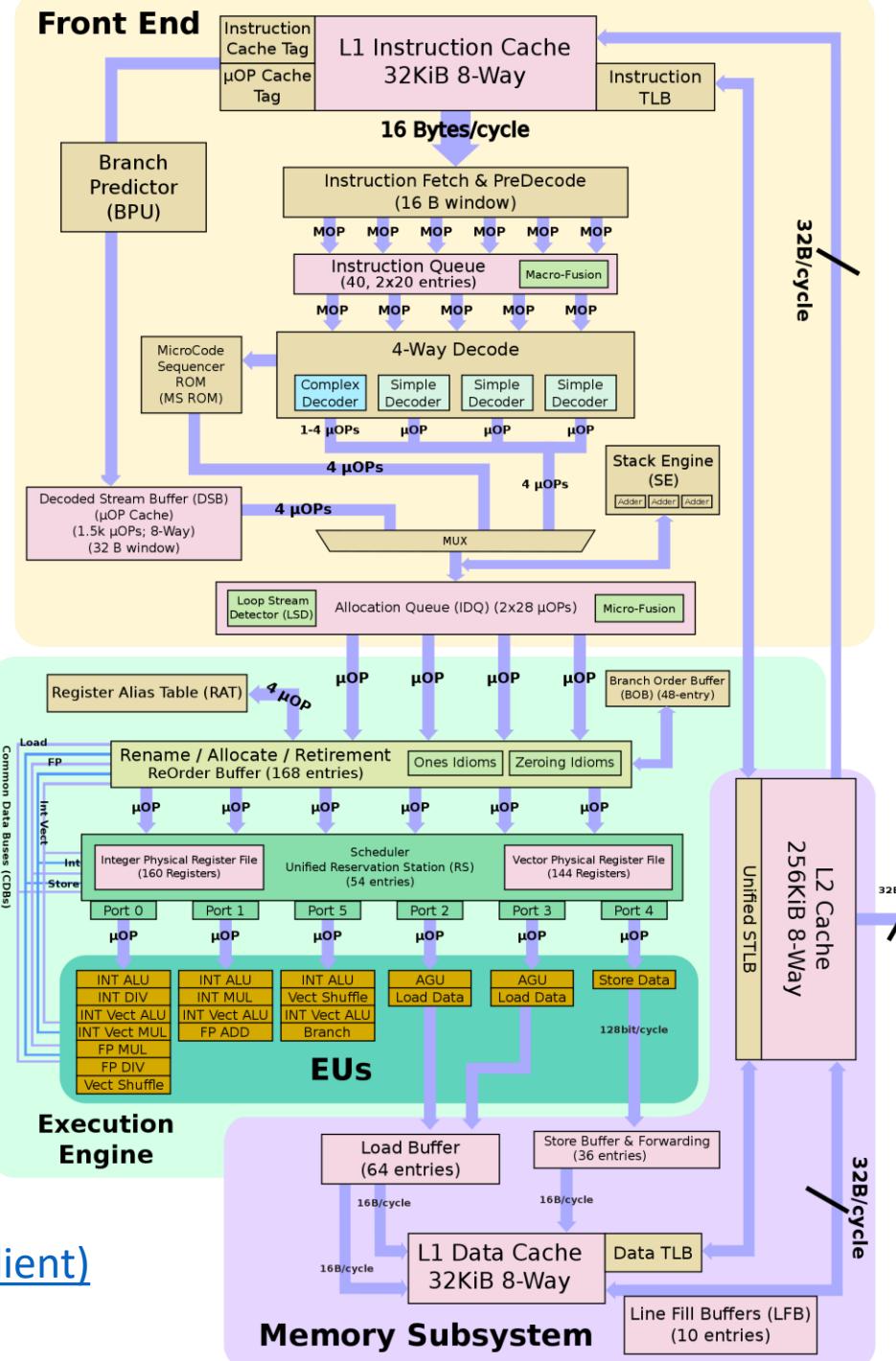
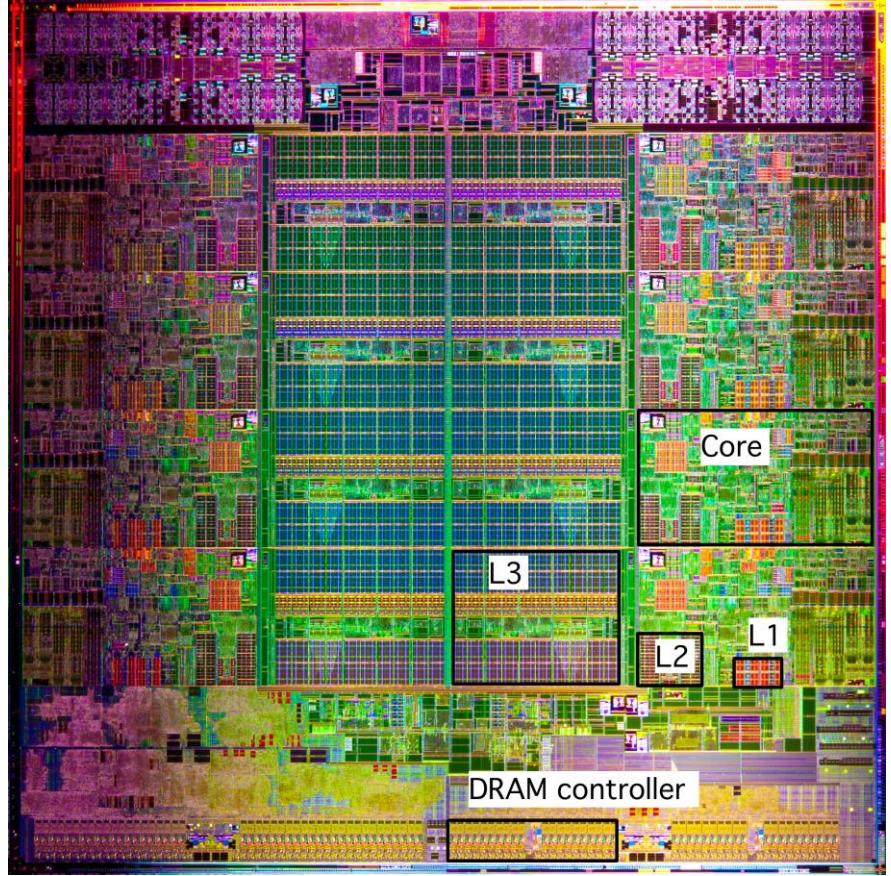
- A pipelined processor can speed up operations by a factor of 4 (or more) with respect to earlier CPUs.
- But dependencies and branches have to be taken into account
- These days, CPUs can have 20-stage pipelines or more (up to 31).
Are they incredibly fast?

<https://www.hardwaresecrets.com/inside-pentium-4-architecture/2/>

<https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/a-journey-through-the-cpu-pipeline-r3115/>

- Chip designers continue to increase the clock rate, and the pipeline segments can no longer finish their work in one cycle, so they are further split up. Sometimes there are even segments in which nothing happens: that time is needed to make sure data can travel to a different part of the chip in time.
- Thanks to pipelining, for modern CPUs there was a simple (maybe too simple) relation between the clock speed and the **peak** performance: #FPU x clock speed
- BUT now vector operations and multicore...

What does a CPU look like? The Intel Sandy Bridge processor

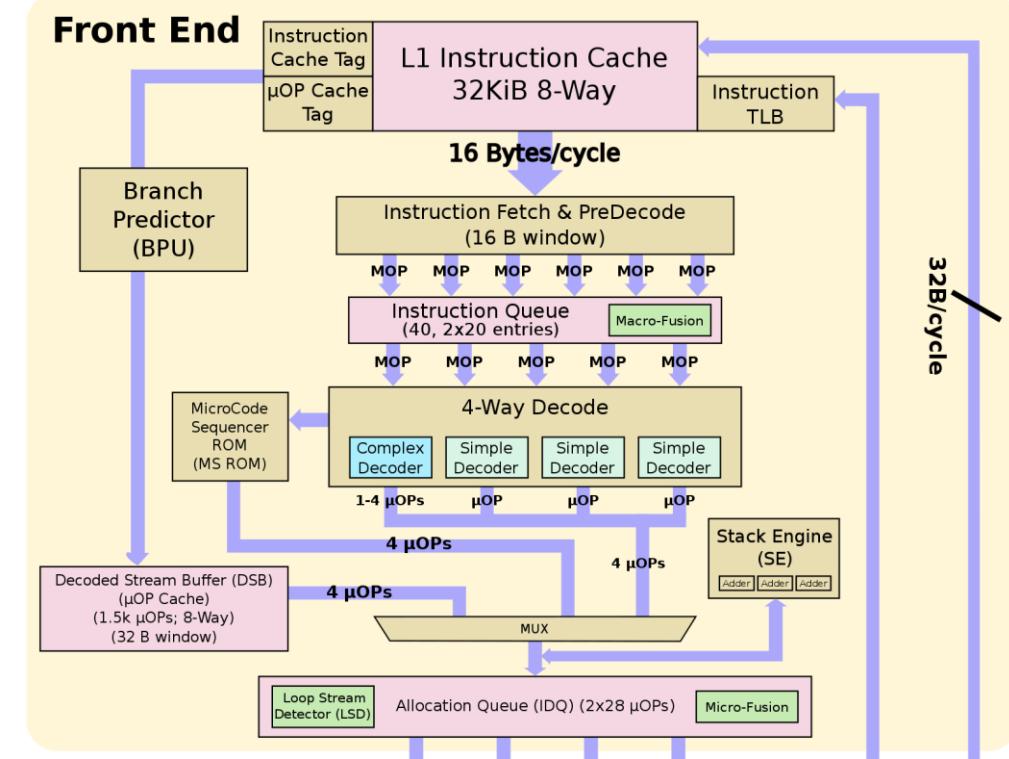


[https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client))

<https://www.realworldtech.com/sandy-bridge/3/>

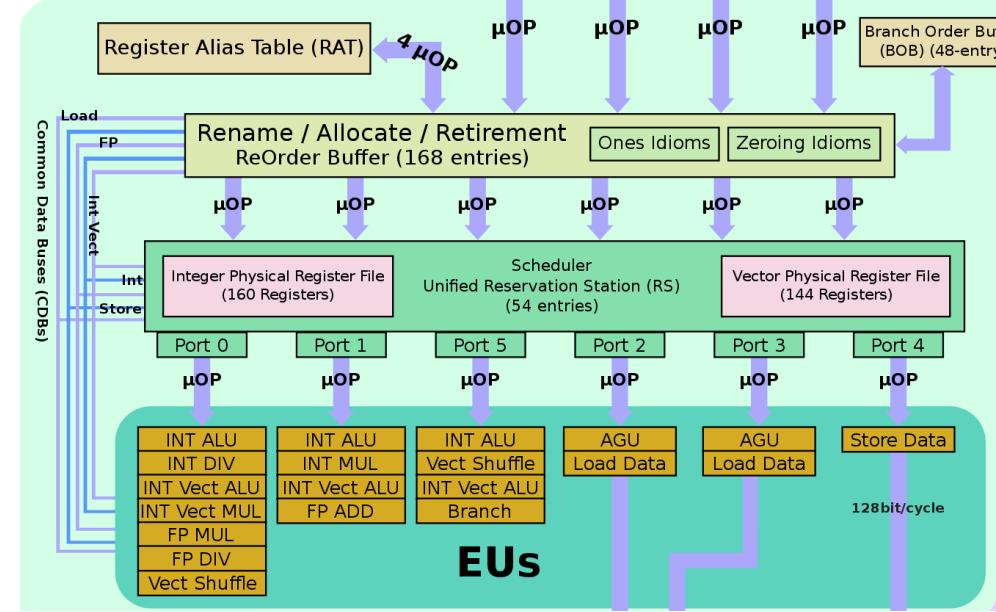
Sandy Bridge – Front End

- The **front-end** is tasked with the challenge of fetching the complex x86 instructions from memory, decoding them, and delivering them to the execution units.
- x86 instructions are complex and variable length.
At the **pre-decode** buffer the instruction boundaries get detected and marked. The result is a set of Macro-Operation (MOP), a more complex version of a micro-operation as handled by the microprocessor.
- The **branch prediction unit** (BPU) attempts to guess the flow of instructions. Due to the deep pipeline, a flush is a rather expensive event which ends up discarding over 150 instructions that are in-flight.
- The pre-decoded instructions are delivered to the **Instruction Queue**, that does macro-op fusion if possible, e.g. single compare-and-branch instructions.



Sandy Bridge – Execution Engine

- It deals with the execution of out-of-order operations and treats the three classes of μOPs (Floating Point, INTeger and Vector) separately.
- On each cycle, up to 4 μOPs can be delivered here from the front-end. At this stage architectural registers are mapped onto the underlying physical registers. Other additional bookkeeping tasks are also done at this point such as allocating resources for stores, loads, and determining all possible scheduler ports.
- The scheduler holds the μOPs while they wait to be executed. A μOP could be waiting on an operand that has not arrived (e.g., fetched from memory or currently being calculated from another μOPs) or because the execution unit it needs is busy. Once the μOP is ready, they are dispatched through their designated port.
- The scheduler will send the oldest ready μOP to be executed on each of the six ports each cycle. Port 0, 1, and 5 are used for executing computational operations. Ports 2, 3, and 4, are used for executing memory related operations such as loads and stores. Sandy Bridge introduced the AVX extension, a new 256-bit x86 floating point SIMD extension.
- Once a μOP executes it can be retired.



ILP

- In the von Neumann model processors operate through **control flow**: instructions follow each other linearly or with branches without regard for what data they involve.
- Today CPUs switched to the **data flow** model: they analyze several instructions to find data dependencies, and execute instructions in parallel that do not depend on each other
 - instructions that are independent can be started at the same time
 - Independent instructions can be rearranged if the resulting execution will be more efficient (out-of-order execution)
 - data can be speculatively requested before any instruction needing it is actually encountered (prefetching)
 - Pipelining and branch prediction

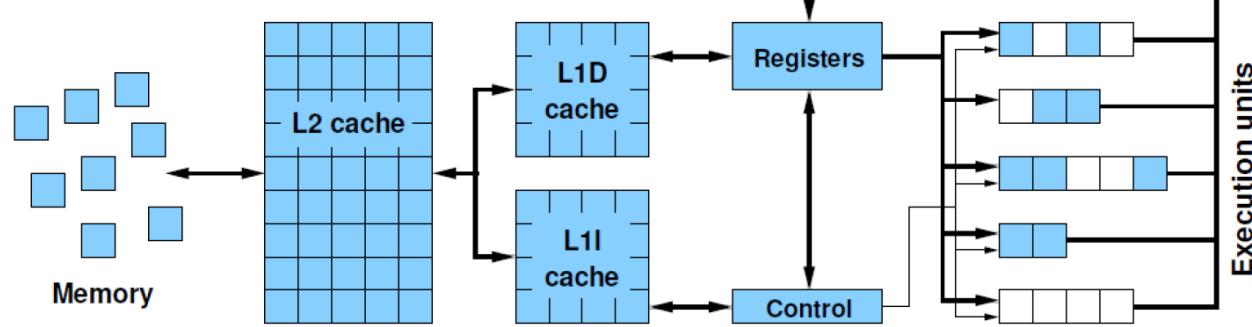
From ILP to multicore

- As clock frequency has gone up, the processor pipeline has grown in length to make the segments executable in less time.
- However ILP has its limits, therefore making pipelines longer (sometimes called ‘deeper’) no longer pays off
 - E.g. branch misprediction results in branch penalty
- Therefore chip designers have moved to multicore architectures as a way of more efficiently using the transistors on a chip

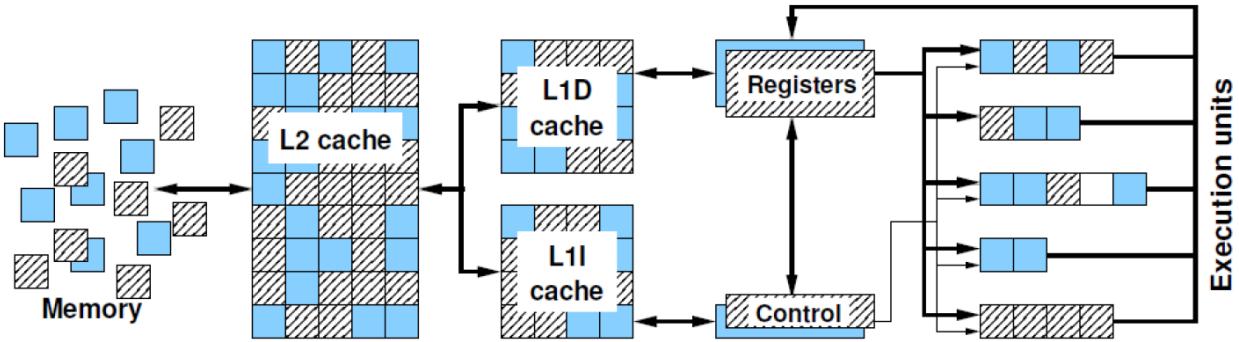
Simultaneous MultiThreading

SMT makes a **single physical core** appear as two or more “logical” cores, i.e. multiple threads/processes run concurrently. It can increase the throughput.

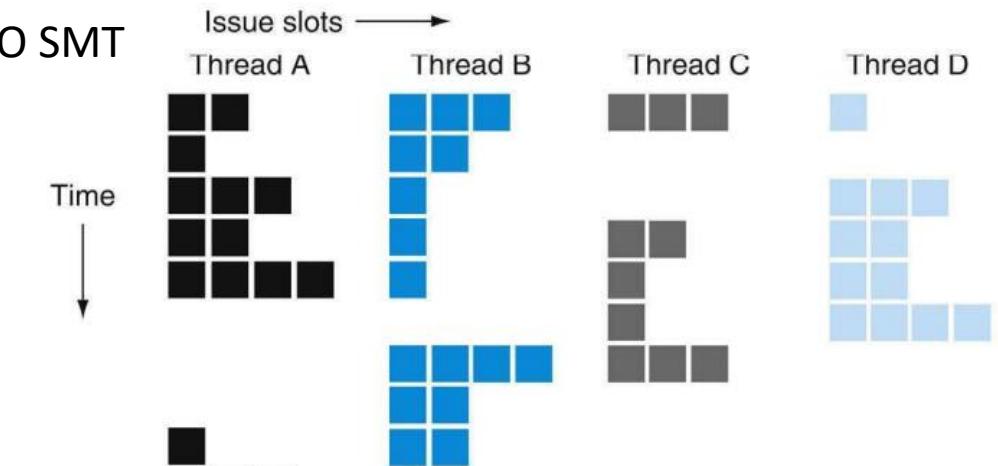
Standard core



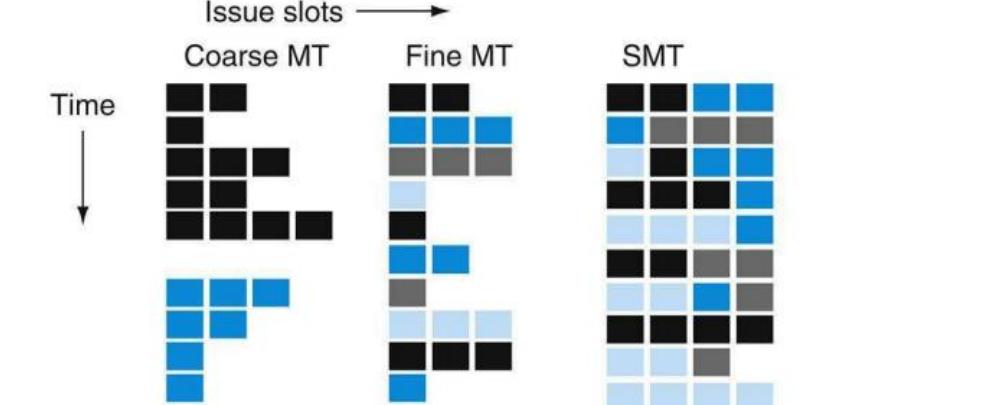
2-way SMT



NO SMT



Issue slots
Time



SMT

- Possible benefit: Better pipeline throughput
 - Filling otherwise unused pipelines
 - Filling pipeline bubbles with other thread's executing instructions:

Thread 0:

```
do i=1,N  
    a(i) = a(i-1)*c  
enddo
```

Dependency → pipeline
stalls until previous MULT
is over

Thread 1:

```
do i=1,N  
    b(i) = s*b(i-2)+d  
enddo
```

Unrelated work in other
thread can fill the pipeline
bubbles

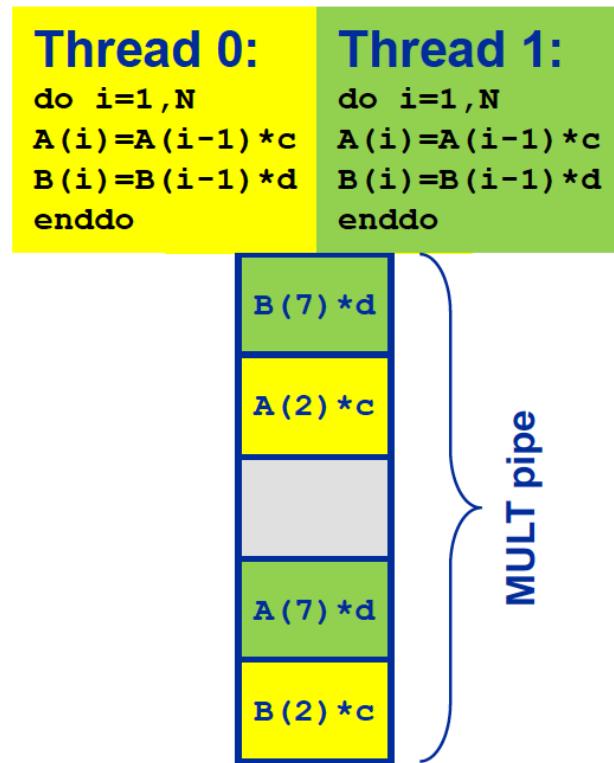
- Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N  
    a(i) = a(i-1)*c  
    b(i) = s*b(i-2)+d  
enddo
```

The ideal workload for SMT

Simple loop-carried dependency benchmark $A(i) = s * A(i-1)$

- Bottleneck: MULT pipeline latency
 - Haswell CPU: 5 cy/it best case
- Running 2 threads via SMT: expect 2.5 cy/it if no other bottlenecks turn up
- Further improvement?
 - Multiple independent streams of instructions per thread
 - What about the data transfer?



Many more details:

<http://joomla.di.unipi.it/~vannesch//SPA/p29-ungerer.pdf>

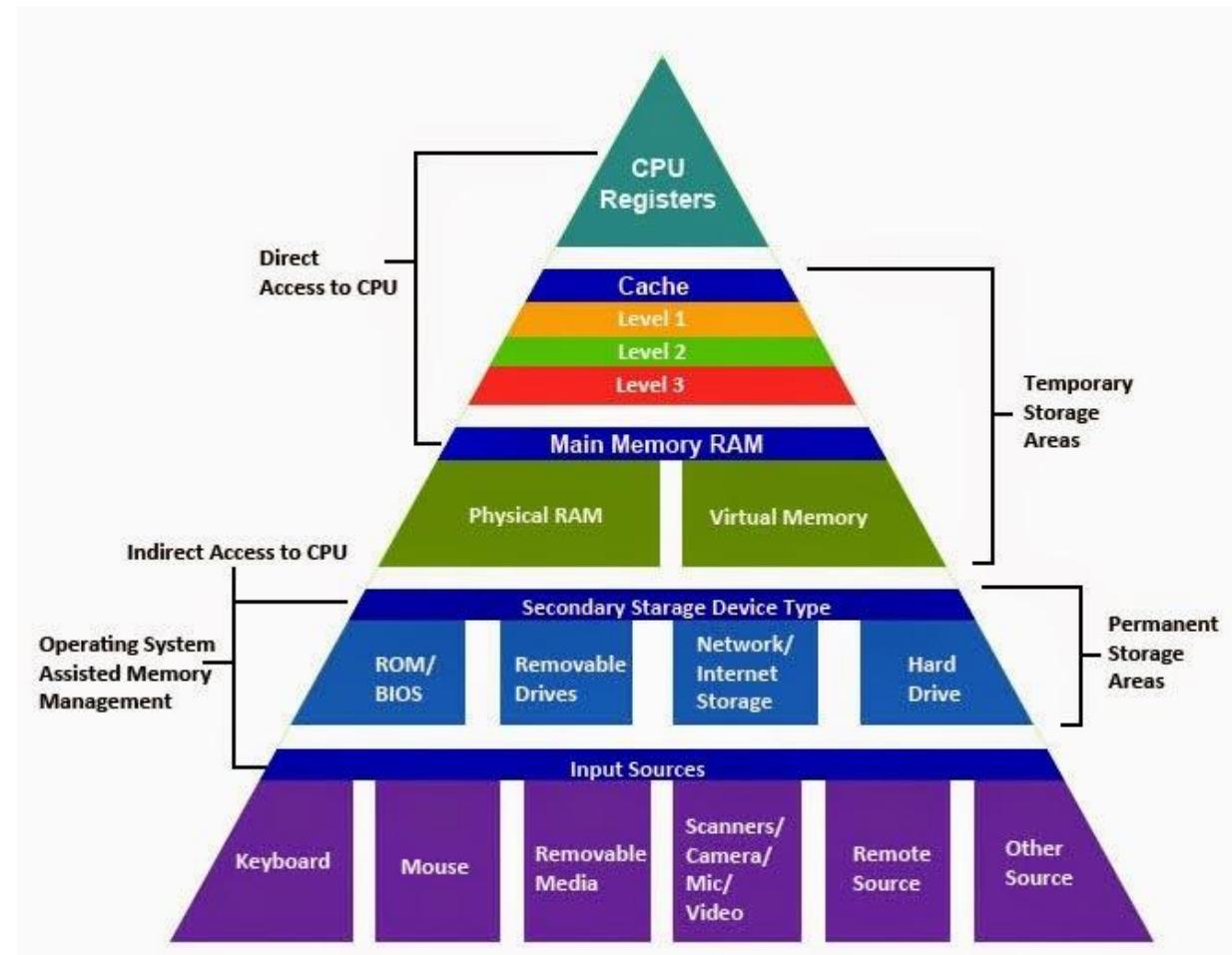
To summarize

- Allows the CPU to switch to another task when the current task is stalled
 - A *context switch* has essentially zero cost
 - The CPU can switch to another task even on stalls of short durations (e.g., waiting for memory operations), so it runs >1 instruction stream (thread/process) at a time
 - Requires CPU with specific support
 - Full register set available per SMT thread
- If the bottleneck is not core execution, SMT will be of limited use
- HyperThreading is Intel's implementation of SMT
- Use `lscpu` / `lstopo` in Linux to get details

```
Architecture:          x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):               256
On-line CPU(s) list: 0-255
Thread(s) per core:  4
Core(s) per socket:  64
Socket(s):            1
NUMA node(s):         1
Vendor ID:            GenuineIntel
CPU family:           6
Model:                87
Model name:           Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz
Stepping:              1
CPU MHz:              1501.601
BogoMIPS:             2600.12
L1d cache:            32K
L1i cache:            32K
L2 cache:              1024K
NUMA node0 CPU(s):    0-255
```

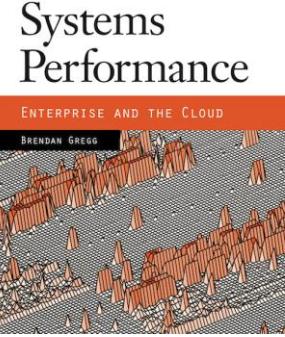
Memory hierarchy

- Von Neuman architecture: data is loaded immediately from memory to the processors, where it is operated on
- There are several memory levels, the memory hierarchy.
These try to alleviate **the memory wall problem** by making **recently used data available quicker** than it would be from main memory.
- Of course, this presupposes that the algorithm and its implementation allow for data to be used multiple times.



Memory Hierarchy

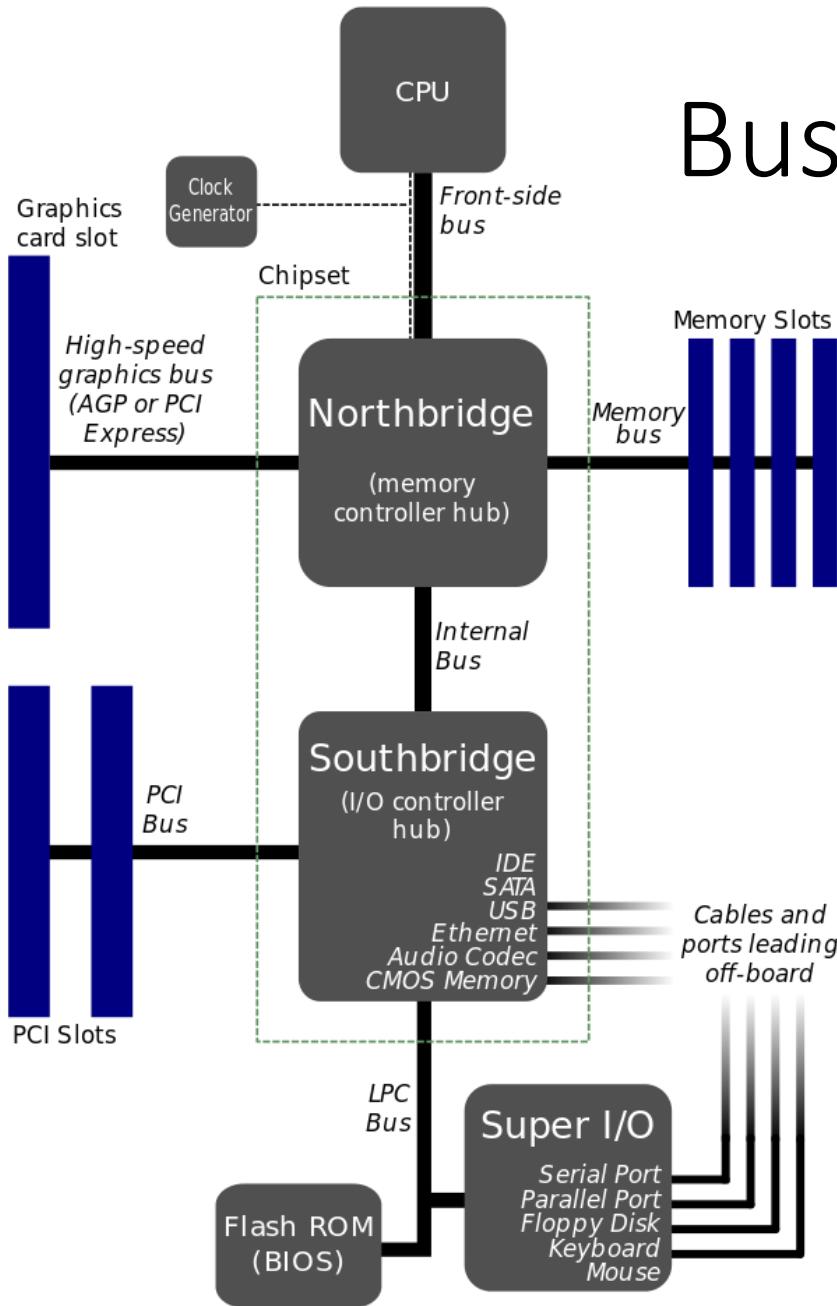
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μ s	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia



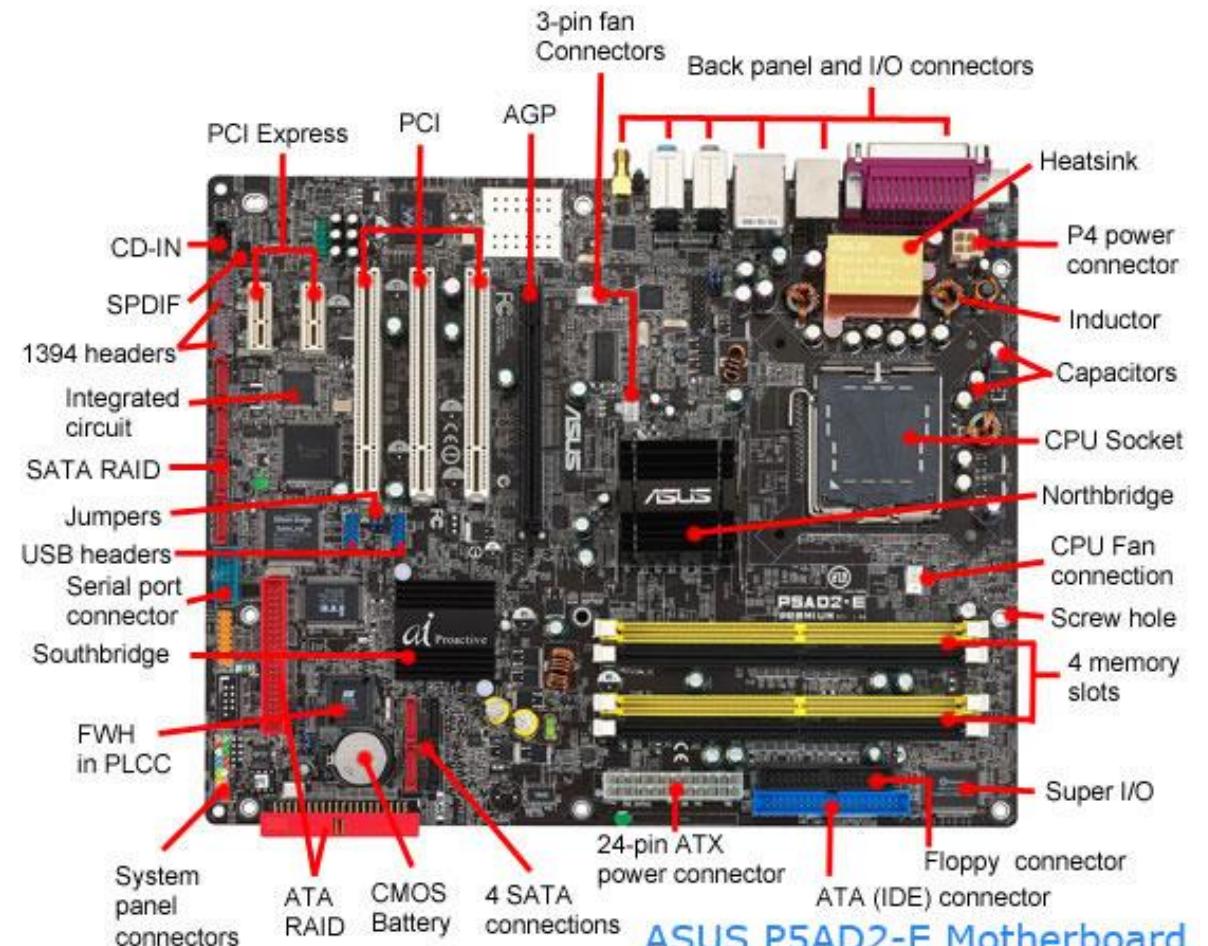
- Clock Speed
- Execution Optimization
- Cache

The **computer performance shell game**, also known as "find the bottleneck", is played between four resources: CPU, Disk, Network, Memory. But which one? How long will you wait?

Buses



- The wires that move data around in a computer are called buses.
- The bus is typically slower than the processor.
- The bandwidth of a bus is also determined by the number of bits that can be moved per clock cycle.



ASUS P5AD2-E Motherboard

Latency and Bandwidth

The two most important terms related to performance for memory subsystems and networks:

- **Latency:** is the delay between the processor issuing a request for a memory item, and the item actually arriving
- It is generally measured in nanoseconds (milliseconds for network latency) or clock periods.
- Sometimes addresses are predictable and compiler will schedule the fetch. Otherwise the out-of-order execution can hide this time (latency hiding)
- **Bandwidth:** the rate at which data arrives at its destination, after the initial latency is overcome
- Units are B/sec (MB/sec, GB/sec, etc.)
- The time needed to transfer n byte is

$$T(n) = \alpha + \beta n$$

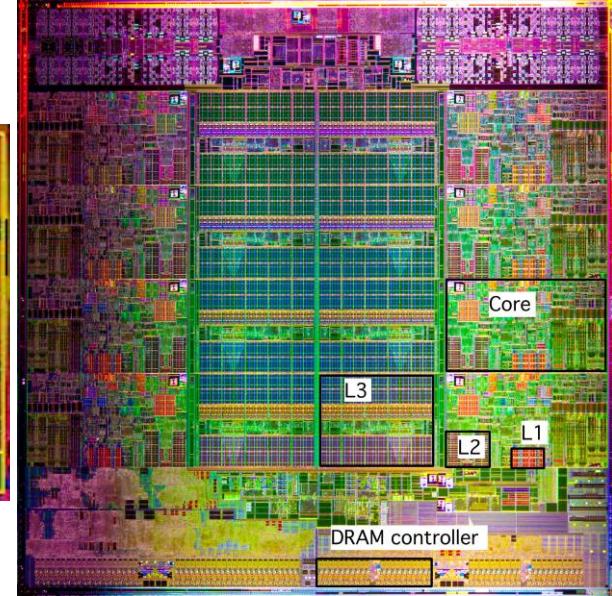
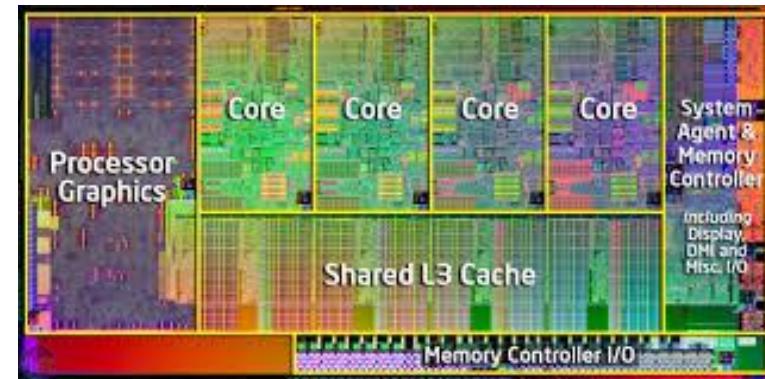
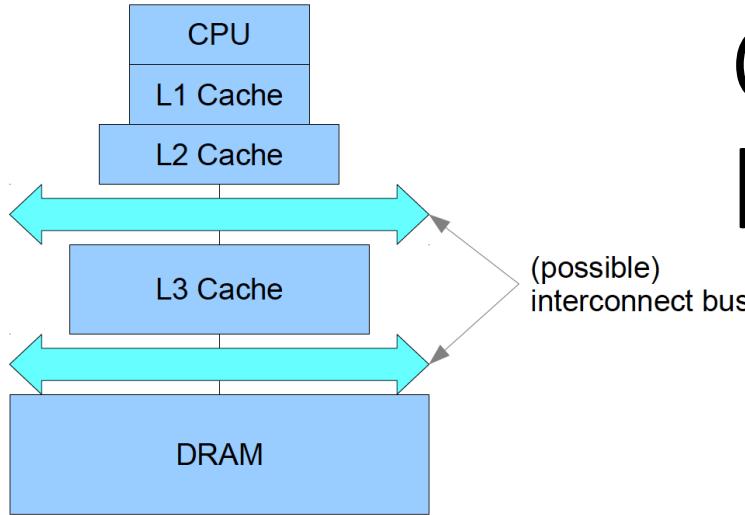
α = latency, a constant value

β = 1/bandwidth, the time per byte

Caches

- In between the registers, which contain the immediate input and output data for instructions, and the main memory where lots of data can reside for a long time, are various levels of cache memory, that have lower latency and higher bandwidth than main memory and where data are kept for an intermediate amount of time.
- Data from memory travels through the caches to wind up in registers. The advantage to having cache memory is that if a data item is reused shortly after it was first needed, it will still be in cache, and therefore it can be accessed much faster than if it would have to be brought in from memory.
- There is an important difference between cache memory and registers: while data is moved into register by explicit assembly instructions, the move from main memory to cache is entirely done by hardware.
- Thus cache use and reuse **is outside of direct programmer control**. You can only **influence it** by exploiting data locality.

Cache levels



- L1 Cache: Data cache closest to registers
 - It is usually split into instruction cache and the data cache
- L2 Cache: Secondary data cache, stores both data and instructions
 - Data from L2 has to go through L1 to registers
 - L2 is 10 to 100 times larger than L1
- Most systems have an L3 cache, ~10x larger than L2
- Cache line
 - The smallest unit of data transferred between main memory and the caches (or between levels of cache)
 - N sequentially-stored, multi-byte words. Typically 64 **bytes** (i.e. 8 double values).

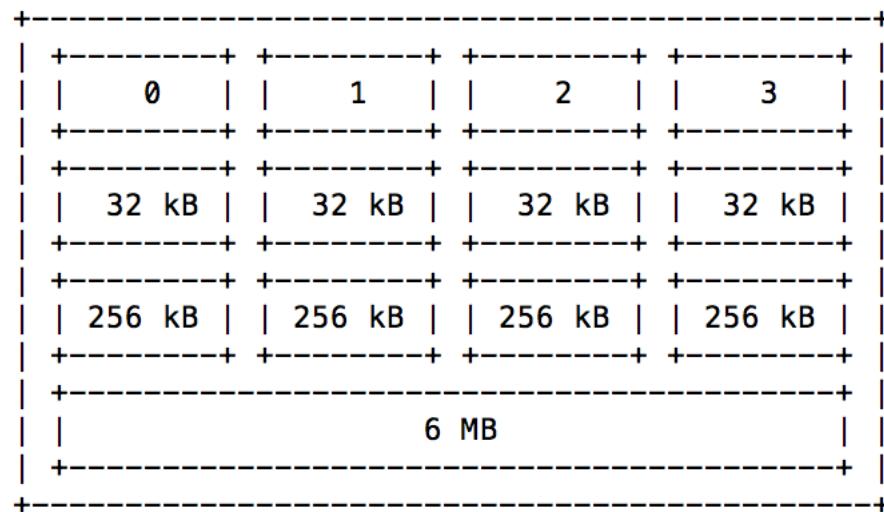
Details

On my 2 machines

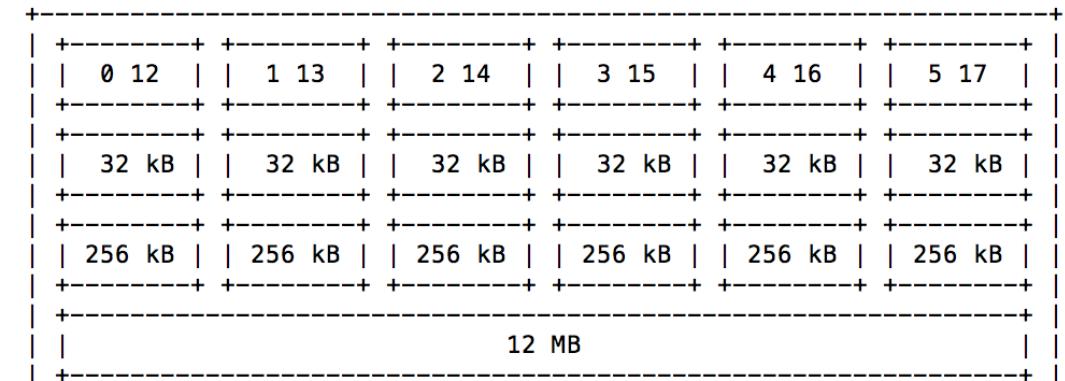
Install perf and

<https://github.com/RRZE-HPC/likwid/wiki>
/usr/local/bin/likwid-topology -g

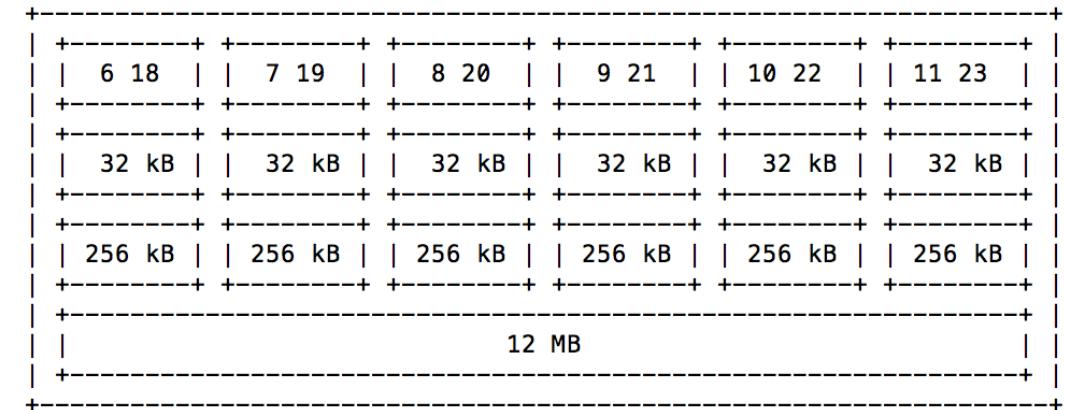
Socket 0:



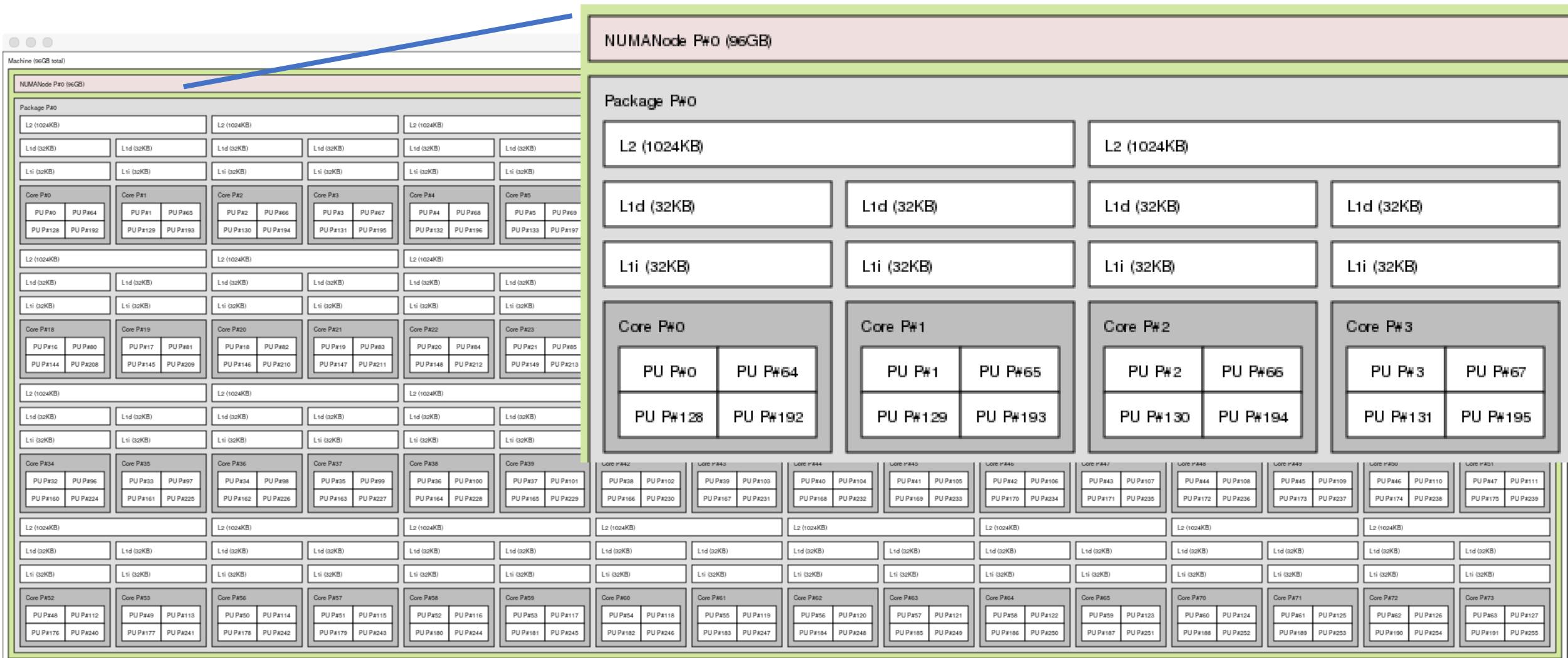
Socket 0:



Socket 1:



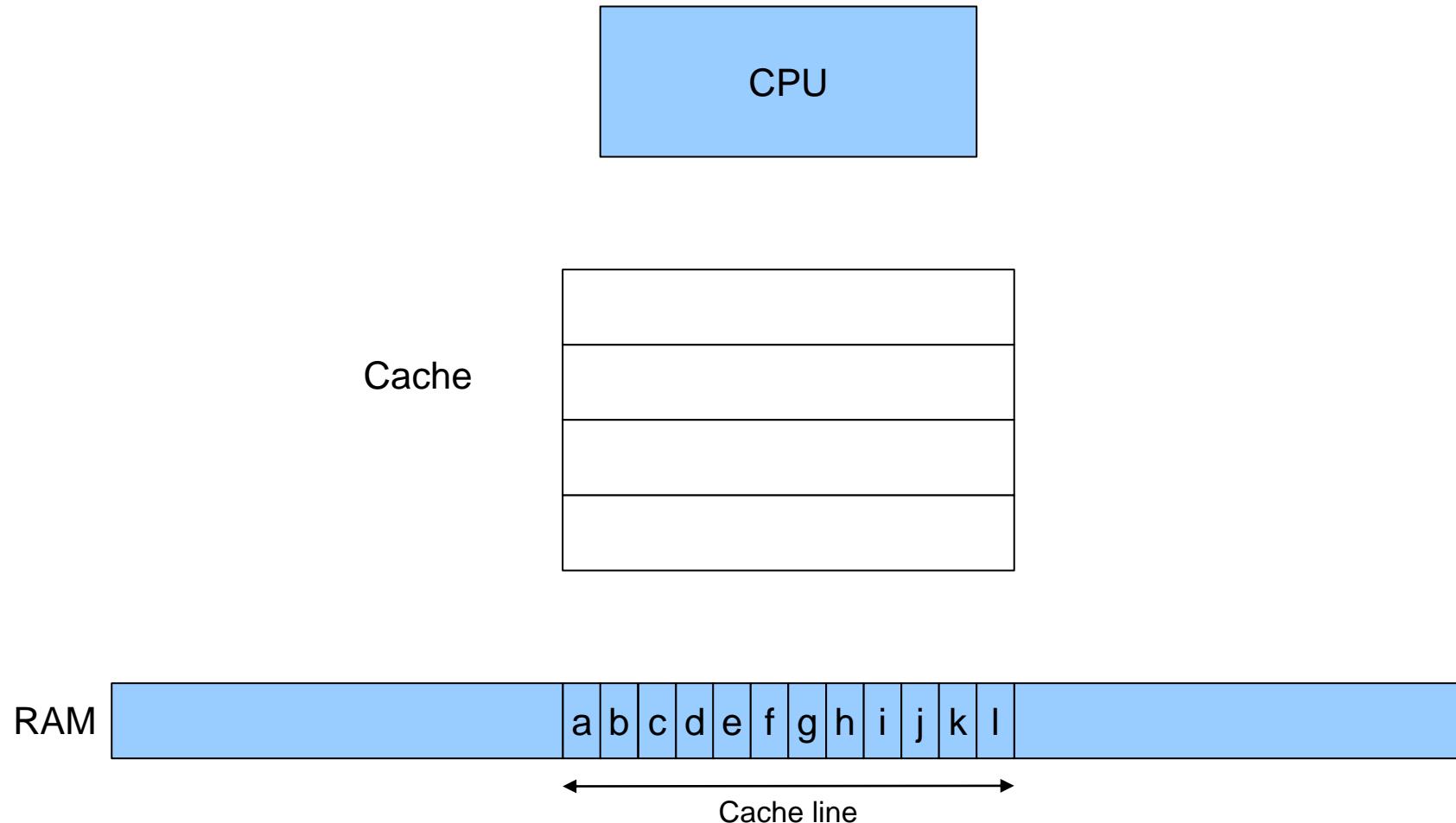
Details – KNL Xeon Phi 7210 (64 cores * 4 thread)



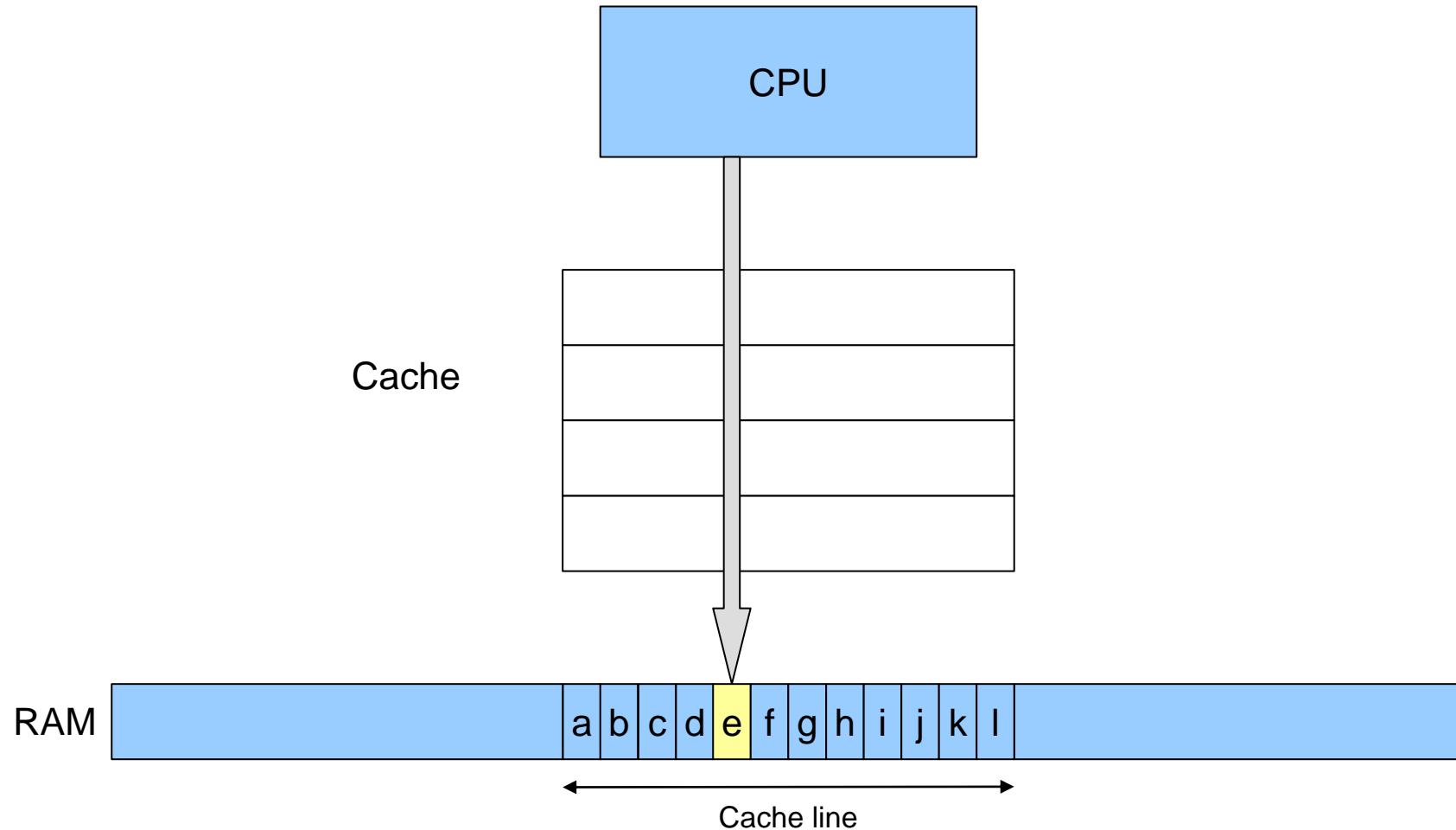
How the cache works

- Data needed in some operation gets copied into the various caches on its way to the processor.
- If, some instructions later, a data item is needed again, it is first searched for in the L1 cache; if it is not found there, it is searched for in the L2 cache; if it is not found there, it is loaded from main memory.
- Finding data in cache is called a cache hit, and not finding it a cache miss.
- Unlike main memory, which is expandable, caches are fixed in size. If a version of a processor chip exists with a larger cache, it is usually considerably more expensive.

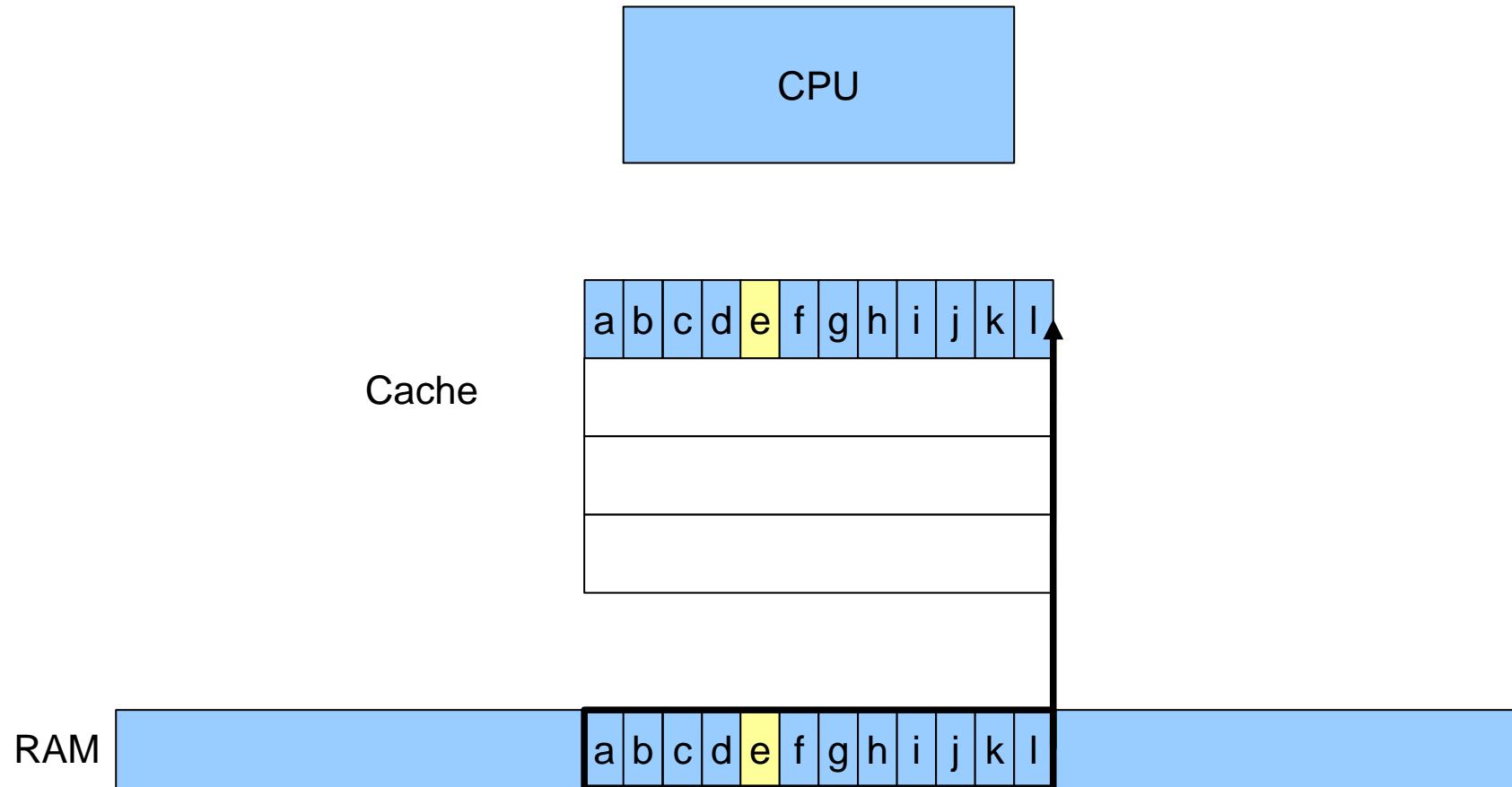
Example



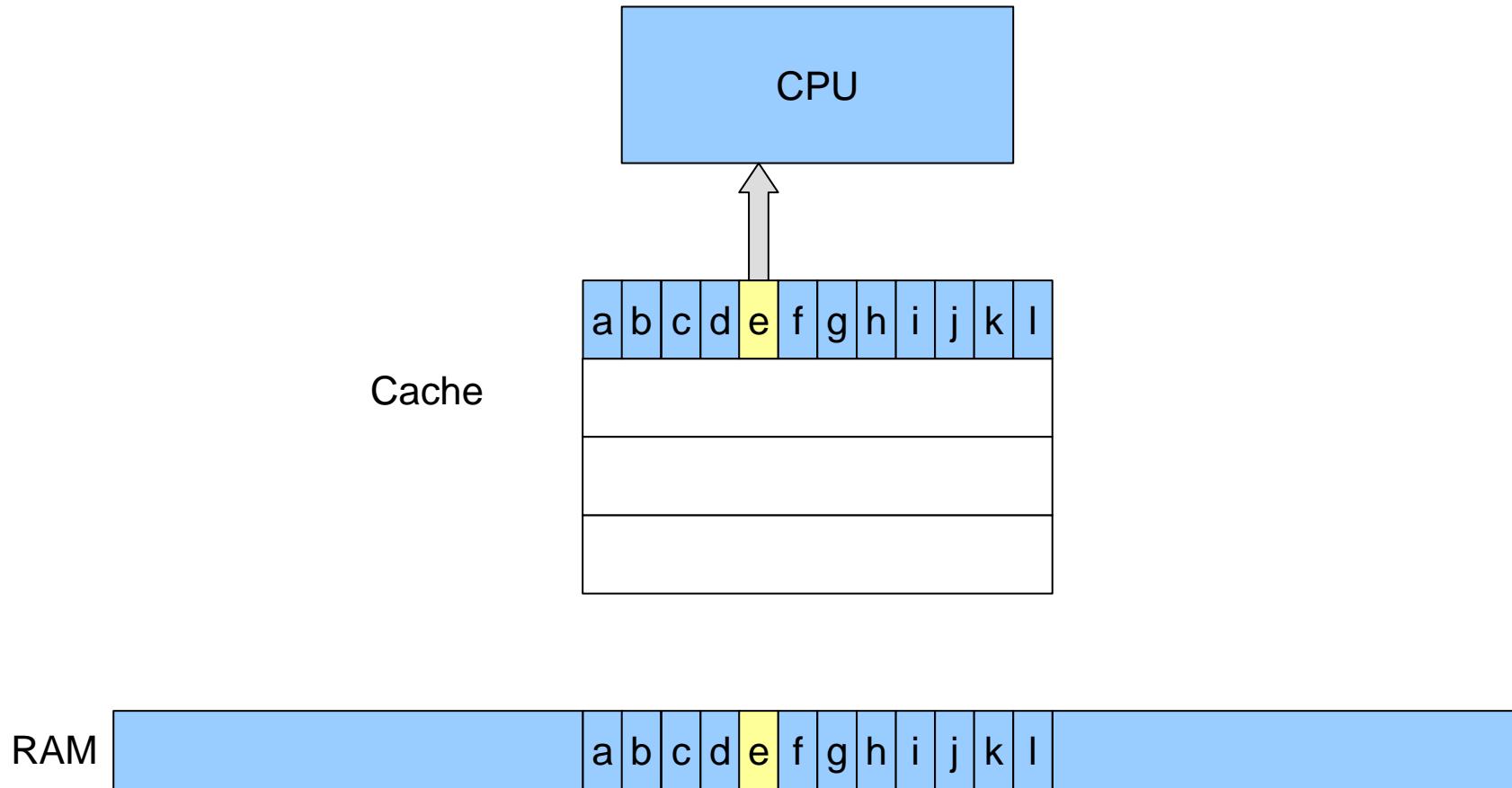
Example



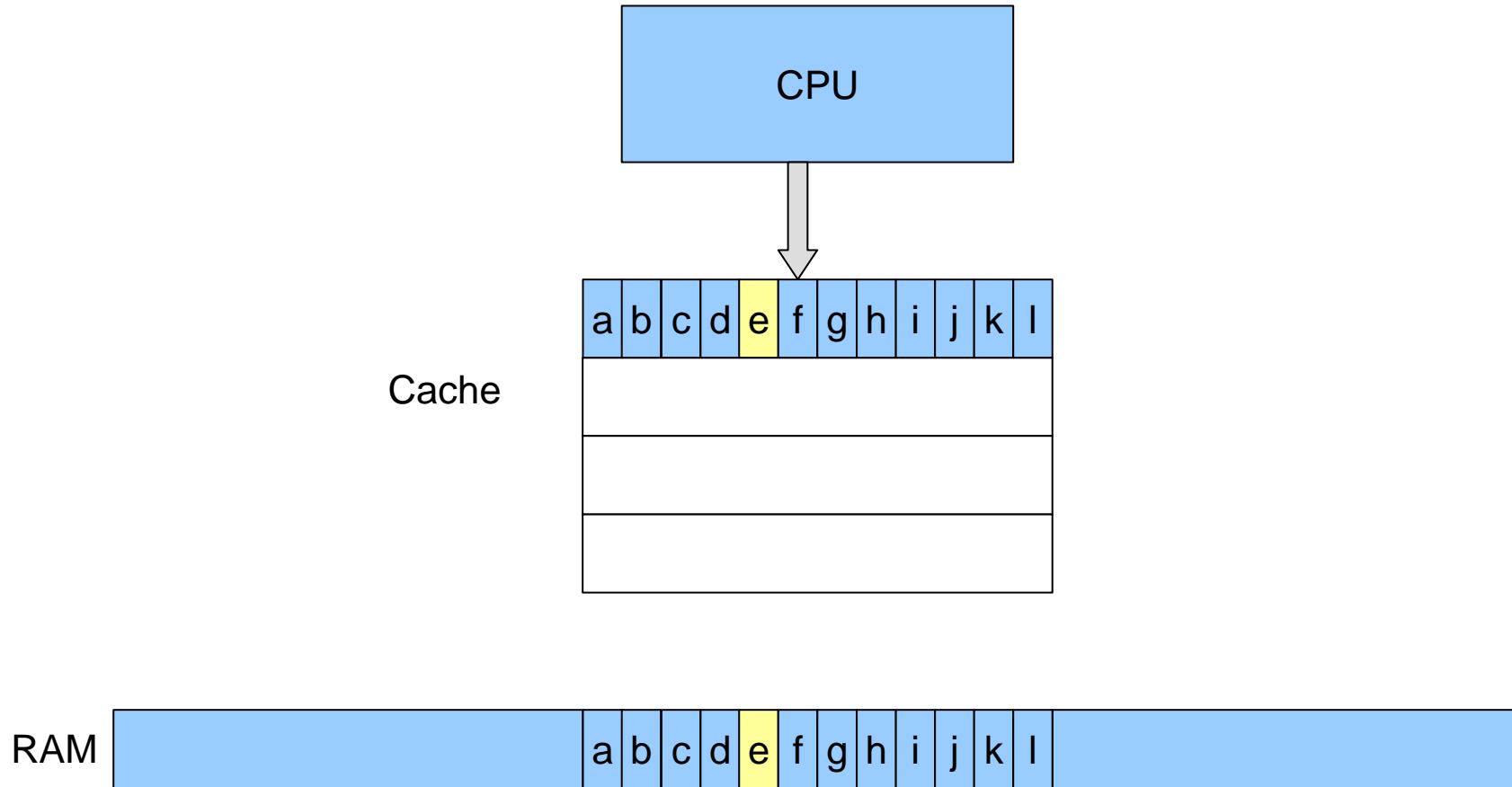
Example



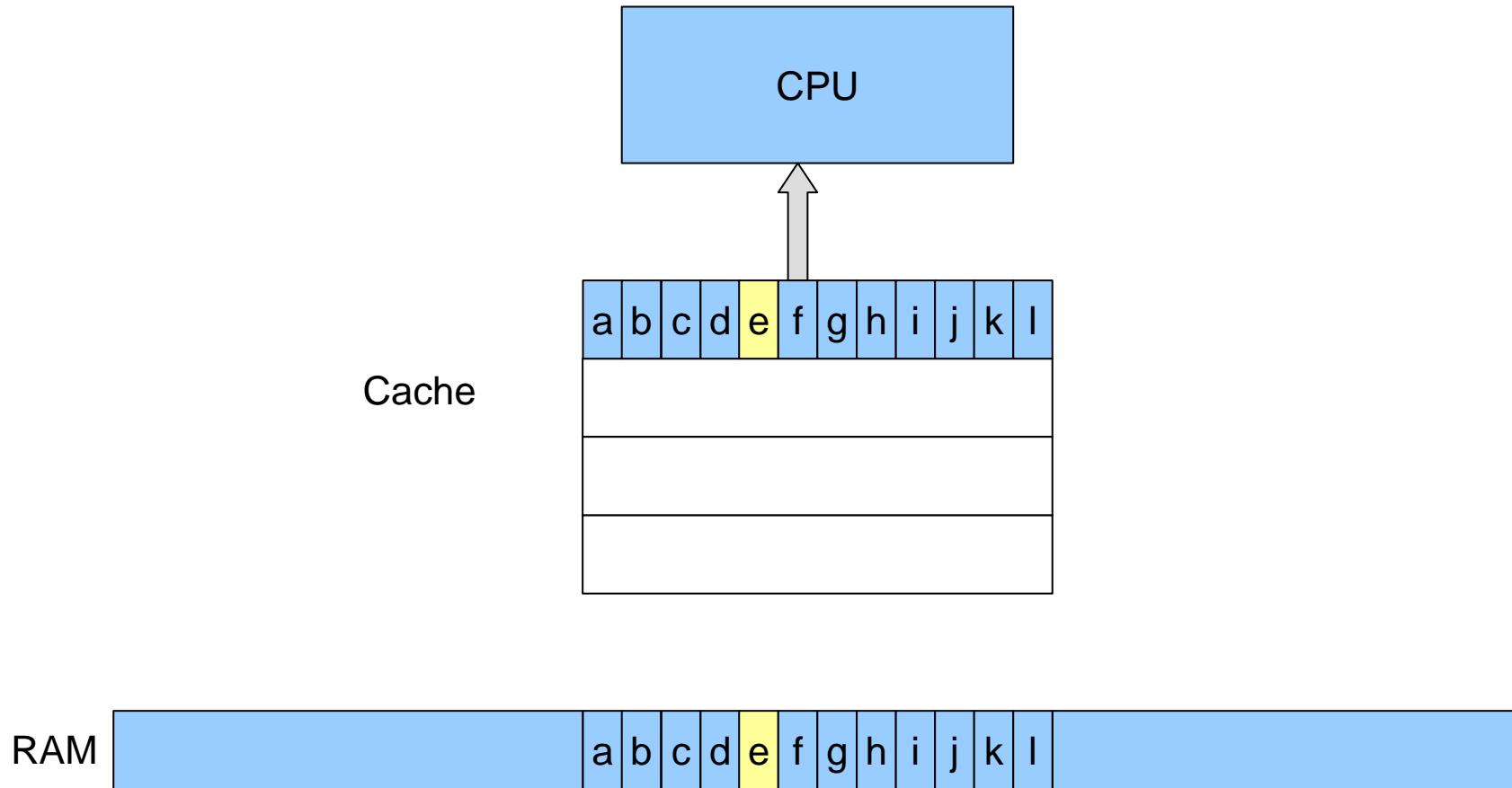
Example



Example



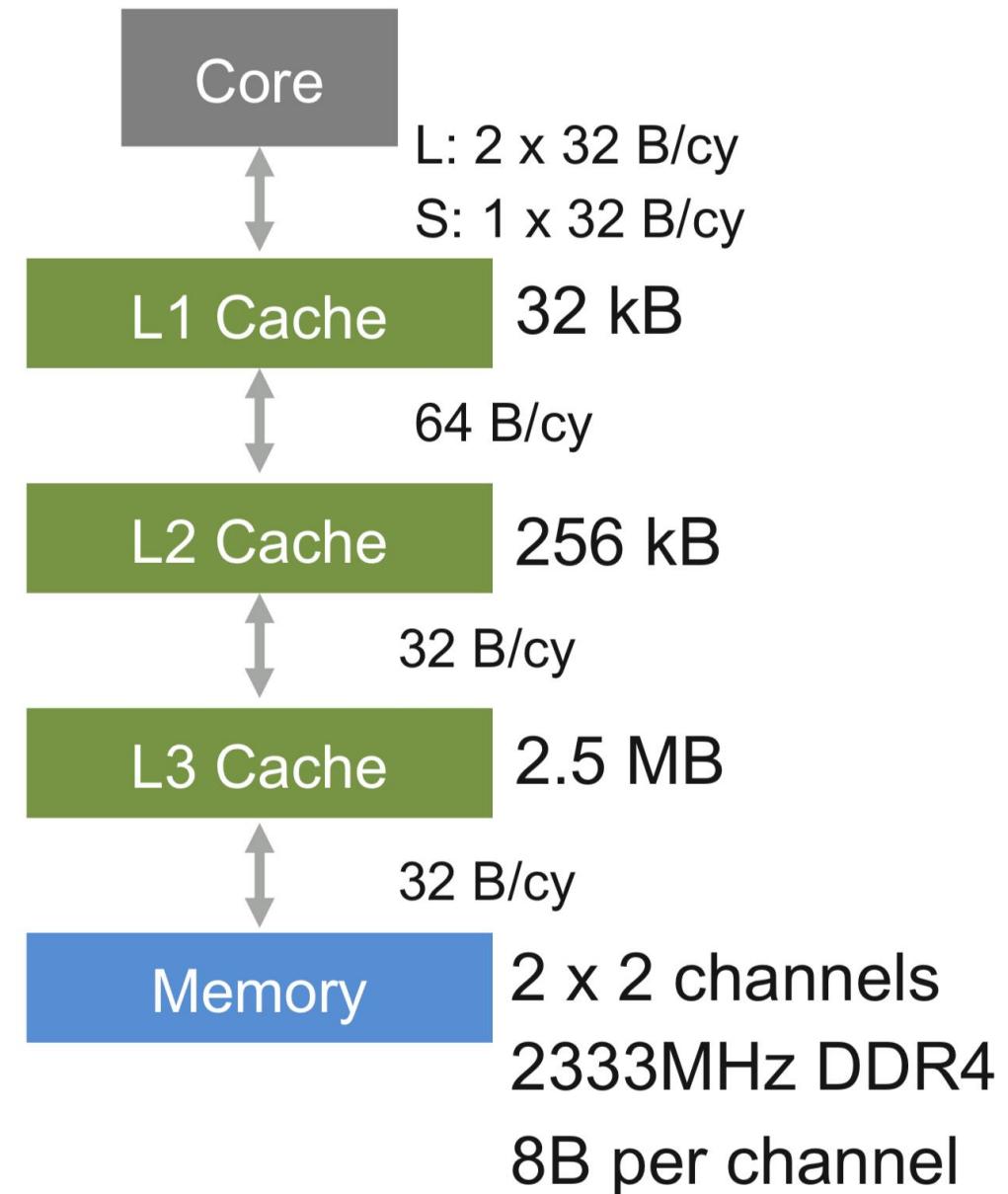
Example



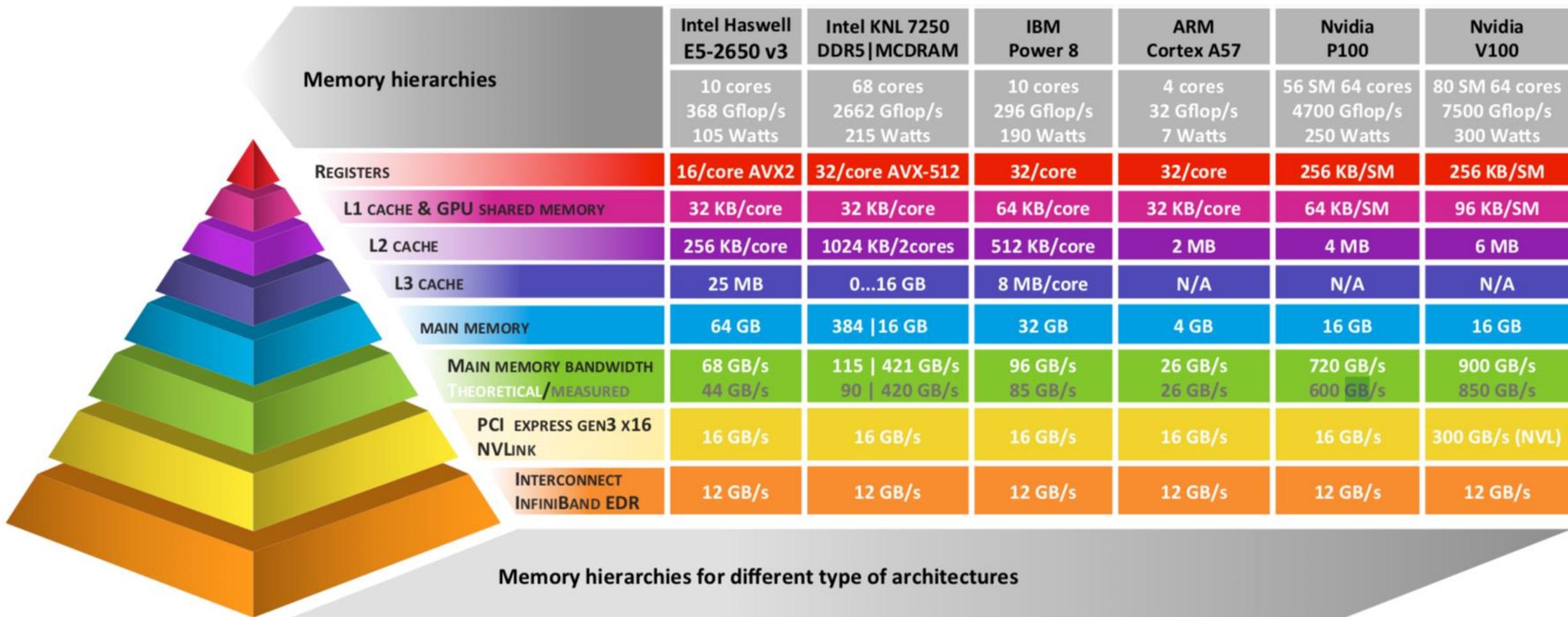
Terminology

- Cacheline (CL): Smallest unit of data that can be transferred within the memory hierarchy. On x86 usually 64 **Bytes**.
- Cacheline states
 - Modified (dirty): CL is only in current cache and does not match main memory
 - Exclusive (clean): CL is only in current cache and matches main memory
 - Shared (clean): CL is unmodified and may be stored in other caches
 - Invalid: CL is unused
- Cacheline evict: A cache miss triggers a CL allocation. The replaced CL is evicted from the cache.

Haswell @ 2.3GHz



Some numbers

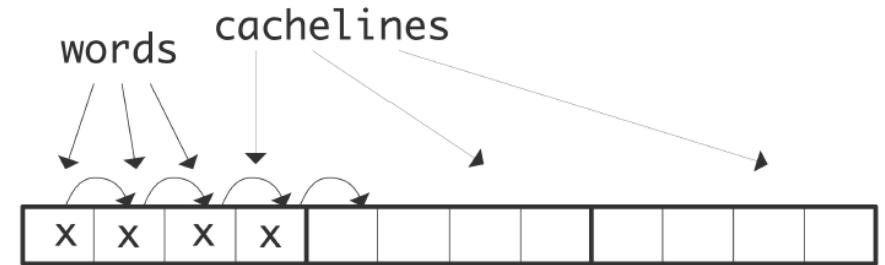


Cache line and stride

- An efficient program tries to use the other items on the cache line, since access to them is effectively free.
- This phenomenon is visible in code that accesses arrays by **stride**: elements are read or written at regular intervals.

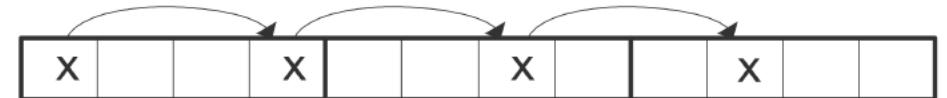
Perfect
use

```
for (i=0; i<N; i++)  
    ... = ... x[i] ...
```



Cache here
is useless

```
for (i=0; i<N; i+=stride)  
    ... = ... x[i] ...
```



Locality and Cache misses

Cache memory works well when applications exhibit spatial and/or temporal locality

- Spatial locality: Accessing adjacent memory locations is OK
- Temporal locality: Repeatedly accessing the same memory location(s) is OK

But, obviously, the first time you reference data you will always incur a cache miss

- This is called **compulsory cache miss**
- The second type of cache misses is due to the size of your working set: a **capacity cache miss** is caused by data having been overwritten because the cache can simply not contain all your problem data.
- The third type is **conflict misses**, caused by one data item being mapped to the same cache location as another, while both are still needed for the computation, and there would have been better candidates to evict.
- In a multicore system there is a fourth type, the invalidation miss. This happens if an item in cache has become invalid because another core changed the value of the corresponding memory address.

Temporal Locality

- This describes the use of a data element within a short time of its last use.
- Since most caches have an LRU replacement policy, if in between the two references less data has been referenced than the cache size, the element will still be in cache and therefore be quickly accessible.
- With other replacement policies, such as random replacement, this guarantee can not be made.
- Which example is likely to provide higher performance?

```
for (loop=0; loop<10; loop++) {  
    for (i=0; i<N; i++) {  
        ... = ... x[i] ...  
    }  
}
```

Subsequent use of $x[i]$ are spaced too far for large N

```
for (i=0; i<N; i++) {  
    for (loop=0; loop<10; loop++) {  
        ... = ... x[i] ...  
    }  
}
```

Memory accesses are reduced to 1/10

Details (just a sketch)

Cache details in
/sys/devices/system/cpu/cpu0/cache/
(one dir per cache level)

Line of 64 bytes

perf stat -e task-clock,cycles,instructions, cache-references, cache-misses,branches,faults ./<executable>

```
#include <stdio.h>
#include <stdlib.h>

#define N 10000000

int main() {
    float* x;
    int i, loop;
    x = (float*) malloc(N*sizeof(float));

    for(i=0; i<N; ++i) x[i]=1.0;
    for(i=0; i<N; ++i) {
        for (loop = 0; loop < 10; loop++) {
            x[i] = x[i]+1.0;
        }
    }
    return(0);
}
```



Cycle on N followed by loop

```
248,317799 task-clock
 922294038 cycles
1663682890 instructions
 781000 cache-references
 600226 cache-misses
140667844 branches
   678 faults

#      0,998 CPUs utilized
#      3,714 GHz
#     1,80  insns per cycle
#     3,145 M/sec
#    76,854 % of all cache refs
#   566,483 M/sec
#     0,003 M/sec

0,248922993 seconds time elapsed
```

Cycle on loop followed by N

```
278,346717 task-clock
1034468468 cycles
1594211469 instructions
1195224 cache-references
 930542 cache-misses
110740474 branches
   678 faults

#      0,997 CPUs utilized
#      3,716 GHz
#     1,54  insns per cycle
#     4,294 M/sec
#    77,855 % of all cache refs
#   397,851 M/sec
#     0,002 M/sec

0,279143032 seconds time elapsed
```

Details on the performance metrics

<https://www.intel.it/content/www/it/it/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html>

Spatial Locality

Matrix-matrix product: $2n^3$ ops, $2n^2$ data (if we have $n \times n$ matrices)

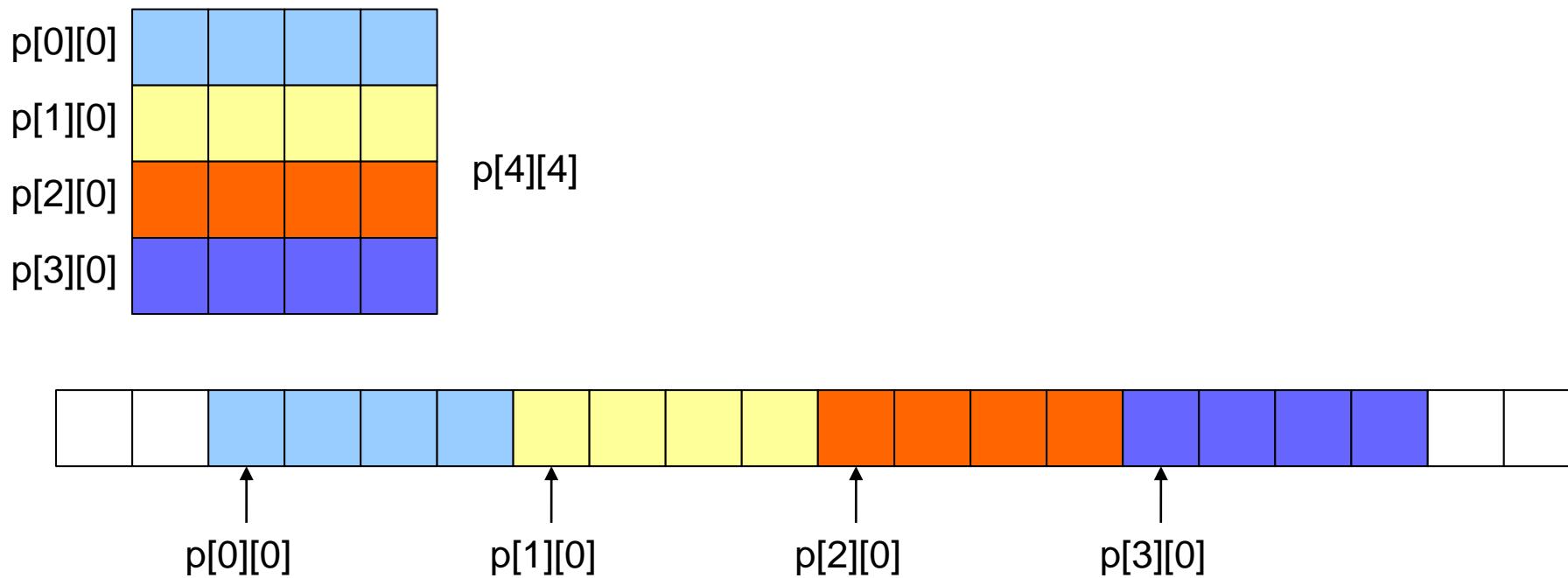
- Is it a cache-aware algorithm?

$$\begin{matrix} & j \\ i & \bullet \\ \hline C & \end{matrix} = \begin{matrix} & i \\ & \text{---} \\ \hline A & \end{matrix} \quad = \quad \begin{matrix} & j \\ & | \\ \hline B & \end{matrix}$$

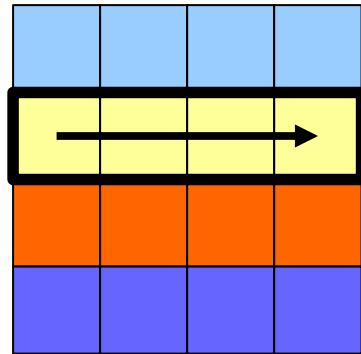
```
for i=1..n  
  for j=1..n  
    for k=1..n  
      c[i, j] += a[i, k] * b[k, j]
```

Matrix representation

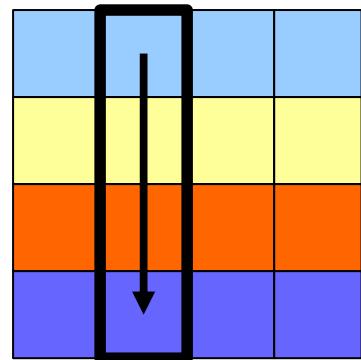
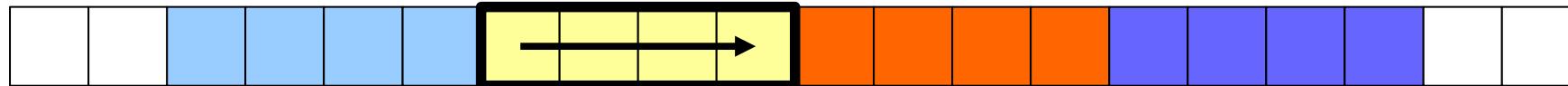
- Matrices in C are stored in *row-major order*
 - Elements of each row are contiguous in memory
 - Adjacent rows are contiguous in memory



Row-wise vs column-wise access



Row-wise access is OK: the data is contiguous in memory, so the cache helps (spatial locality)



Column-wise access is **NOT OK**: the accessed elements are not contiguous in memory (*strided* access) so the cache does NOT help



Spatial Locality

Matrix-matrix product: $2n^3$ ops, $2n^2$ data (if we have $n \times n$ matrices)

- $c[i,j]$ is loaded and stored only once, thus a register
- $a[i,k]$ is accessed by row, so spatial locality (n accesses).
- $b[k,j]$, loaded every time, so $n^2 \times n$ times



$$\begin{matrix} & j \\ i & \bullet \\ \end{matrix} = \begin{matrix} i & \text{---} \\ \end{matrix} \begin{matrix} & j \\ & | \\ \end{matrix}$$

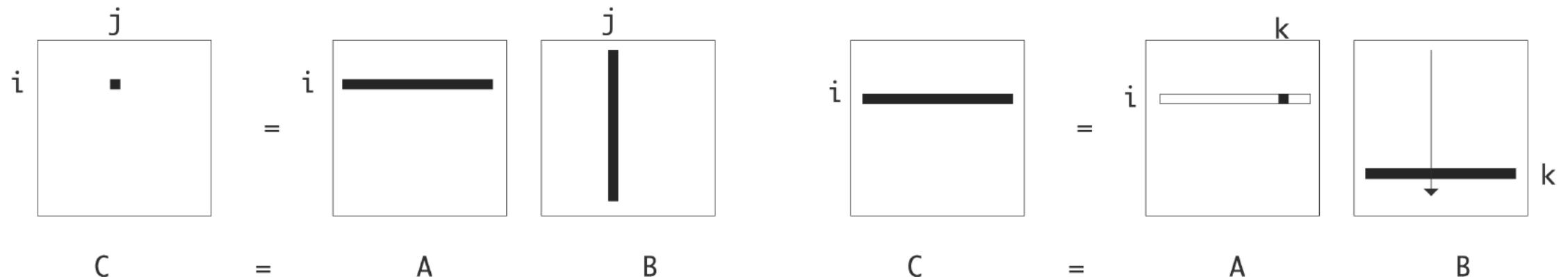
C $=$ A B

```
for i=1..n  
  for j=1..n  
    for k=1..n  
      c[i,j] += a[i,k]*b[k,j]
```

Better algorithm

```
for i=1..n  
  for j=1..n  
    for k=1..n  
      c[i, j] += a[i, k]*b[k, j]
```

```
for i=1..n  
  for k=1..n  
    for j=1..n  
      c[i, j] += a[i, k]*b[k, j]
```



- $c[i,j]$ now requires n^3 **store** operation wrt n^2
- A and b exploits spatial and temporal locality

Loop tiling

FROM

```
for (i=0; i<n; i++)  
    ...
```

Sometimes performance can be increased by breaking up a loop into two nested loops, an outer one for the blocks in the iteration space, and an inner one that goes through the block. This is known as loop tiling: the (short) inner loop is a tile, many consecutive instances of which form the iteration space.

TO

```
bs = ...          /* the blocksize */  
nblocks = n/bc /* assume that n is a multiple of bs */  
for (b=0; b<nblocks; b++)  
    for (i=b*bs, j=0; j<bs; i++, j++)  
        ...
```

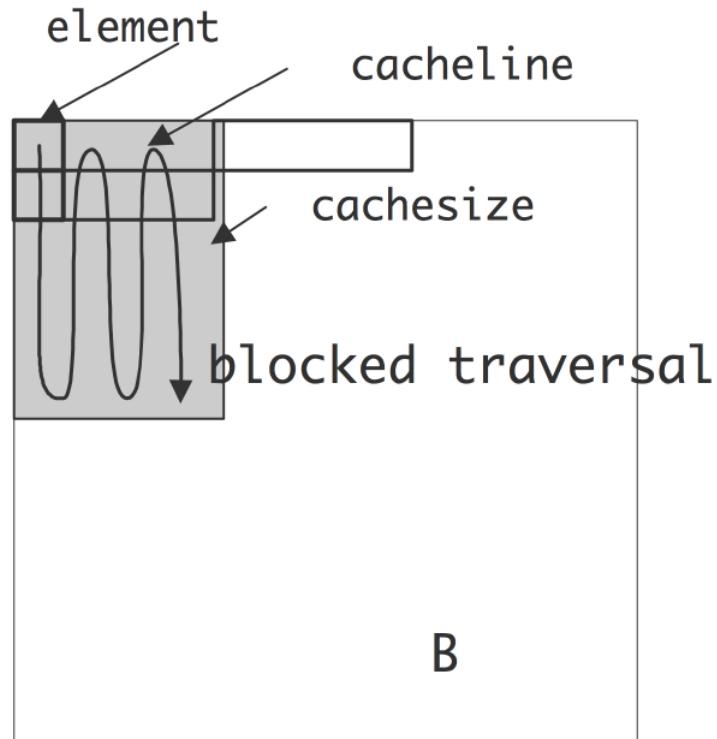
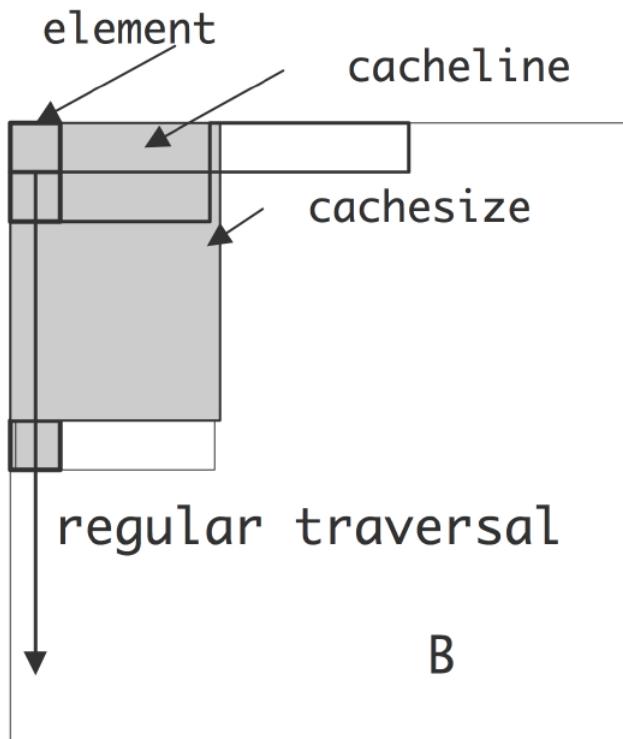
WHY?

Because a data structure (i.e. array, matrix) is divided into blocks that fit in cache!

Example: matrix transposition

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        A[i][j] += B[j][i];
```

```
for (int ii=0; ii<N; ii+=blocksize)
    for (int jj=0; jj<N; jj+=blocksize)
        for (int i=ii*blocksize; i<MIN(N, (ii+1)*blocksize); i++)
            for (int j=jj*blocksize; j<MIN(N, (jj+1)*blocksize); j++)
                A[i][j] += B[j][i];
```



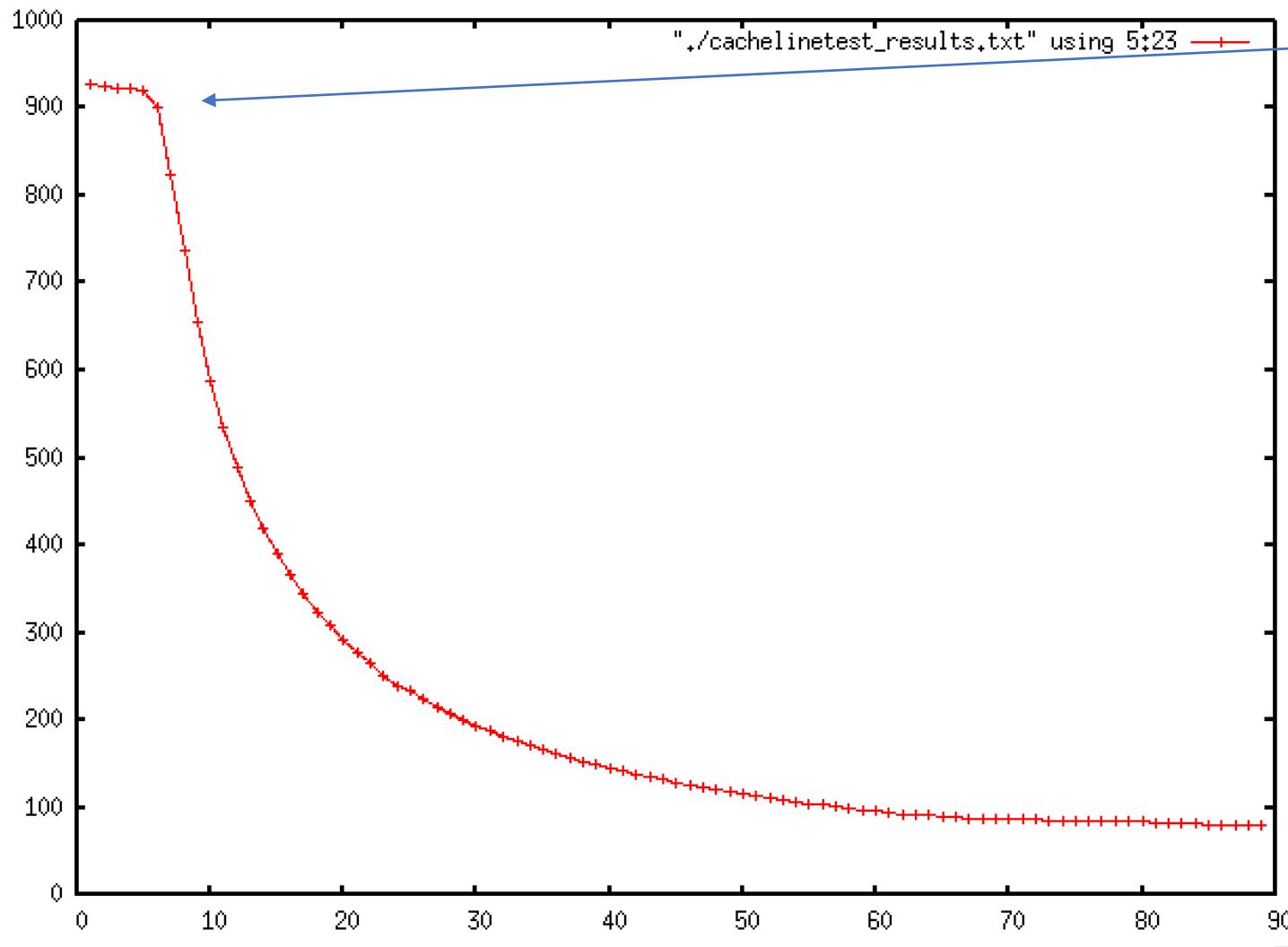
And the matrix-matrix multiplication?

Have a good read!

- <https://scialert.net/fulltextmobile/?doi=rjit.2011.61.67>
- <http://www.cs.utexas.edu/users/pingali/CS378/2008sp/papers/gotoPAPER.pdf>

Let's try a test

- Compile and run the code in `cacheline*.c`
`gcc -fopenmp -O2`
- Plot a graph using gnuplot as follows:-
- Launch gnuplot which should return a prompt
- Issue the following line
 - plot "cachelinetest_results.txt" using 5:23 with linespoints
 - plot "./cachelinetest_double_results.txt" using 5:23 with linespoints
 - plot "./cachesizebwidth.txt" using 4:22 with linespoints
- where does the graph flatten out?



We use L1

Caches become useless

Conclusions

- A basic knowledge of the computers' architecture is fundamental for understanding the performance achievable by an application
- ILP is the first kind of parallelism we can exploit in sequential applications
- A proper cache exploitation is a fundamental factor for every application
- BUT the main solution for achieving high performance is represented by the following two kinds of parallelism, data parallelism and task parallelism.

Introduction to Parallel Computing

Parallel Architectures

Levels of parallelism

- Instruction level (e.g. FMA= fused multiply and add).
- Vector processing (e.g. data parallelism)
- Hyperthreading(e.g. 4 hardware threads/core for Intel KNL, 8 for PowerPC).
- Cores / processor (e.g. 18 for Intel Broadwell)
- Processors (or sockets) / node
- Processors + accelerators (e.g. CPU+GPU)
- Nodes in a system
- Many systems (e.g. Grid, Cloud)

To reach the maximum (*peak*) performance of a parallel computer,
all levels of parallelism need to be exploited.

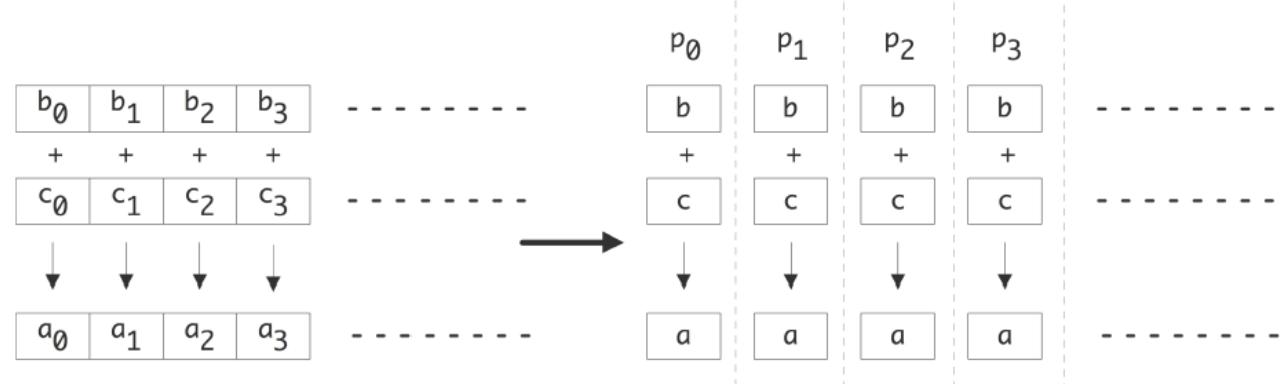
Introduction

- A general (and too simple) definition of parallel computers is «machines with more than one CPU that can be set to work on the same problem».
- But parallelism is hard to define precisely, since it can appear on several levels
 - inside a CPU several instructions can be ‘in flight’ simultaneously. Is the ILP, and it is outside explicit user control: it derives from the compiler and the CPU deciding which instructions can be processed simultaneously.
 - At the other extreme is the sort of parallelism where more than one instruction stream is handled by multiple processors. This type of parallelism is typically explicitly scheduled by the user.

Introduction

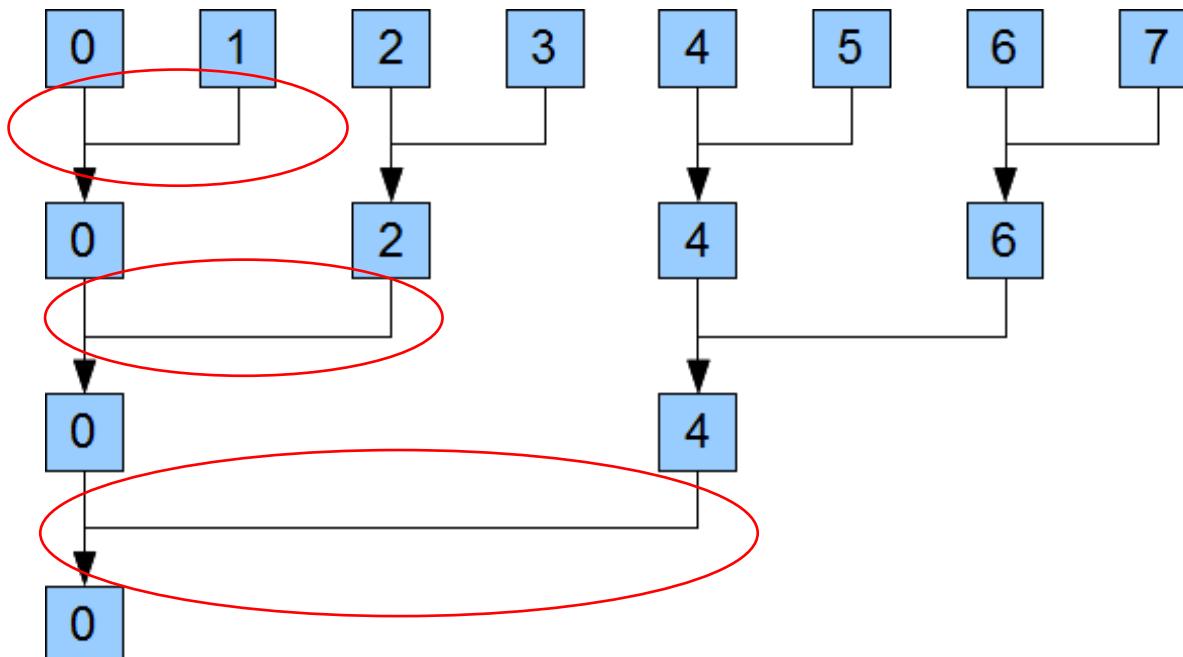
- In scientific (and also non scientific) codes, there is often a large amount of work to be done, and it is often regular to some extent, i.e. the same operation being performed on many data.
- If each operation takes a unit time, the original algorithm takes time t on n elements.
- The parallel execution on p processors takes, most of the time $t > t_p > n/p$, with $p \leq n$
- How can I write such application?

```
for (i=0; i<n; i++)
    a[i] = b[i] + c[i];
```



Parallel reduction

- Let us consider summing the elements of a vector, i.e. the reduction
- Seems difficult to parallelize in this form
- But



```
s = 0;  
for (i=0; i<n; i++)  
    s += x[i]
```

```
for (s=2; s<2*n; s*=2)  
    for (i=0; i<n-s/2; i+=s)  
        x[i] += x[i+s/2]
```

If $n = 8$ (and $p = 8$)
 $S = 2, l = 0, < 7, += 2$ i.e. 0,2,4,6
 $S = 4, l = 0, < 4, += 4$
 $S = 8, l = 0, < 4, += 8$

$n/p * \log_2 n$
3 sums wrt 7

Introduction

Even from this simple example we can see some of the characteristics of parallel computing:

- Sometimes algorithms need to be rewritten to make them parallel
- A parallel algorithm may not show perfect speedup
- if each processors has its x_i (or even the full vector) in memory, there is the need to communicate data for the parallel reduction, not for the vector sum.

The prefix-sum problem

Given `int[] input`, produce `int[] output` where
`output[i]` is the sum of
`input[0]+input[1]+...+input[i]`

Sequential can be a CS1 exam problem:

```
int[] prefix_sum(int[] input){  
    int[] output = new int[input.length];  
    output[0] = input[0];  
    for(int i=1; i < input.length; i++)  
        output[i] = output[i-1]+input[i];  
    return output;  
}
```

Does not seem parallelizable

- Work: $O(n)$, Span: $O(n)$
- This *algorithm* is sequential, but a *different algorithm* has Work: $O(n)$, Span: $O(\log n)$

Parallel prefix-sum

- The parallel-prefix algorithm does two passes
 - Each pass has $O(n)$ work and $O(\log n)$ span
 - So in total there is $O(n)$ work and $O(\log n)$ span
 - So like with array summing, parallelism is $n/\log n$
 - An exponential speedup
- First pass builds a tree bottom-up: the “up” pass
- Second pass traverses the tree top-down: the “down” pass

Historical note:

- Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977

The algorithm, part 1

1. Up: Build a binary tree where
 - Root has sum of the range $[x, y]$
 - If a node has sum of $[lo, hi)$ and $hi > lo$,
 - Left child has sum of $[lo, middle)$
 - Right child has sum of $[middle, hi)$
 - A leaf has sum of $[i, i+1)$, i.e., `input[i]`

This is an easy fork-join computation: combine results by actually building a binary tree with all the range-sums

- Tree built bottom-up in parallel
- Could be more clever with an array like with heaps

Analysis: $O(n)$ work, $O(\log n)$ span

The algorithm, part 2

2. Down: Pass down a value **fromLeft**

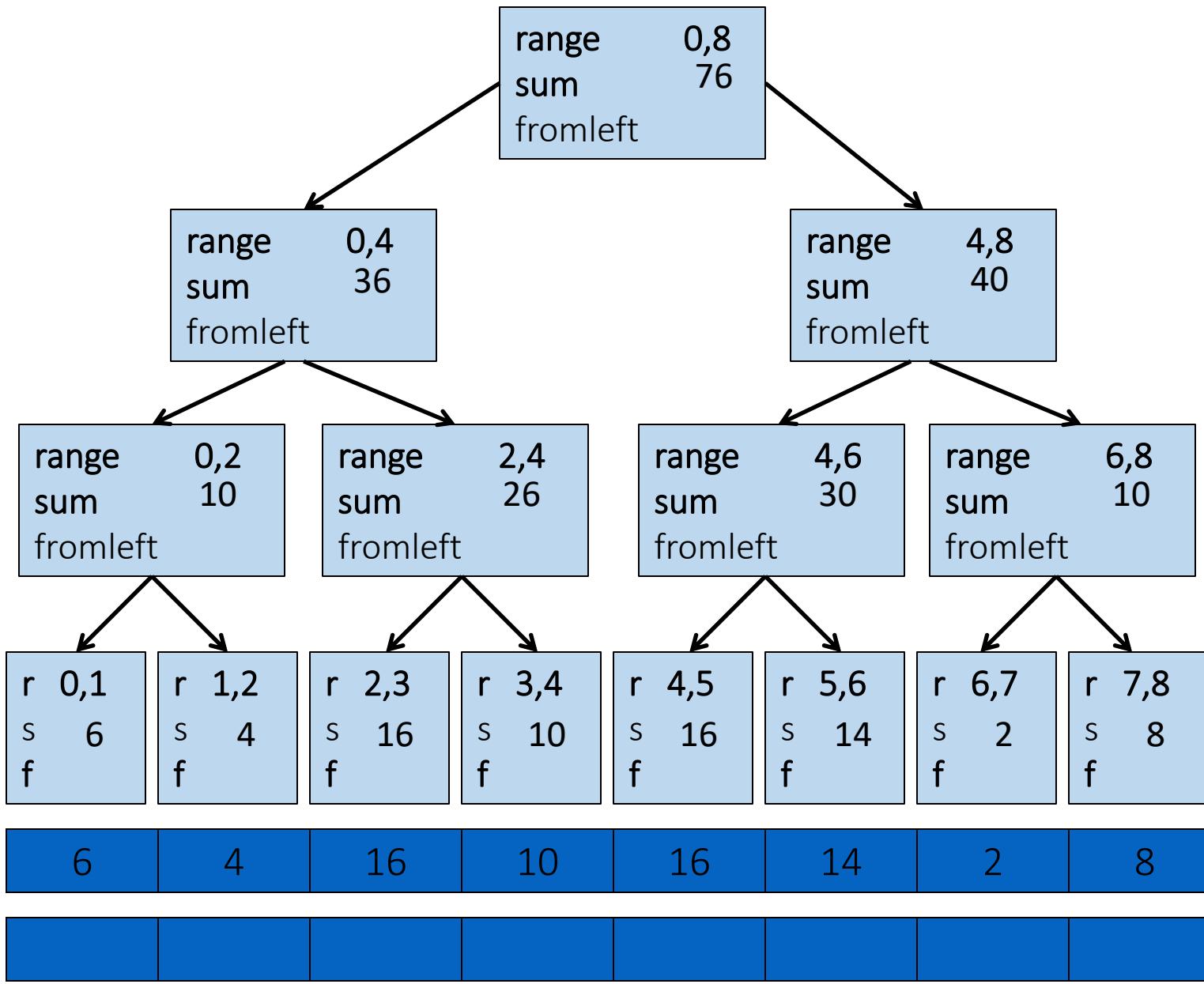
- Root given a **fromLeft** of 0
- Node takes its **fromLeft** value and
 - Passes its left child the same **fromLeft**
 - Passes its right child its **fromLeft** plus its left child's **sum** (as stored in part 1)
- At the leaf for array position **i**,
output[i] = fromLeft + input[i]

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result

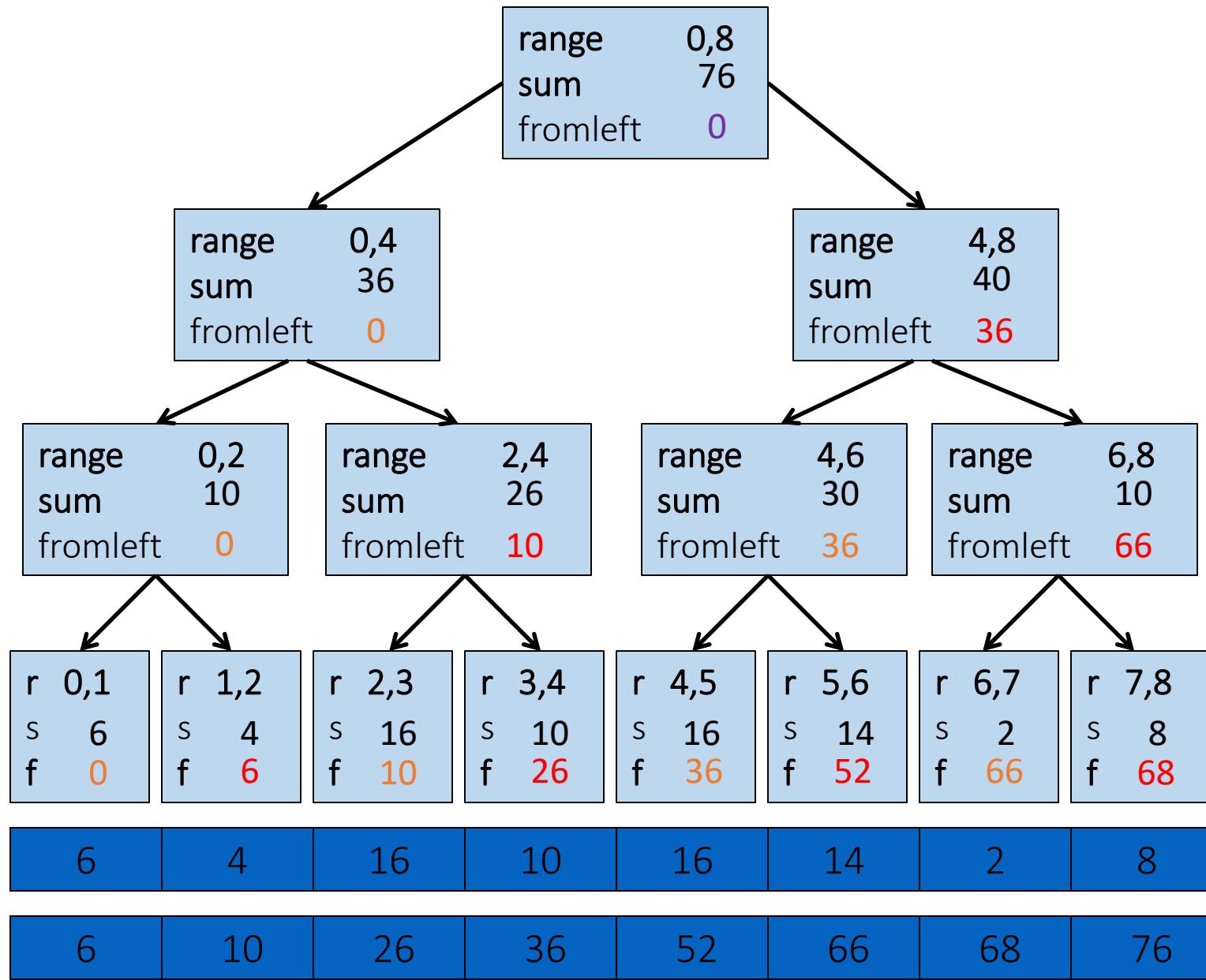
- Leaves assign to **output**
- Invariant: **fromLeft** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\log n)$ span

Example



Example



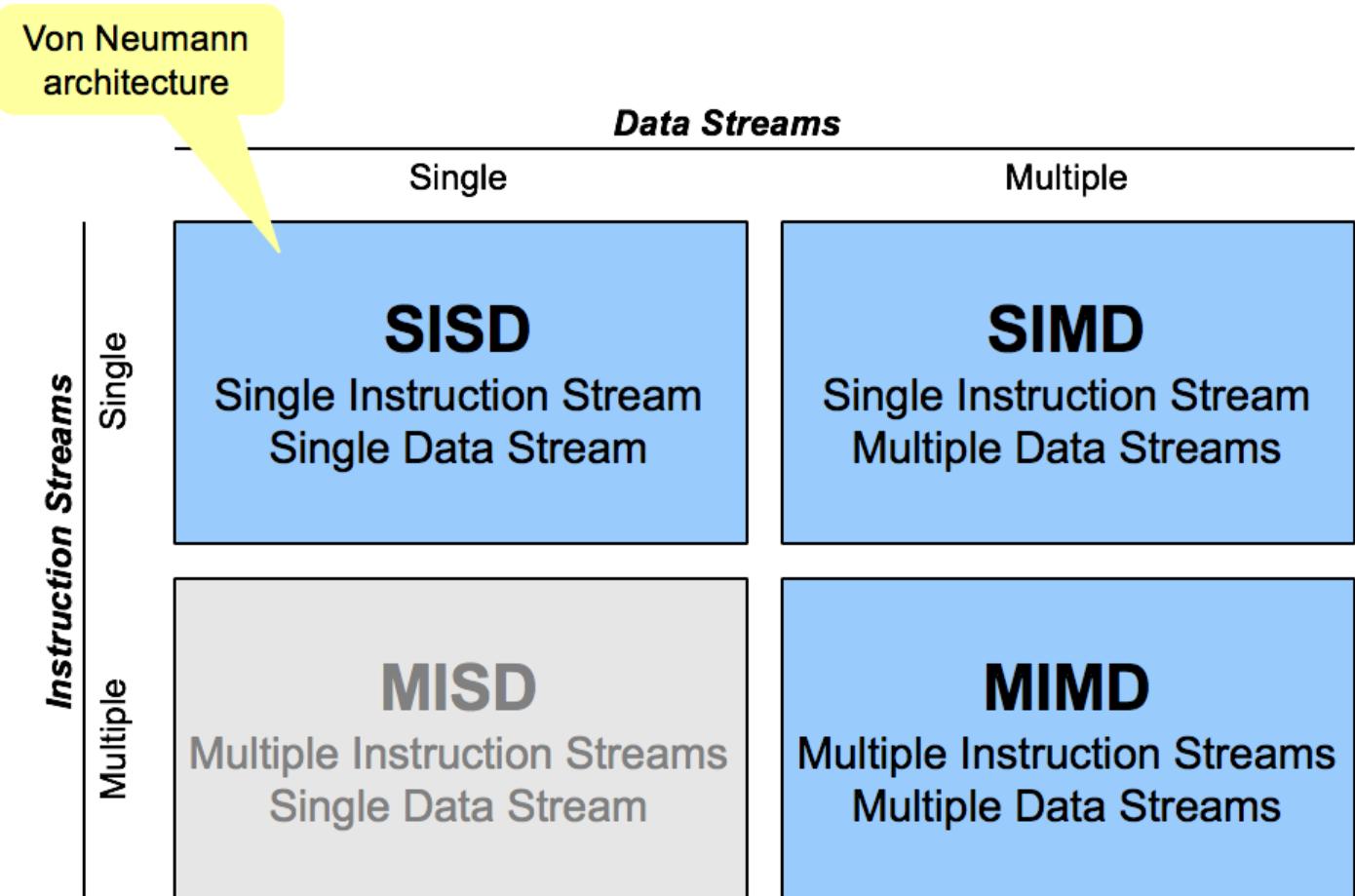
Which kind of parallelism

- **Data parallelism:** the same operation/instruction is applied in parallel to many data elements. This is a common scenario in scientific computing: parallelism often stems from the fact that a data set (vector, matrix, graph, . . .) is spread over many processors, each working on its part of the data.
- **Task parallelism:** the other cases, e.g. the **same a (sub)program** is applied in parallel to many data elements.
- We already saw ILP

```
...
if CPU="a" then
    do task "A"
else if CPU="b" then
    do task "B"
end if
...
```

Parallel Architectures

- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of **Instruction Stream** and **Data Stream**.



- Other taxonomies, e.g.

[https://www.researchgate.net/publication/316511032 Parallel Computer Architectural Schemes](https://www.researchgate.net/publication/316511032)

Why are parallel architectures important?

- There is no "typical" parallel computer: different vendors use different architectures
- See <https://www.top500.org/statistics/list/>
- There is currently no “universal” programming paradigm that fits all architectures
 - Parallel programs must be tailored to the underlying parallel architecture
 - The architecture of a parallel computer limits the choice of the programming paradigm that can be used (e.g. GPU vs distributed memory systems)

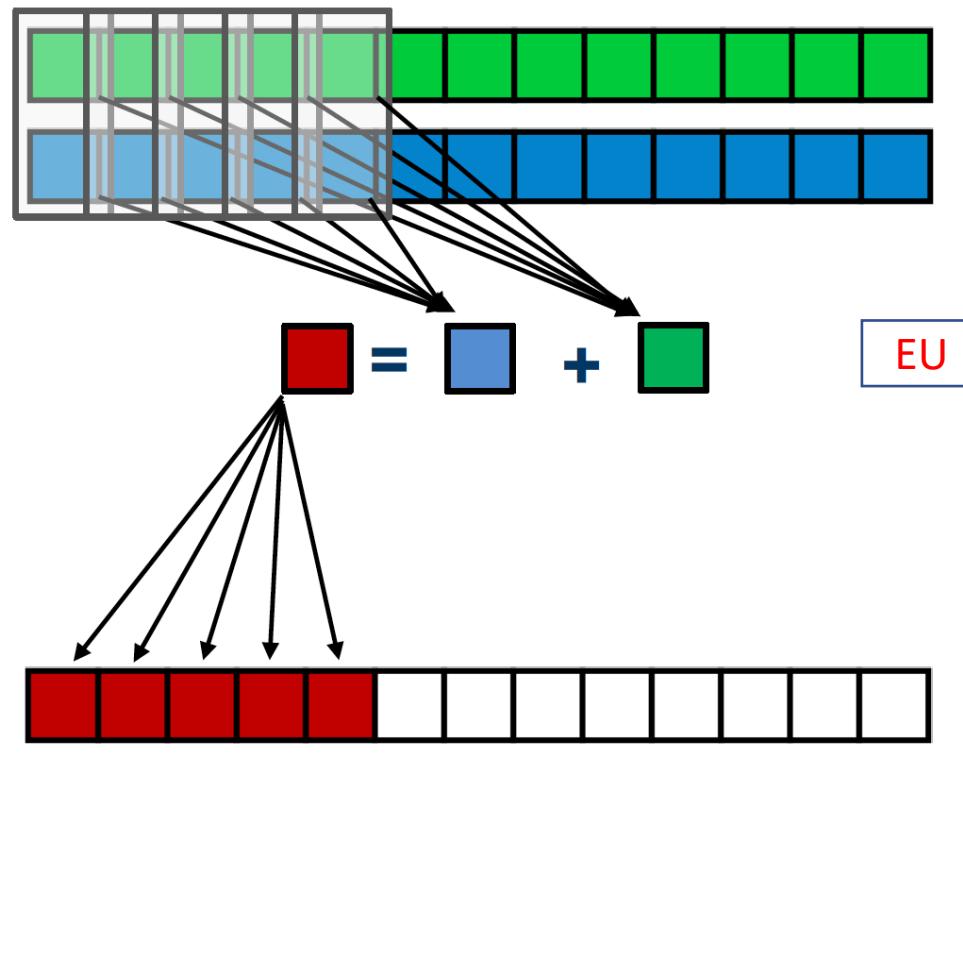
SIMD

```
double *A, *B, *C;  
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register widths

- 1 operand
- 2 operands (SSE)
- 4 operands (AVX)
- 8 operands (AVX512)

Scalar execution



```
double *A, *B, *C;  
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register widths

- 1 operand

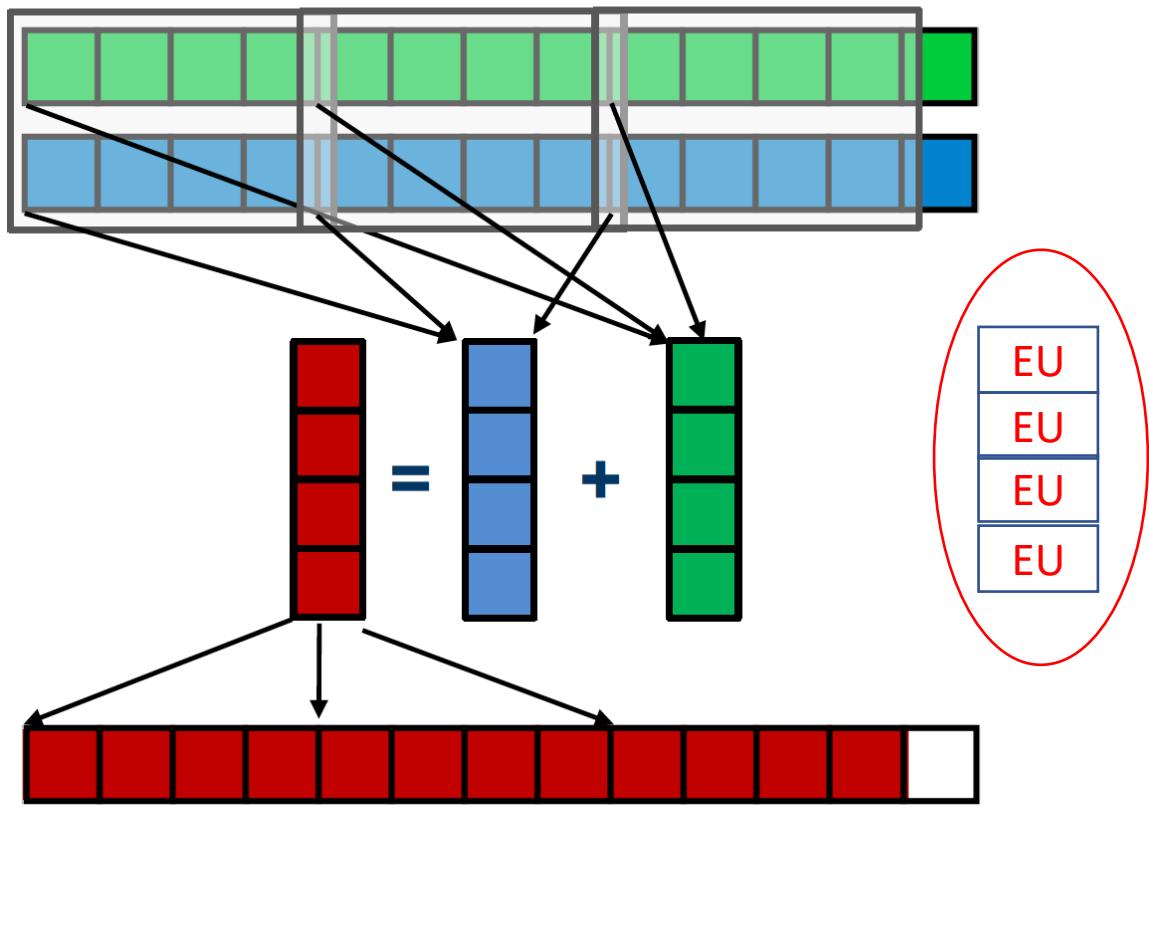
- 2 operands (SSE)

- 4 operands (AVX)

- 8 operands (AVX512)

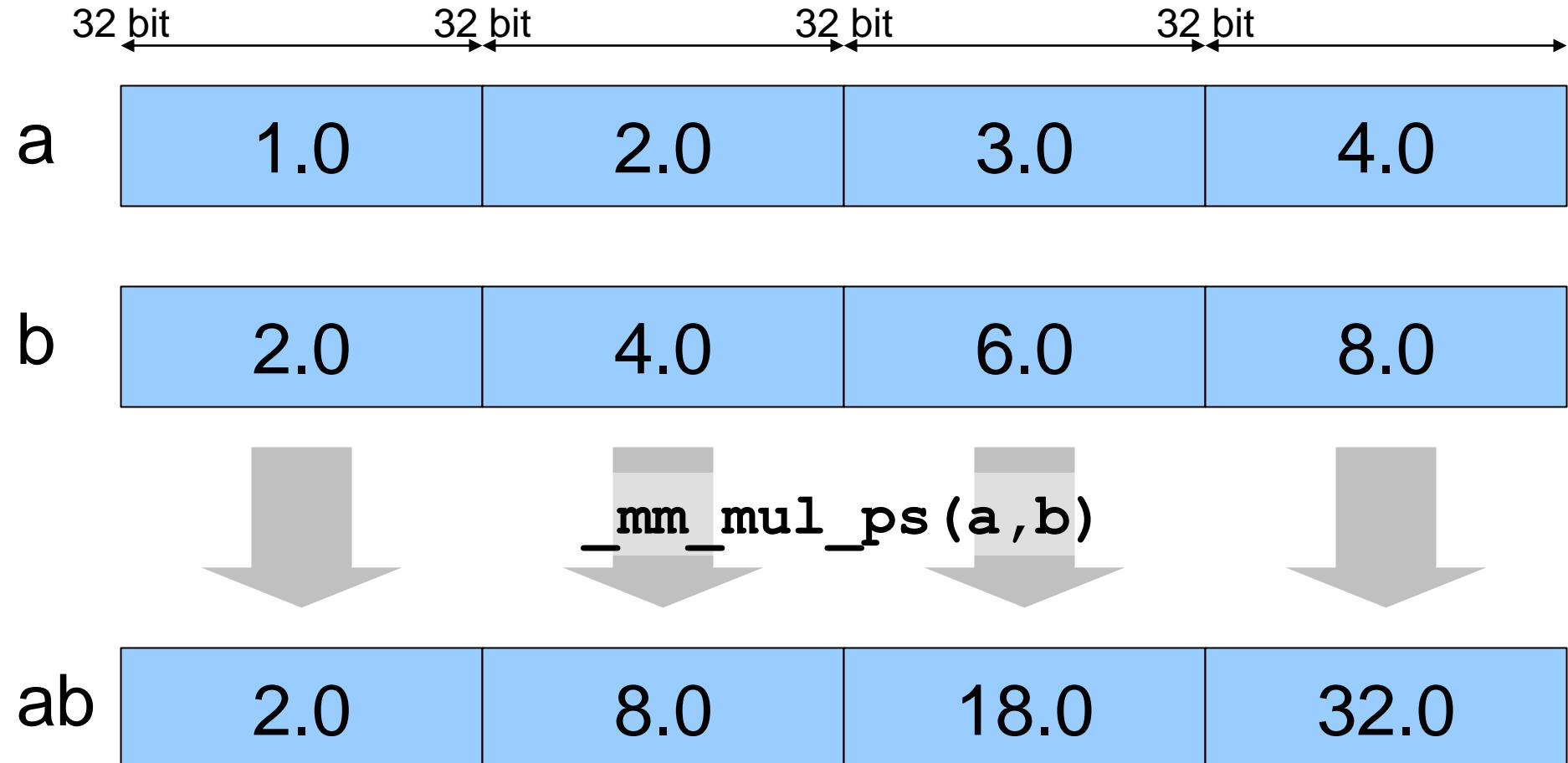

SIMD

SIMD execution



Example a^*b

```
__m128 a = _mm_set_ps( 1.0, 2.0, 3.0, 4.0 );
__m128 b = _mm_set_ps( 2.0, 4.0, 6.0, 8.0 );
__m128 ab = _mm_mul_ps(a, b);
```

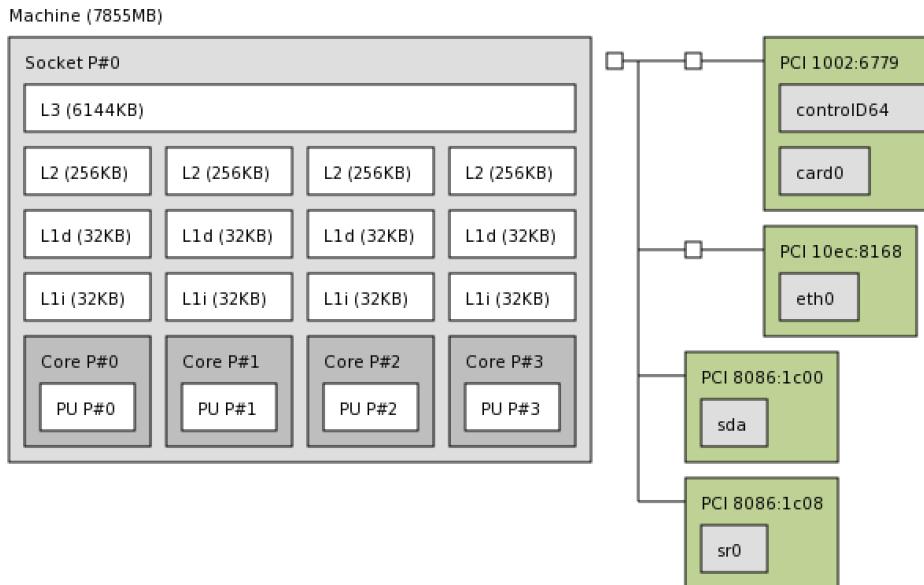


<https://felix.abecassis.me/2011/09/cpp-getting-started-with-sse/>

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

/proc/cpuinfo

<https://ark.intel.com/content/www/it/it/ark/products/52209/intel-core-i5-2500-processor-6m-cache-up-to-3-70-ghz.html>

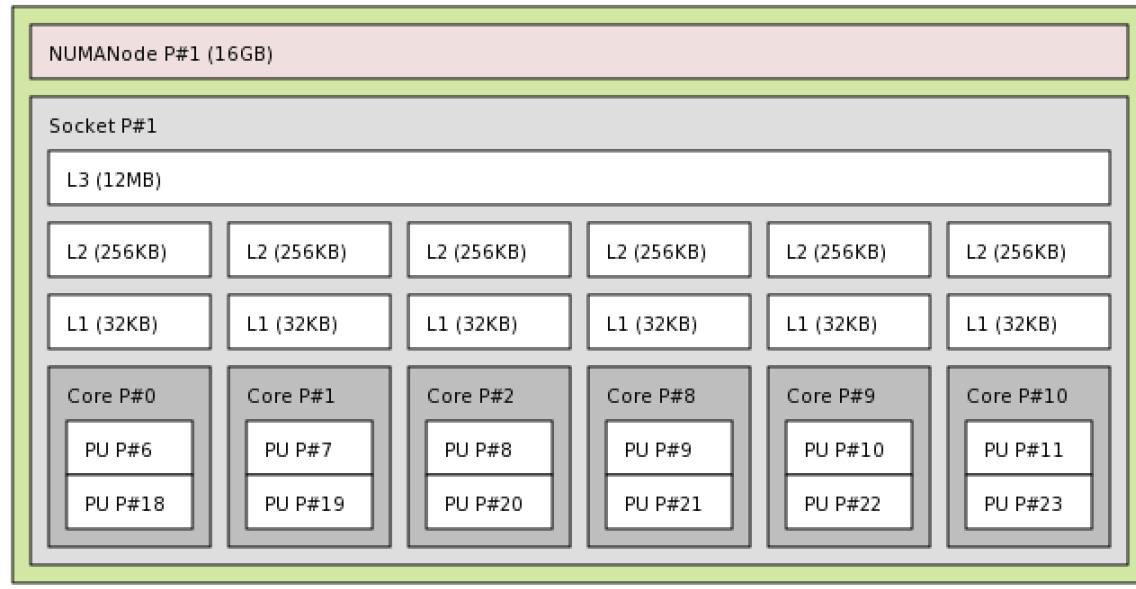
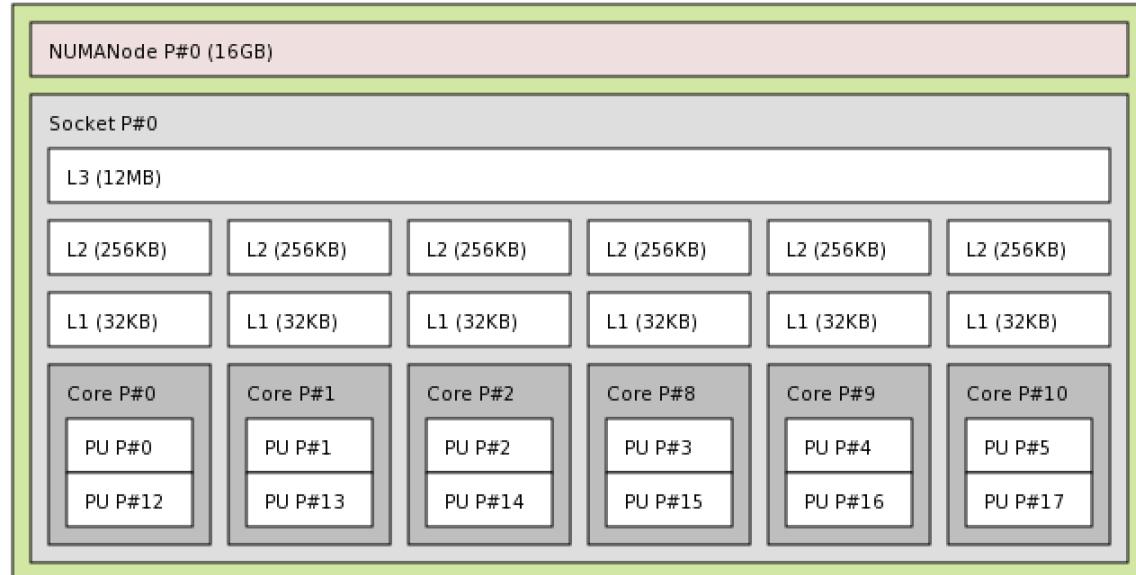


Cpuinfo, lstopo.... Many utilities to get the info
On Mac sysctl -a | grep machdep.cpu

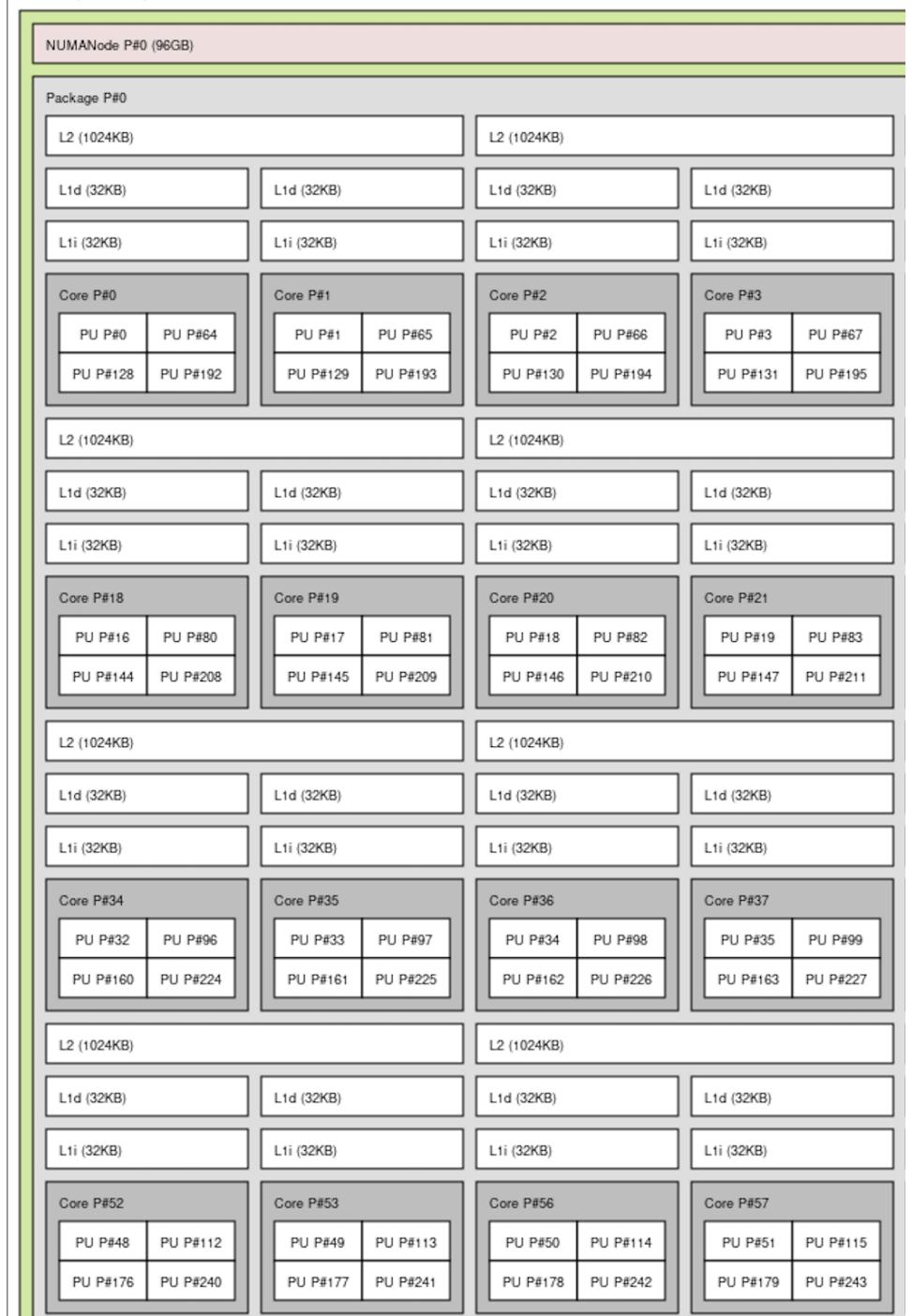
```
[dago@girasole ~]$ cat /proc/cpuinfo
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 42
model name   : Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz
stepping       : 7
cpu MHz       : 1600.000
cache size    : 6144 KB
physical id   : 0
siblings       : 4
core id        : 0
cpu cores     : 4
apicid         : 0
initial apicid: 0
fpu            : yes
fpu_exception  : yes
cpuid level   : 13
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                  cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
                  rdtscp lm constant_tsc arch_perfmon pebs bts rep_good xtopology nonstop_t
                  sc aperf mperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx
                  16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave
                  avx lahf_lm ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority
                  ept vpid
bogomips      : 6797.74
clflush size  : 64
cache_alignment: 64
address sizes  : 36 bits physical, 48 bits virtual
power management:

processor      : 1
vendor_id     : GenuineIntel
```

Machine (32GB)



Machine (96GB total)





GPU, the “miracle” architecture

- Lots of hype about incredible speedup / high performance for low cost.
- Graphics processing: identical (and fairly simple) operations on lots of pixels
- Doesn't matter when any individual pixel gets processed, as long as they all get done in the end
- GPU is a SIMD engine
...and scientific computing is often very data--parallel

Tesla P100 – Pascal

- 5.3 TFLOPS for FP64 (double)
 - 10.6 TFLOPS for FP32
 - 21.2 TFLOPS for FP16
- Less than 3 TF

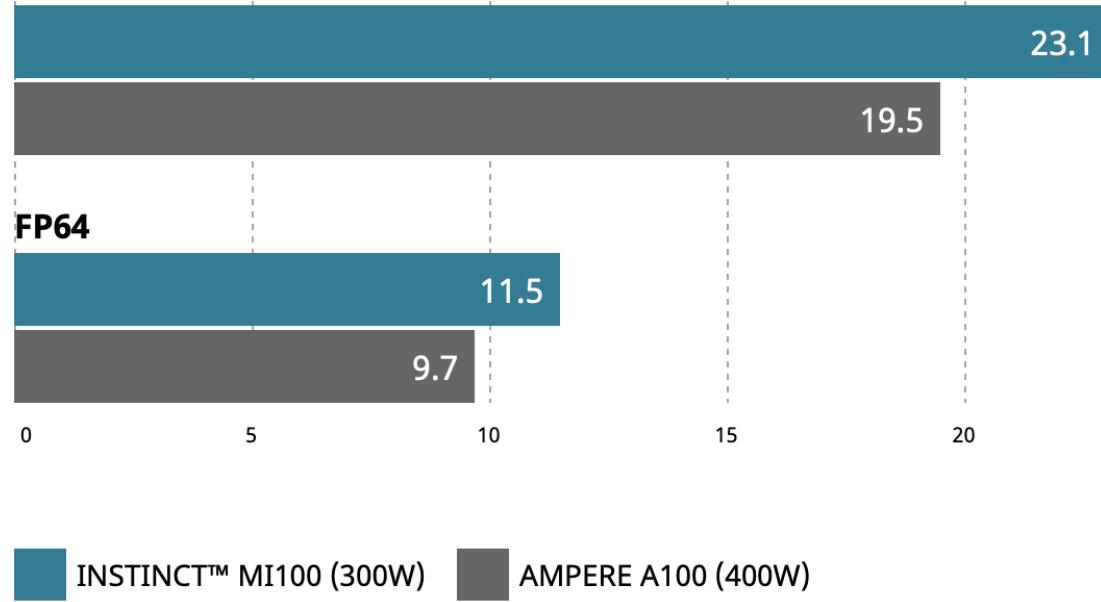
General Info	
Designer	Intel
Manufacturer	Intel
Model Number	8280
Part Number	CD8069504228001
S-Spec	SRF9P
Market	Server
Introduction	April 2, 2019 (announced) April 2, 2019 (launched)
Release Price	\$10,009.00 (tray)

<https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-cascade-lake-sp-intel-xeon-processor-scalable-family-cpus/>

Today

(Peak TFLOPS) across range of mixed-precision compute¹

FP32

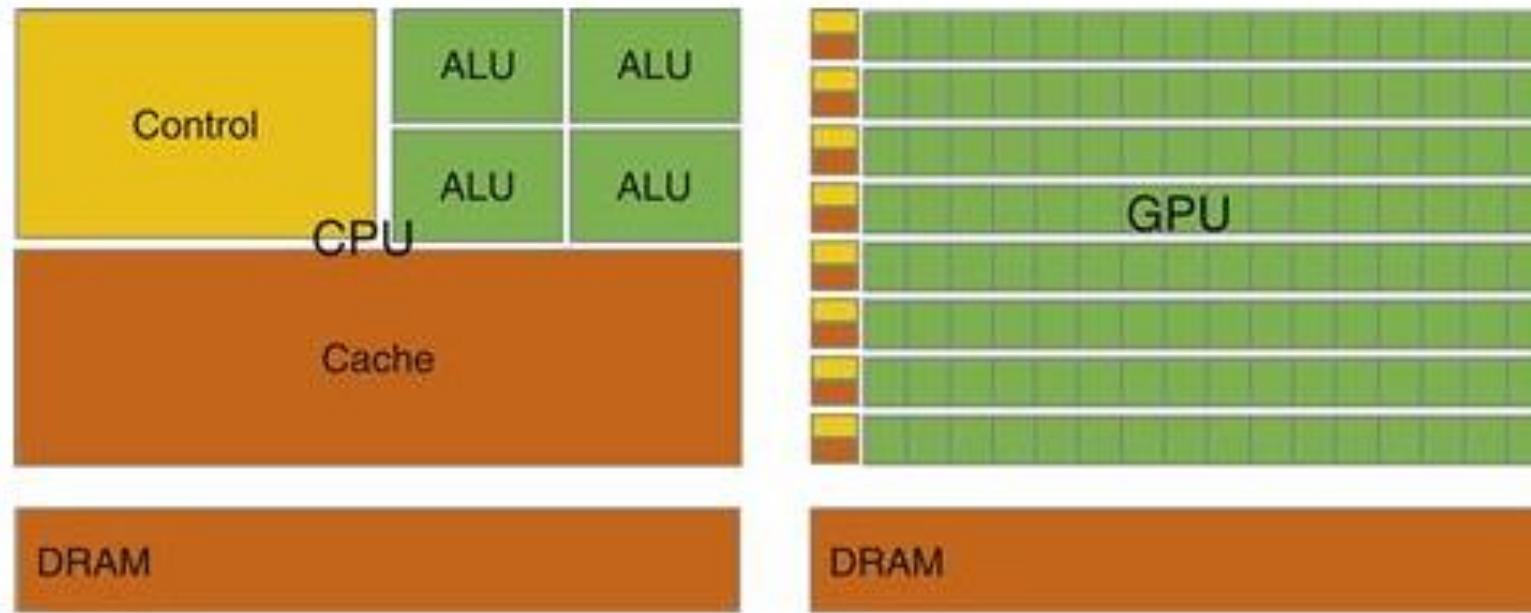


	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64			9.7 TFLOPS	
FP64 Tensor Core			19.5 TFLOPS	
FP32		19.5 TFLOPS		
Tensor Float 32 (TF32)			156 TFLOPS 312 TFLOPS*	
BFLOAT16 Tensor Core			312 TFLOPS 624 TFLOPS*	
FP16 Tensor Core			312 TFLOPS 624 TFLOPS*	
INT8 Tensor Core			624 TOPS 1248 TOPS*	
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth		1,555GB/s	1,935GB/s	1,555GB/s
			2,039GB/s	
Max Thermal Design Power (TDP)	250W	300W	400W	400W

<https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-ice-lake-sp-intel-xeon-processor-scalable-family-cpus/>

<https://www.intel.com/content/www/us/en/support/articles/000005755/processors.html>

GPUs

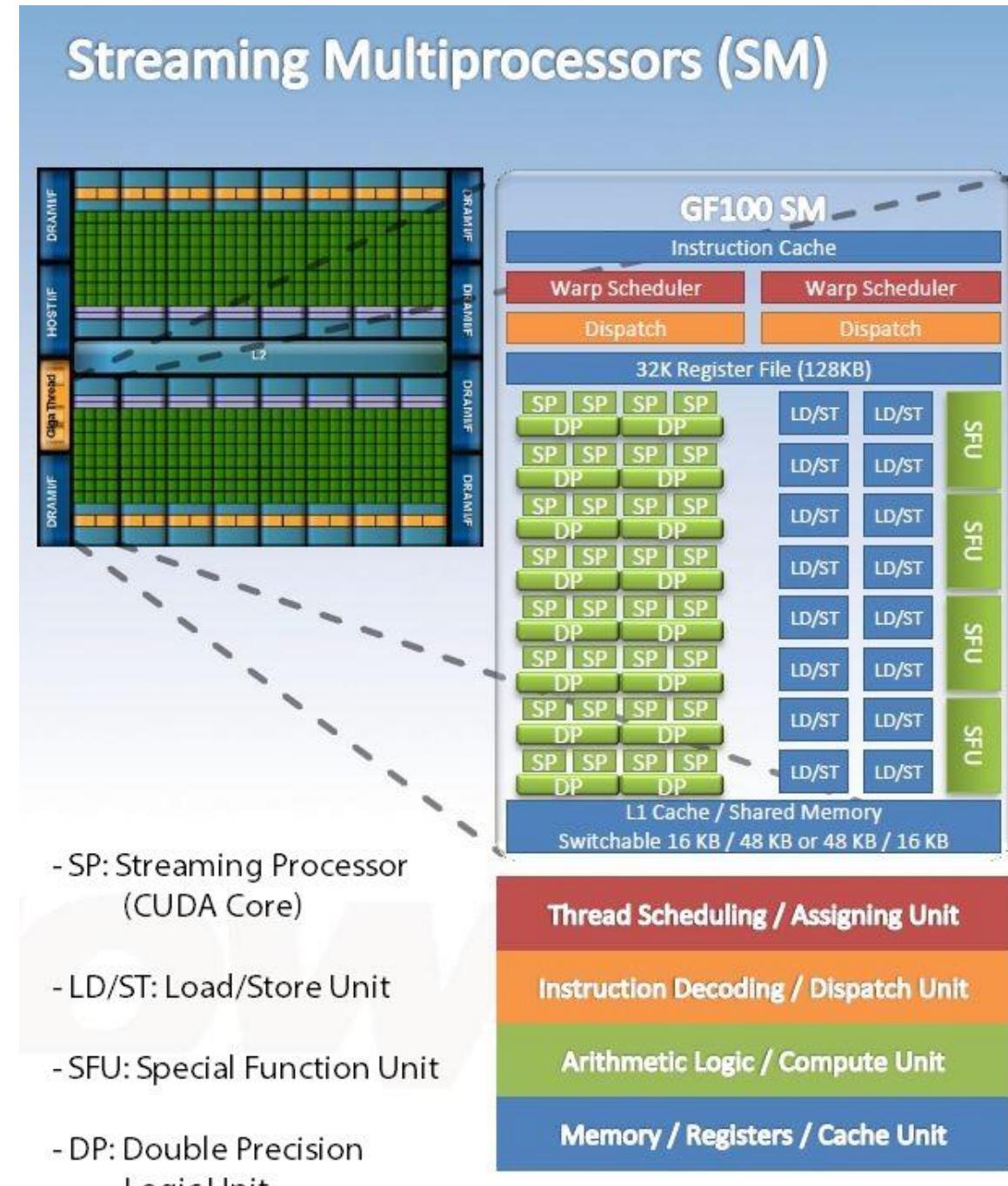


- GPU has a large number of «in order» cores;
- GPU cores **share control unit** (SIMD like style or better SIMT - Single Instruction Multiple Thread);
- GPU cores share small caches;
- GPU cores have lower frequency;

GPU is a (fine grained) parallel throughput-oriented computing engine

GPU core

- An SM contains a fetch/decode unit shared among multiple ALUs (the actual cores)
 - If there are 16 DP ALU, each instruction can operate on 16 DP values simultaneously
- Each GPU core maintains multiple execution contexts, and can switch between them at virtually zero cost
- A GPU with 16 SM x 16 DP ALU = 256 operations in parallel
- Details later in the course



Intel Xeon Phi Coprocessor

Low power device range based on Intel's Many Integrated Core (MIC) technology.

Large number of low frequency Pentium cores (e.g. 1.0 GHz) loosely connected on a chip with onboard memory.

The first commercially available Xeon Phi device Knight's Corner (KNC) could be used only as an accelerator.

Although runs standard FORTRAN and C/C++, difficult to obtain good performance.

Many application developers did not optimise codes for KNC.



PCI express connected device

Intel Knight's Corner

- 61 cores, 1.0-1.2 GHz
- 8-16 Gb RAM
- 512 bit vector unit
- 1-2 Tflops
- ring topology of cores
- With compiler option, runs standard FORTRAN or C (i.e. no CUDA or OpenCL necessary) and MPI.

Intel Xeon Phi Coprocessor

- The Intel Xeon Phi Coprocessor (IXPC) is designed to appear and operate more like a common multicore processor, though with many more cores. An IXPC has up to 61 cores, where each core implements **most** of the 64-bit x86 scalar instruction set. Instead of SSE and AVX instructions, an IXPC core has 512-bit vector instructions, which perform 16 single precision operations or eight double precision operations in a single instruction.
- Each core has a 32KB L1 instruction cache and 32KB L1 data cache, and a 512KB L2 cache. There is no cache shared across cores. The CPU clock is about 1.1GHz. With 61 cores and 16-wide vector operations, the IXPC has the equivalent of 976 GPU cores.
- The control unit can issue two instructions per clock, though instructions are issued in-order and those two instructions must be from the same thread (unlike Intel Hyperthreading). Each core stores the state of up to four threads, using multithreading to tolerate cache misses.
- The IXPC is packaged as an IO device, similar to a GPU. It has 8GB memory in current devices. The biggest potential advantage to the IXPC is its programmability using existing multi-core programming models. In many if not most cases, you really can just recompile your program and run it natively on an IXPC using MPI and/or OpenMP; however, to get good performance, you will likely have to tune or refactor your application.

Intel Xeon PHI -Knight's Landing (KNL)

- Is a standard Intel Architecture standalone processor
- It triples (3x) the performance of KNC, providing 3 Tflops DP and 6 Tflops SP peak performance
- Programming for Knights Landing is equivalent to programming an Intel® Xeon processor, i.e. both can execute the same binary executable programs
- ...but with extra attention on exploiting lots of parallelism because of the uniquely high degree of scaling possible with Knights Landing

AVX-512PF

AVX-512ER

AVX-512CD

AVX-512F

BMI

AVX2

AVX

SSE...

MMX

x87

Knights Landing

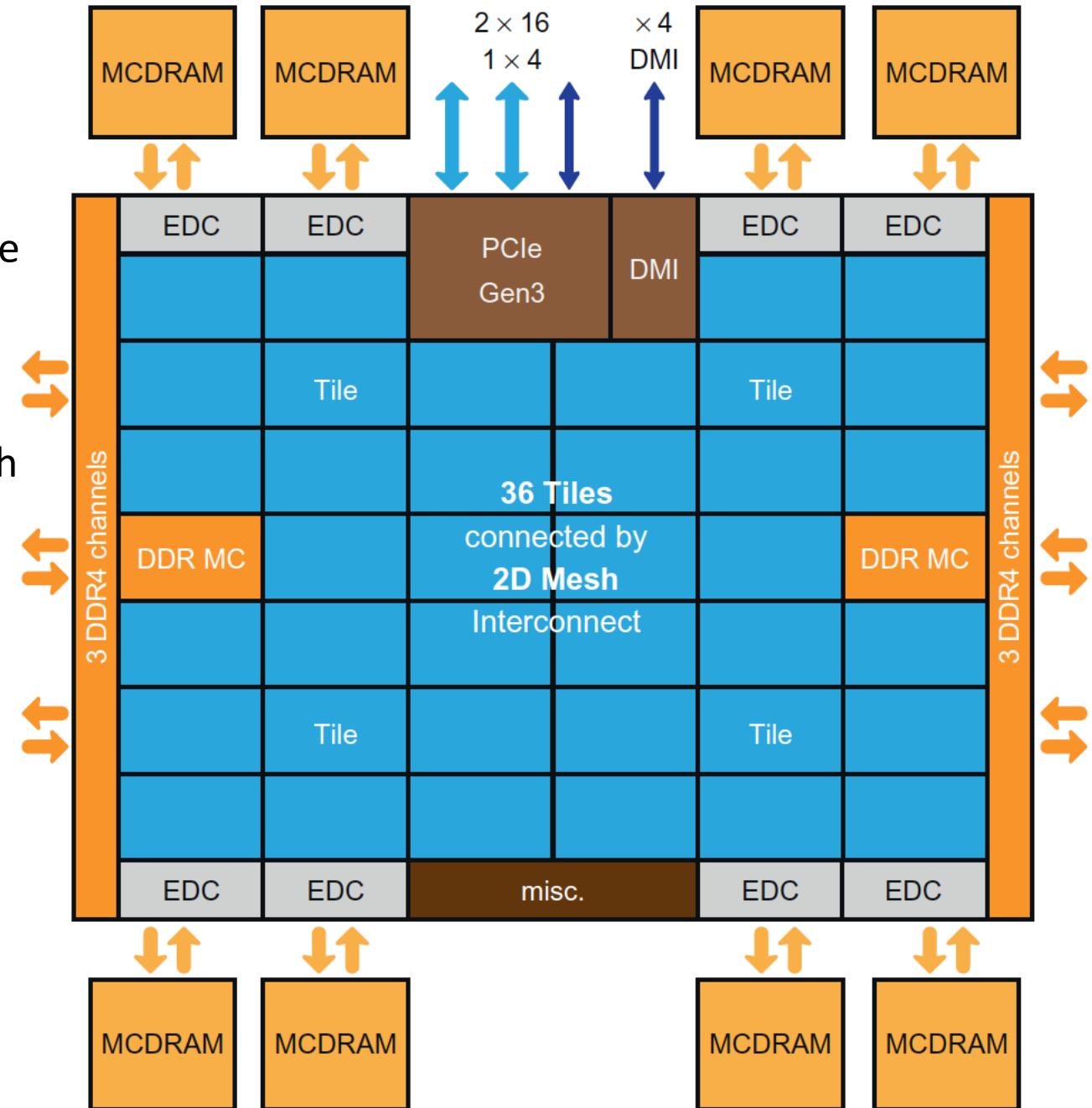
KNL Architecture



- Cores derived from the Intel Atom Silvermont
- Tile: 2 cores, 4 threads per core, 2 Vector processing units per core and 1 MB L2 cache shared among the cores.
- Up to 38 tiles, 36 active: 72 cores, 144 VPUs (32 tiles in our case)
- ISA: AVX-512 instructions, 8 DP (16 Flops) or 16 SP (32 Flops)
- AVX-512ER means exp and reciprocal.
 - A transcendental function is an analytic function that does not satisfy a polynomial equation. E.g. log, exp, trigonometric functions.
 - An intrinsic function is a function which the compiler implements directly when possible, rather than linking to a library-provided implementation

KNL Architecture

- MCDRAM: high bandwidth memory, 8 devices x 2 GB, with separate bus for R and W operations. The bandwidth from all the 8 MCDRAM is 450 GBps.
 - MCDRAM can be used as RAM, as L3 cache for the RAM or in a hybrid way
 - DRAM: 2 DDR4 memory controller, 3 channel each (1 DIMM up to 64 GB per channel). The aggregated bandwidth is 90 GBps
 - On-die interconnect based on a ring architecture (4 rings) capable of 700 GBps of aggregated bandwidth
 - YX routing: transaction always travels vertically first until it hits the target row, makes a turn, and then travels horizontally until its destination. A single hop on the mesh takes 2 clocks in the X-direction and only 1 in the Y-direction.

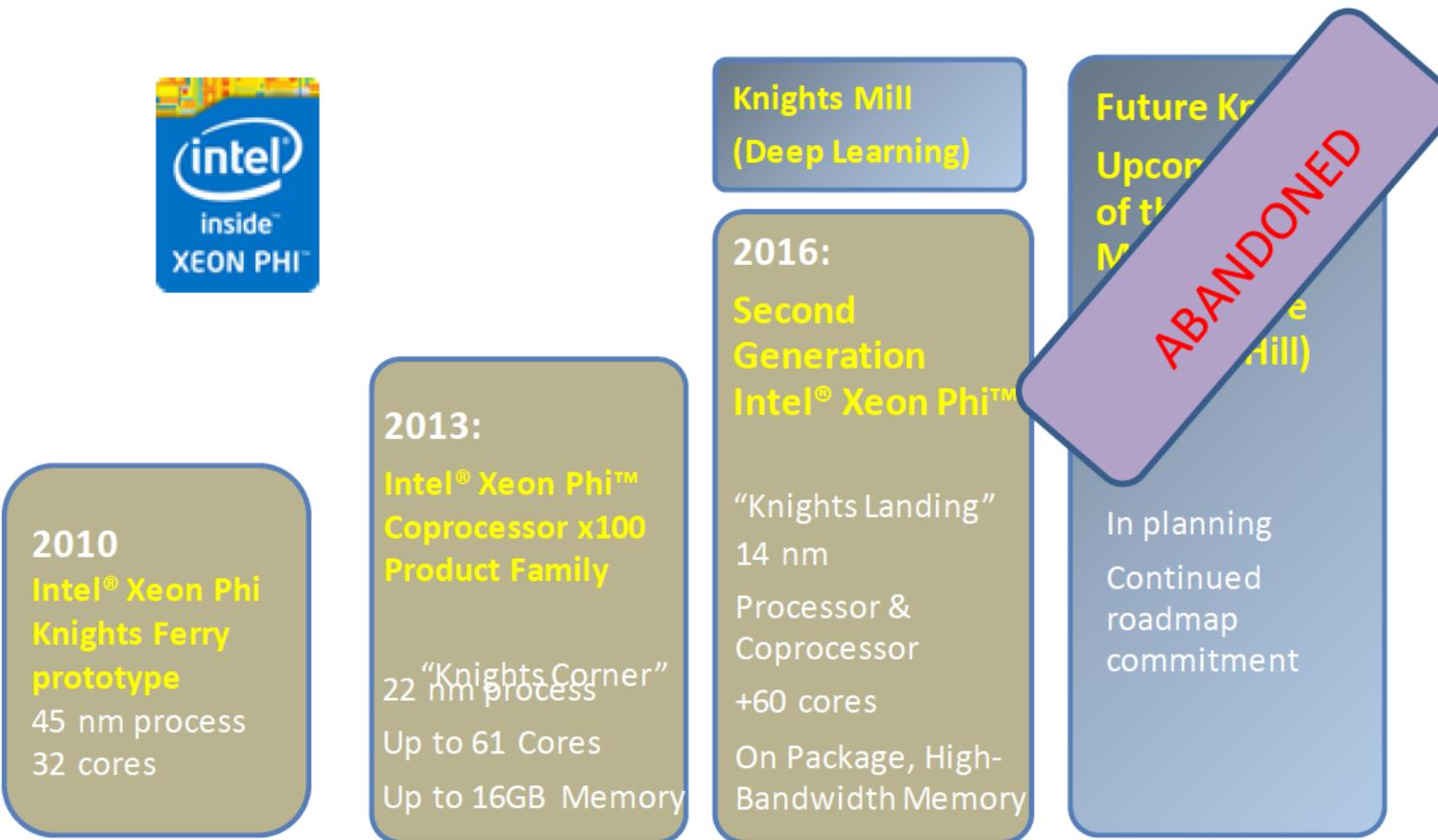


NUMANode P#0 (96GB)

Package P#0



Intel Xeon PHI Roadmap



*Per Intel's announced products or planning process for future products

In November 2017 Intel announced that the Xeon Phi line would be abandoned due to “market and customer needs” (Intel). Probably means that KNL had no market outside HPC.

CPU Flops

$$FLOPS_{\text{system}} = \frac{\text{instructions}}{\text{cycle}} \times \frac{\text{operations}}{\text{instruction}} \times \frac{\text{FLOPs}}{\text{operation}} \\ \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{cores}}{\text{node}} \times \frac{\text{nodes}}{\text{system}}$$

https://en.wikipedia.org/wiki/Xeon_Phi

KNL 7210:

64 core * 32 DP Flops * 1.3 GHz= 2.6 TFLOPS DP

8 DP * 2 EU * 2 FLOP (FMA = * and +)

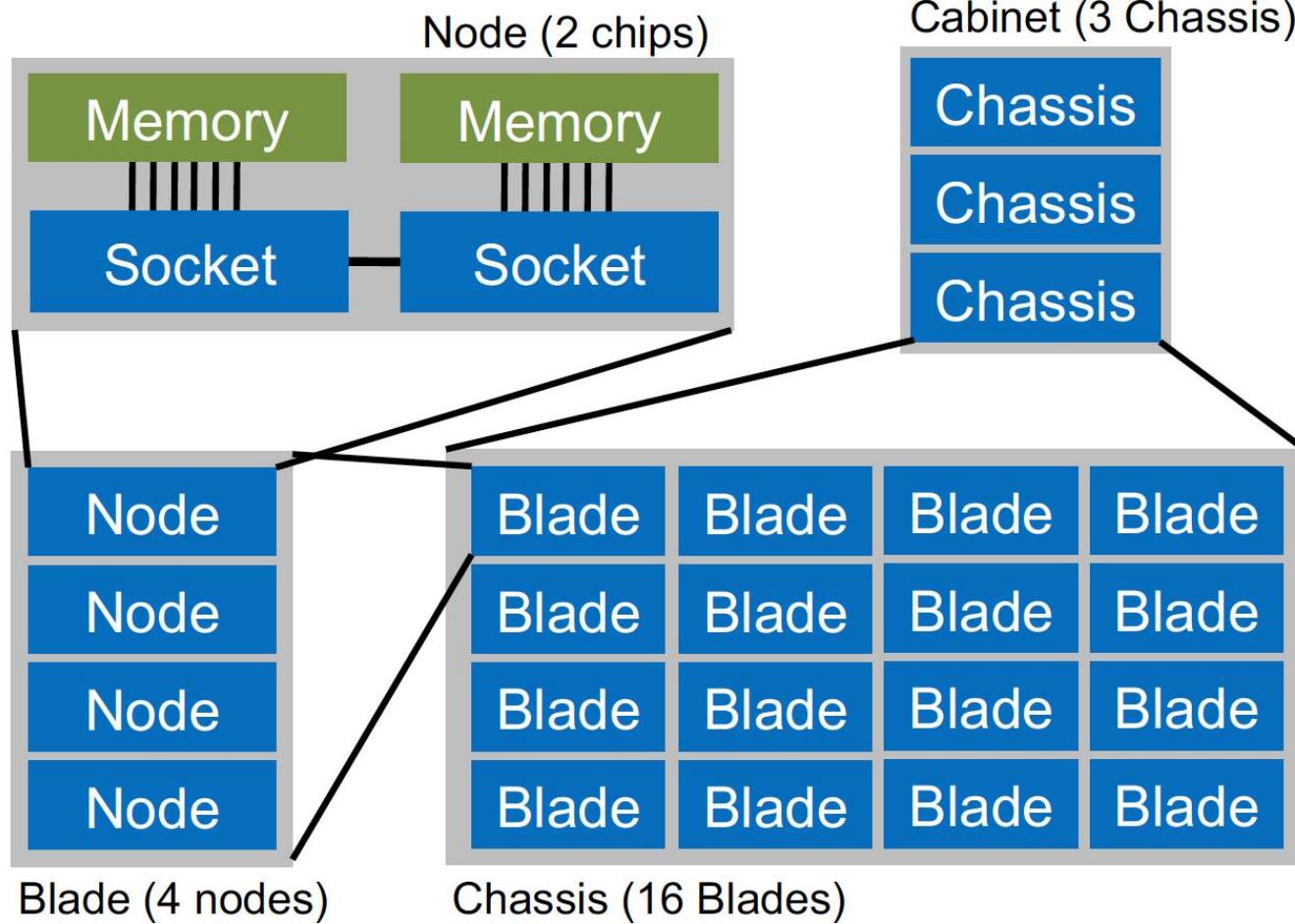
SKU	CORES	BASE (GHz)	SINGLE CORE TURBO (GHz)	ALL CORE TURBO (GHz)	CACHE (MB)	TDP (Watts)	3.8 TFLOPS
8380	40	2.3	3.4	3.0	60	270	

Microarchitecture	FLOPs			ISA		
Intel Microarchitectures						
Core Penryn Nehalem	EUs	1 × 128-bit Multiplication + 1 × 128-bit Addition				
	DP	4 FLOPs/cycle	2 FLOPs + 2 FLOPs			
	SP	8 FLOPs/cycle	4 FLOPs + 4 FLOPs			
Sandy Bridge Ivy Bridge	EUs	1 × 256-bit Multiplication + 1 × 256-bit Addition				
	DP	8 FLOPs/cycle	4 FLOPs + 4 FLOPs			
	SP	16 FLOPs/cycle	8 FLOPs + 8 FLOPs			
Haswell Broadwell Skylake Kaby Lake Coffee Lake Whiskey Lake Amber Lake	EUs	2 × 256-bit FMA				
	DP	16 FLOPs/cycle	2 × 8 FLOPs			
	SP	32 FLOPs/cycle	2 × 16 FLOPs			
Skylake (server)	EUs	2 × 512-bit FMA (varies by SKU)				
	DP	32 FLOPs/cycle	2 × 16 FLOPs			
	SP	64 FLOPs/cycle	2 × 32 FLOPs			
Intel MIC Microarchitectures						
Knights Landing	EUs	2 × 512-bit FMA (varies by SKU)				
	DP	32 FLOPs/cycle	2 × 16 FLOPs			
	SP	64 FLOPs/cycle	2 × 32 FLOPs			

MIMD (or SPMD)

- **Multiple Instruction:** every processor may be executing a different instruction stream
- **Multiple Data:** every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic. Currently, the most common type of parallel computer/supercomputer /systems
- In practice **SPMD:** Single Program Multiple Data:
 - all processors execute the same code, not the same instruction
 - different control flow (i.e. branches, loop)
 - possible different amount of data, i.e. load unbalance

MIMD



Supercomputers: the most powerful computers available in a given period of time.
Powerful is meant in terms of execution speed, memory capacity and accuracy of the machine.

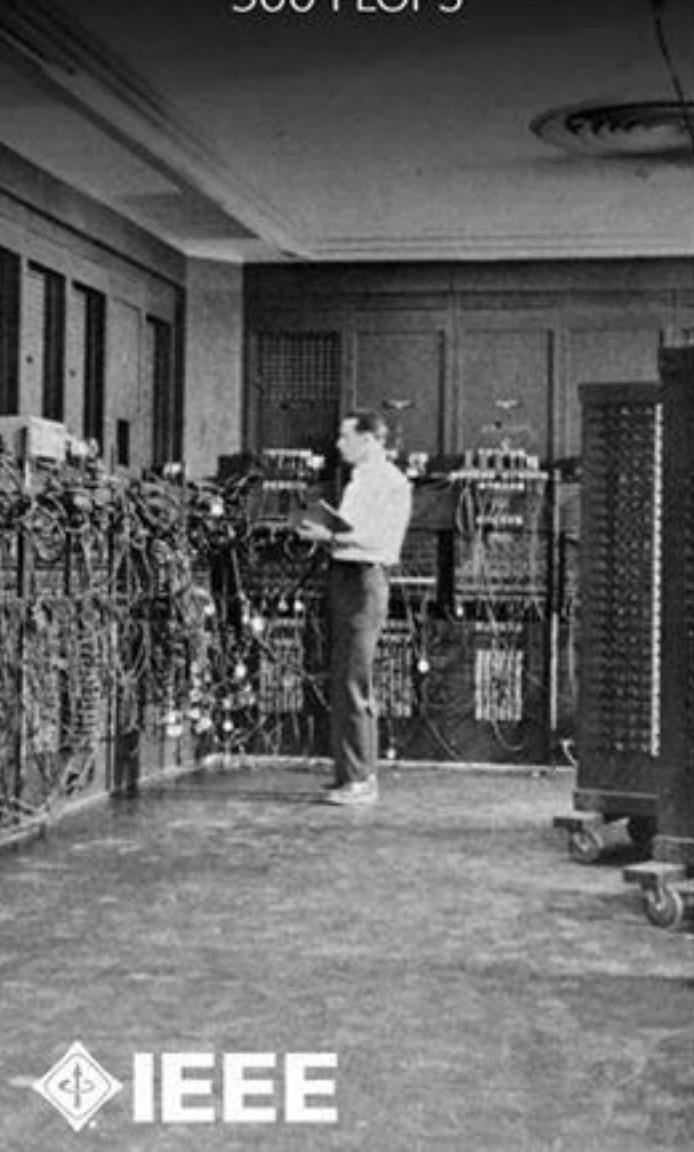


SuperMUC © LRZ

A HPC System consists of many Cabinets!

ENIAC (1947)

\$21,900 yearly electric bill
500 FLOPS



iPad (2013)

\$1.36 yearly electric bill
76,800,000,000 FLOPS



Performance:

↑ 153,599,999%

Energy Usage:

↓ 99.994%



IEEE

SANDIA ASCI RED

Date:

1996

Peak performance:

1.8Teraflops

Floor space:

150m

Power consumption:

800.000 Watt



Sony PLAYSTATION 3

Date:

2006

Peak performance:

>1.8Teraflops

Floor space:

0.08m

Power consumption:

<200 Watt

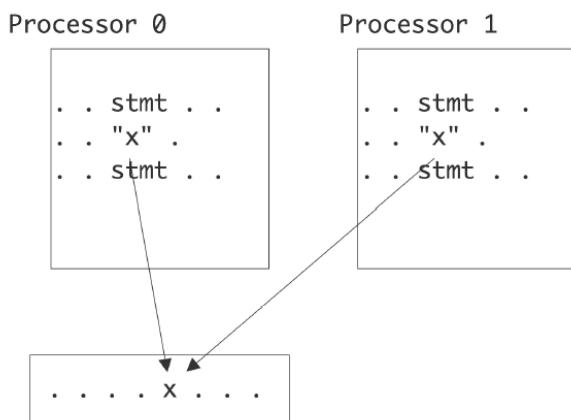


<https://phys.org/news/2010-12-air-playstation-3s-supercomputer.html>

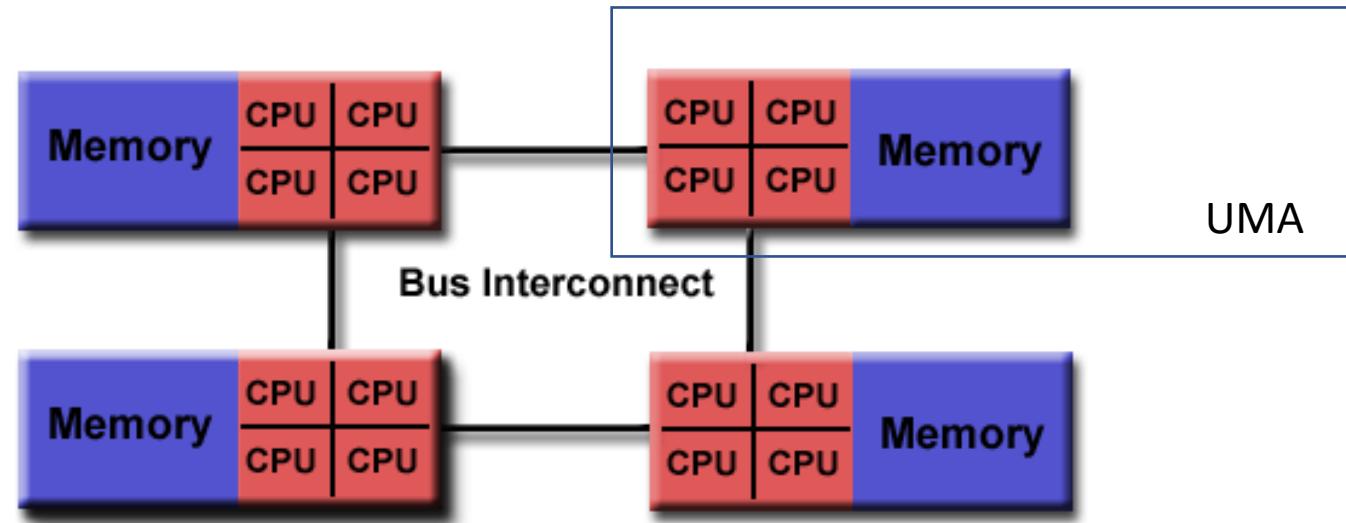
Yesterday's supercomputers could become today's' appliances

Memory Access – Shared Memory

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Historically, shared memory machines have been classified as ***Uniform Memory Access (UMA)*** and ***Non-UMA (NUMA)***

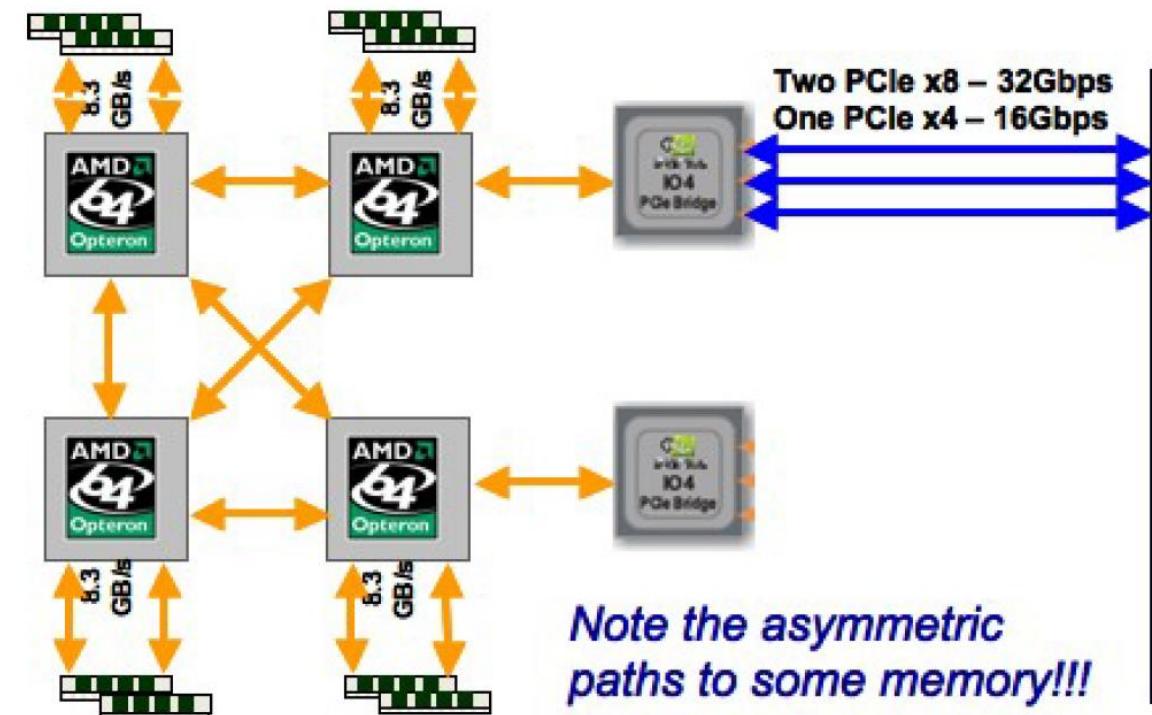
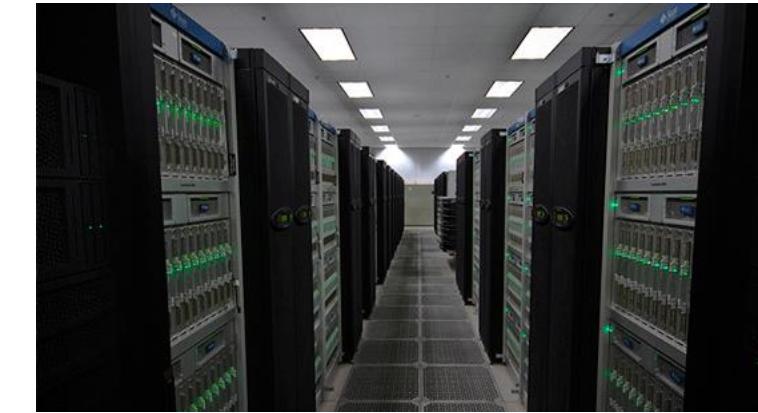
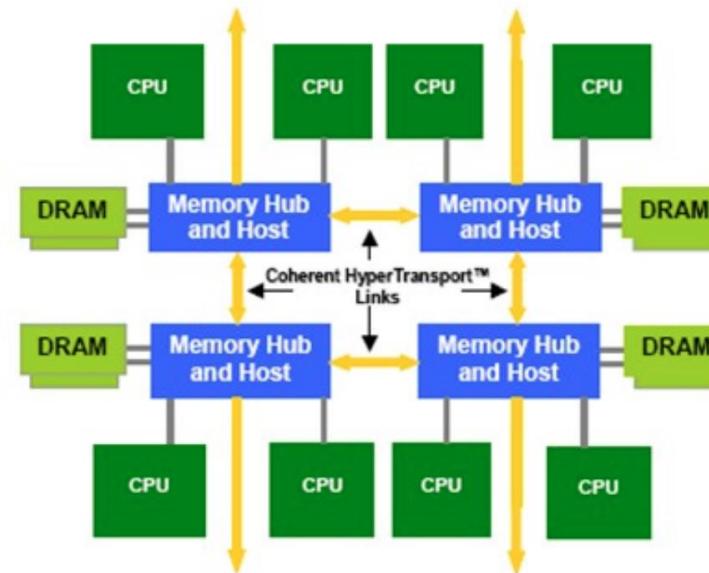


NUMA



Example

- The four-socket motherboard of the Ranger supercomputer.
- Each chip has its own memory (8Gb) but the motherboard acts as if the processors have access to a shared pool of 32Gb.
- Each processor has three connections that could be used to access other memory, but the rightmost two chips use one connection to connect to the network



Numactl --hardware

```
[dadagostino@hpcocapie01 ~]$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66
67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 1
26 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151
152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 1
77 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202
203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 2
28 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253
254 255
node 0 size: 98180 MB
node 0 free: 87172 MB
node distances:
node 0
 0: 10
```

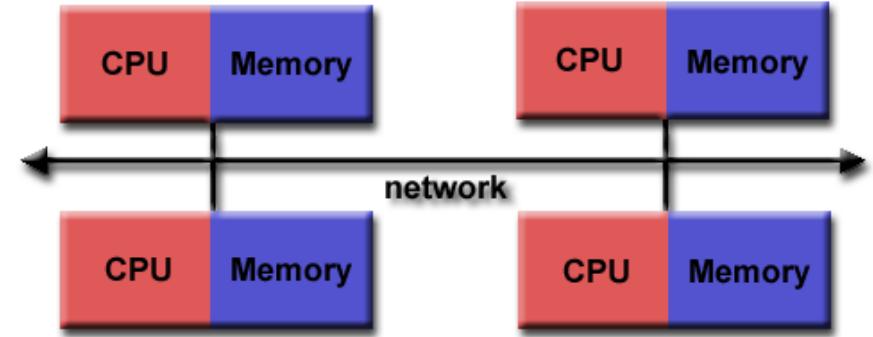
SMP - UMA

numactl is a utility which can be used to control NUMA policy for processes or shared memory. NUMA (stands for Non-Uniform Memory Access) is a memory architecture in which a given CPU core has variable access speeds to different regions of memory. Typically, each core will have a region of memory attached to it directly which it can access quickly (local memory), while access to the rest of the memory is slower (non-local memory).

```
[dago@cluster2 ~]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 16374 MB
node 0 free: 7956 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 16384 MB
node 1 free: 3660 MB
node distances:
node 0 1
 0: 10 21
 1: 21 10
```

NUMA

Memory Access – Distributed Memory



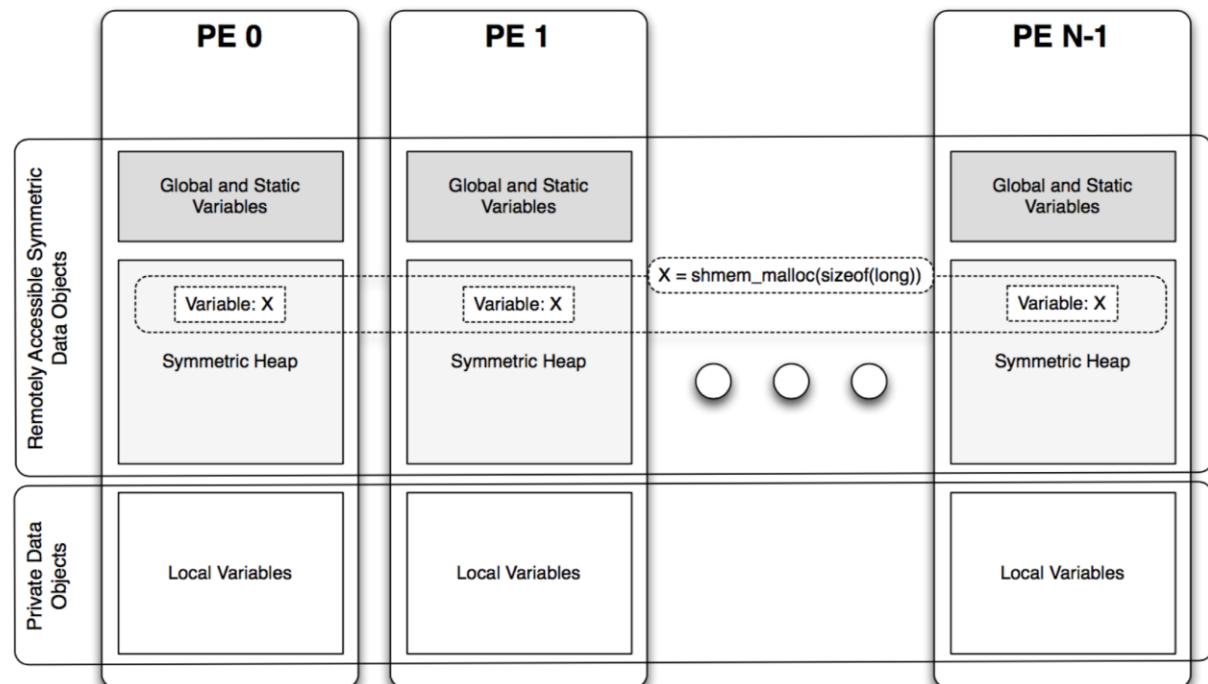
- Processors have their own local memory.
- Memory addresses in one processor do not map to another processor, so there is NO concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.
- Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

But

Via software is possible to offer a NUMA shared memory.

Partitioned Global Adress Space (PGAS) programming models offer HPC programmers an abstracted shared address space, which simplifies programming, while exposing data/thread locality to enhance performance. This can facilitate the development of productive programming languages that can reduce the time to solution, i.e. both development time and execution time.

[<https://developer.nvidia.com/nvshmem>](http://www.openshmem.org/site>Welcome</p></div><div data-bbox=)



Languages

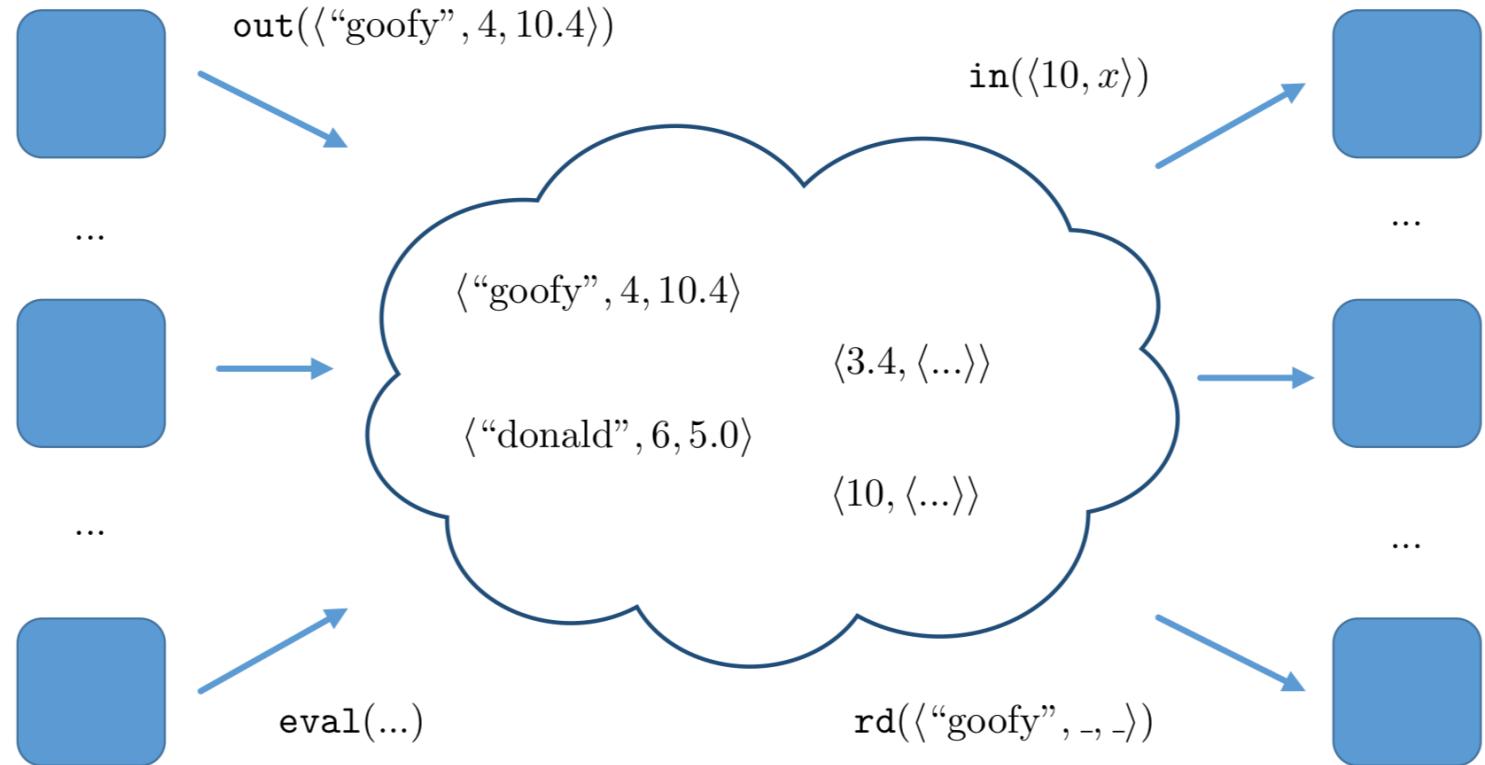
- Unified Parallel C (UPC)
- Co-Array Fortran (CAF)
- X10
- Chapel
- STAPL
- Titanium

Libraries

- UPC++
- **OpenSHMEM**
- Global Arrays
- **DASH**
- ...

Or Tuple spaces

- A **tuple space** is a distributed repository, where processes can add, withdraw or read **tuples** by means of atomic operations.
- **Tuples** may contain different values, and processes can inspect their content via pattern matching.
- The lack of a free reference **implementation** for this paradigm has prevented its widespread.

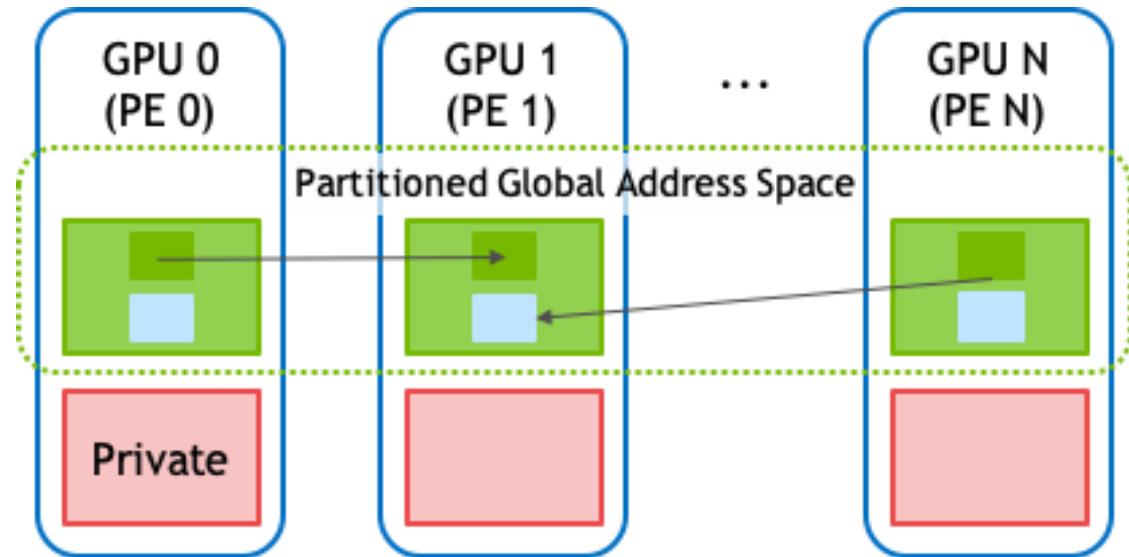
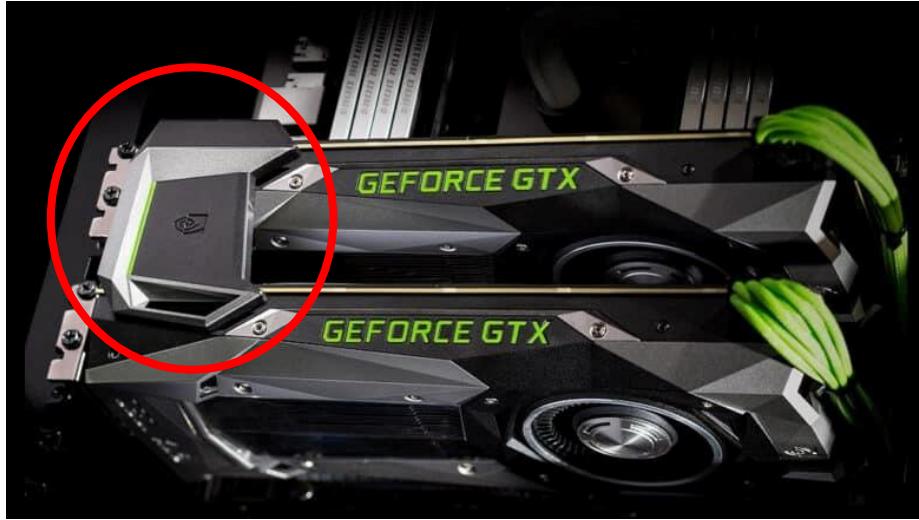


https://en.wikipedia.org/wiki/Tuple_space

<http://wiki.c2.com/?TupleSpace>

<https://arxiv.org/pdf/1612.02979.pdf>

Nvshmem



- NVSHMEM is based on OpenSHMEM, which provides a global address space for data that spans the memory of multiple GPUs.
- An NVSHMEM job consists of several operating system processes, that are referred to as processing elements (PEs).
- An object allocated with `nvshmem_malloc()` is called a symmetric data object. Every symmetric data object has a corresponding data object with the same name, type, and size on all PEs.

To summarize

Shared memory

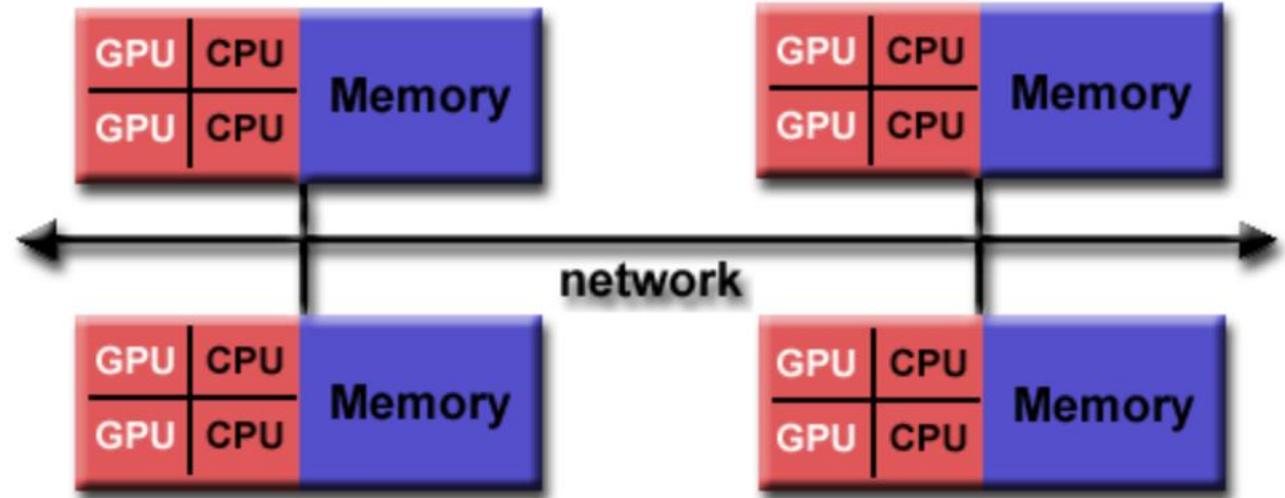
- Advantages:
 - Easier to program
 - Useful for applications with irregular data access patterns (e.g., graph algorithms)
- Disadvantages:
 - The programmer must take care of race conditions
 - Limited memory bandwidth

Distributed memory

- Advantages:
 - Highly scalable, provide very high computational power by adding more nodes
 - Useful for applications with strong locality of reference, with high computation / communication ratio
- Disadvantages:
 - Latency of interconnect network
 - Difficult to program

Actual systems

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component can be a shared memory machine and/or graphics processing units (GPU), the node.
- The distributed memory component is the networking of multiple nodes. Therefore, **network communications** are required to move data from one machine to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Increased scalability is an important advantage
- Increased programmer complexity is an important disadvantage



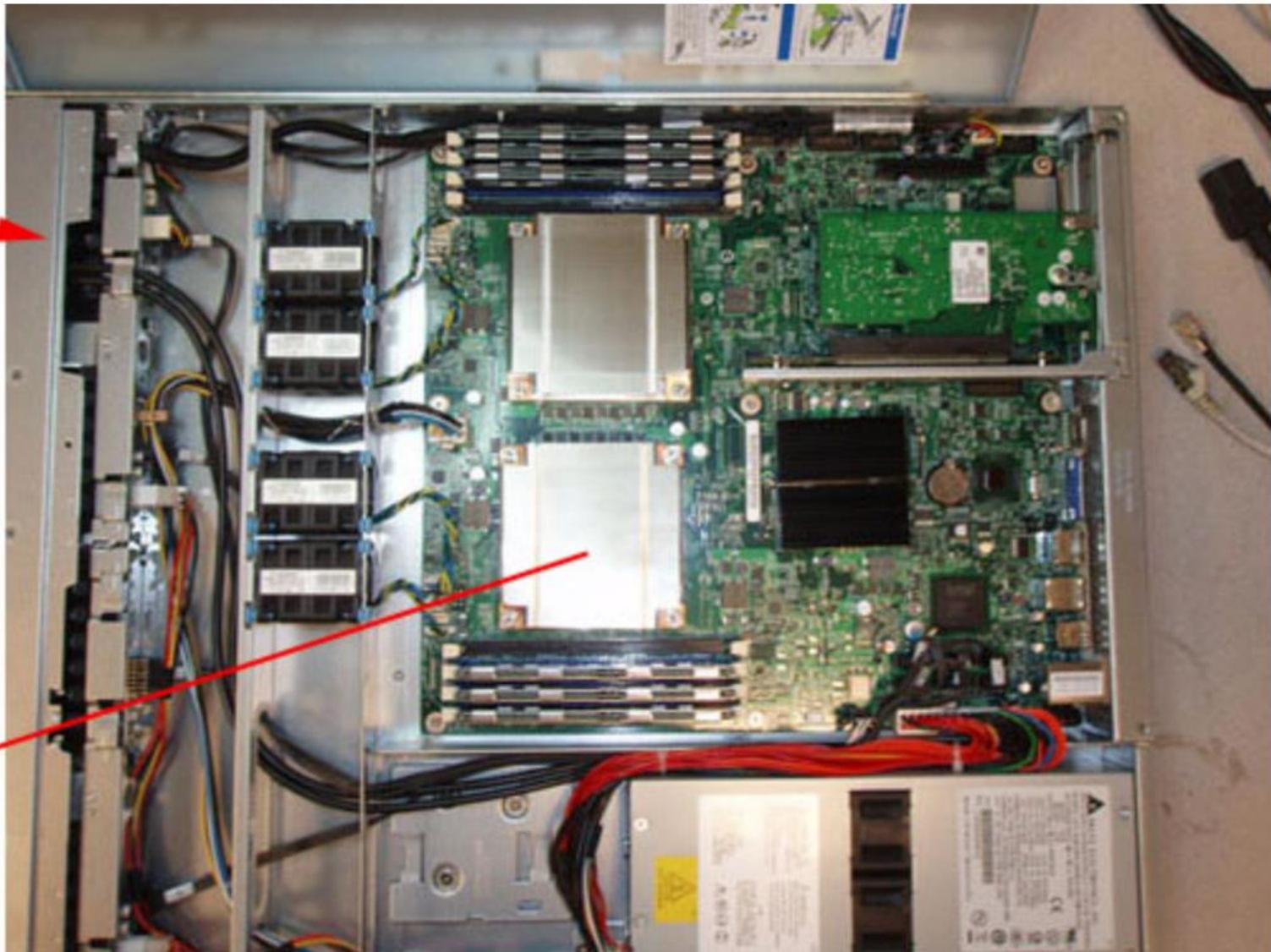
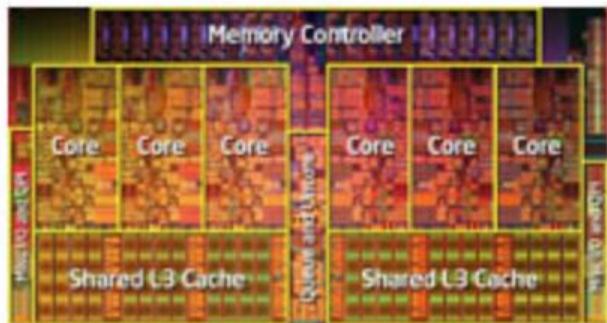


Supercomputer - each blue light is a node

Node - standalone

Von Neumann computer

CPU / Processor / Socket - each has multiple cores / processors.

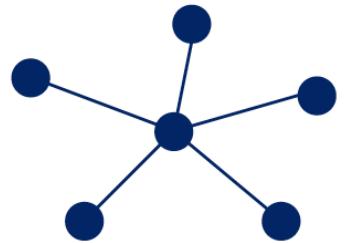


Interconnection

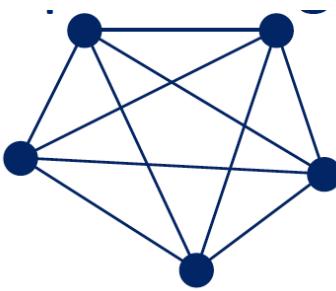
- If a number of processors are working together on a single task, most likely they need to communicate data.
- Here a brief overview on possible interconnection schema
- (Original) Ethernet is a connection scheme where all machines on a network are on a single cable. Simple but limited capacity solution, channel sharing problems...
- In a fully connected configuration, each processor has one wire for the communications with each other processor. This scheme is perfect in the sense that messages can be sent in the minimum amount of time, and two messages will never interfere with each other. Complex to be implemented (and very expensive)..

<http://www.prace-ri.eu/best-practice-guide-modern-interconnects>

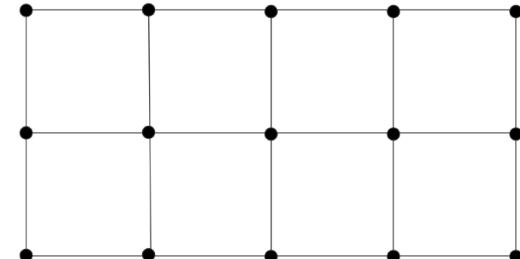
Interconnections



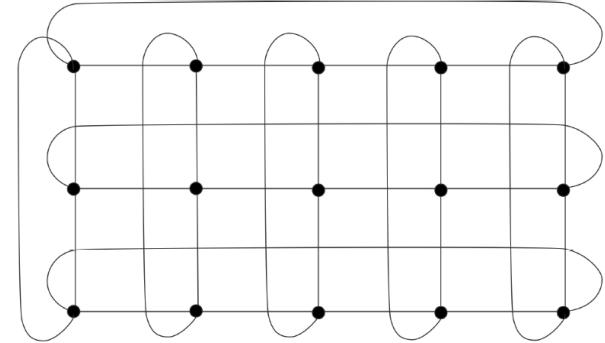
Star



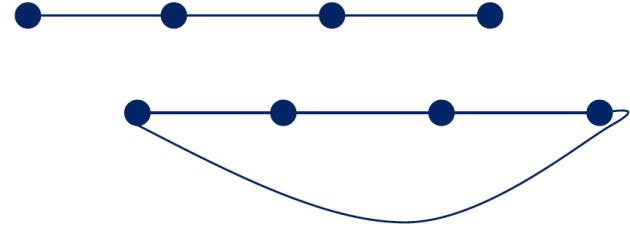
Complete connection



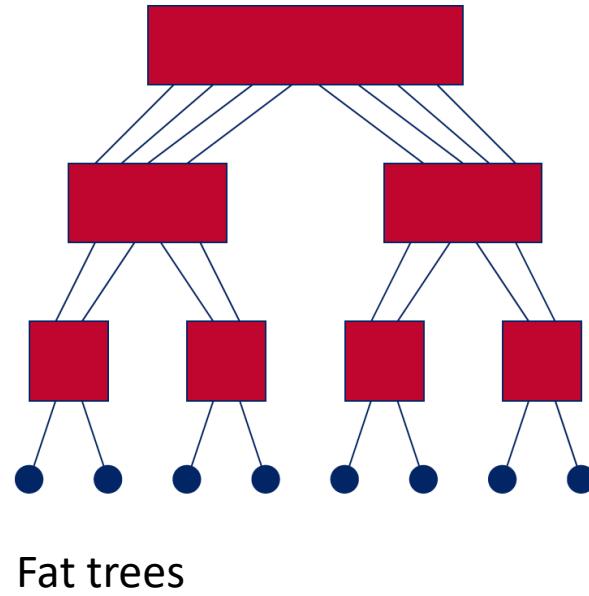
2D Grid



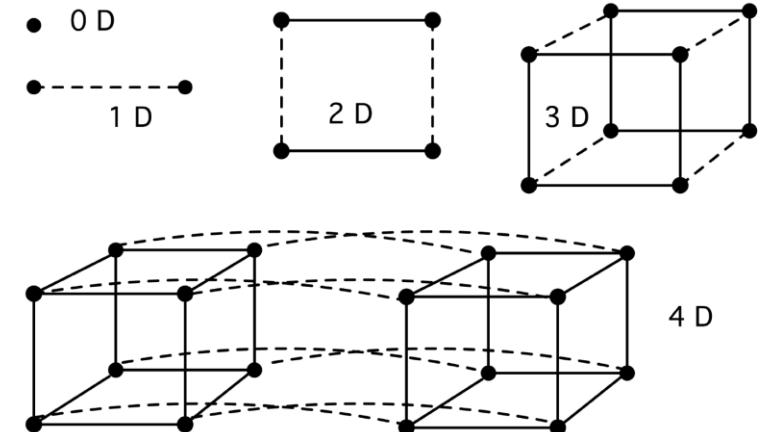
Torus



Ring



Fat trees



Hypercubes

Graph theory concepts

Degree

- How many links to other processors does each node have?
- More is better, but also expensive and hard to engineer

Diameter

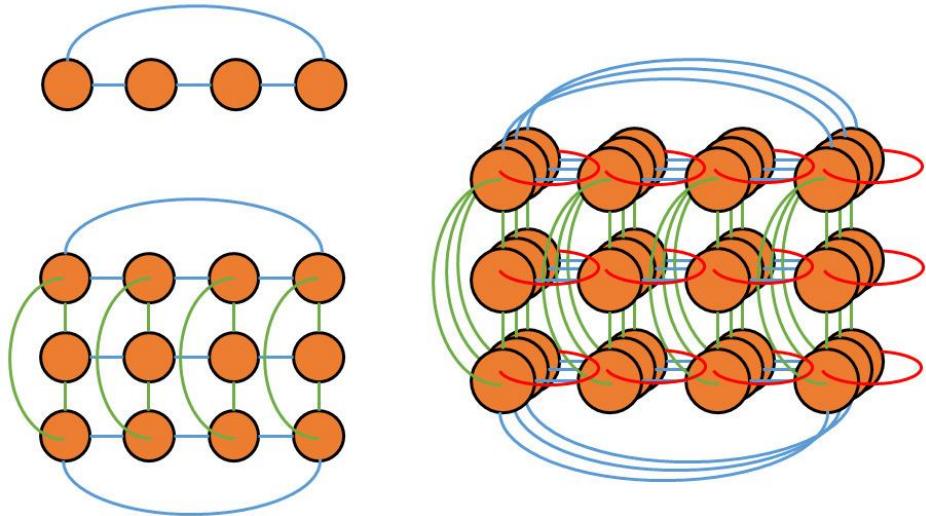
- maximum distance between any two processors in the network:
completely connected network is 1, for star network is 2, for ring is $p/2$ (for p even processors)

Connectivity

- measure # arcs that must be removed to break the connection: 1 for linear array/star/torus (but), 2 for ring/mesh, 4 for torus

Torus

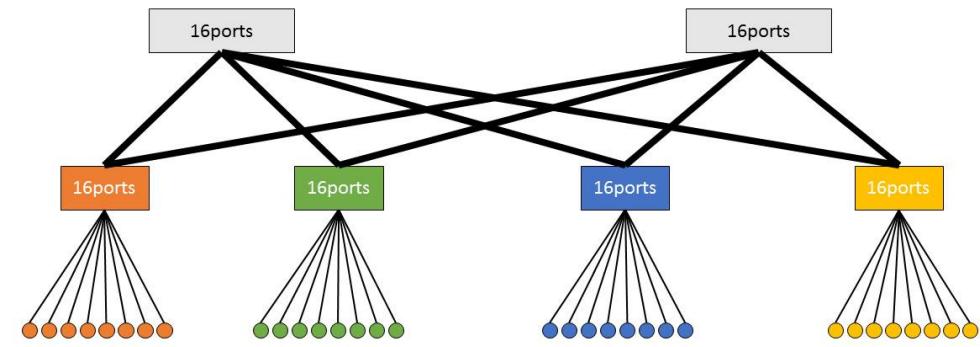
- A torus allows for very fast neighbour-to-neighbour communication. In addition, there can be different paths for moving data between two different nodes.
- The latency is normally quite low and can be distinguished by the maximum number of hops needed.
- As there is no direct central element, the number of individual wires can be very high.
- In an N-dimensional torus, each node has $2N$ individual connections.



Hypercubes and meshes

- It is a fairly natural idea to have 2D or 3D networks, since the world around us is three-dimensional, and computers are often used to model real-life phenomena.
- Hypercubes: smaller diameter mesh, useful for broadcasts
- The main difference between a pure hypercube and a torus are the number of connections of the border nodes. Within the torus setup each network node has the same number of neighbour nodes, while in the hypercube only the inner nodes have the same number of neighbours and the border nodes have fewer neighbours. This allows for fewer required network hops in a torus network if border nodes need to communicate.

Fat Trees



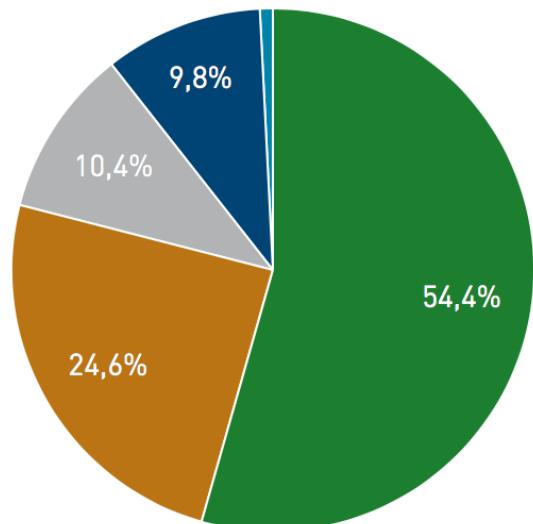
- Are used to provide a low latency network between all involved nodes.
- All computing nodes are located on the leaves of the tree structure and switches are located on the inner nodes.
- The idea of the fat tree is, instead of using the same wire “thickness” for all connections, to have “thicker” wires closer to the top of the tree and provide a higher bandwidth over the corresponding switches.
- In an ideal case a fat-tree is a tree network where each level has the same total bandwidth, so that congestion problem does not occur.

Infiniband and High Performance Ethernet

<http://prace.it4i.cz/sites/prace.it4i.cz/files/files/iohed-01-2018-slides.pdf>

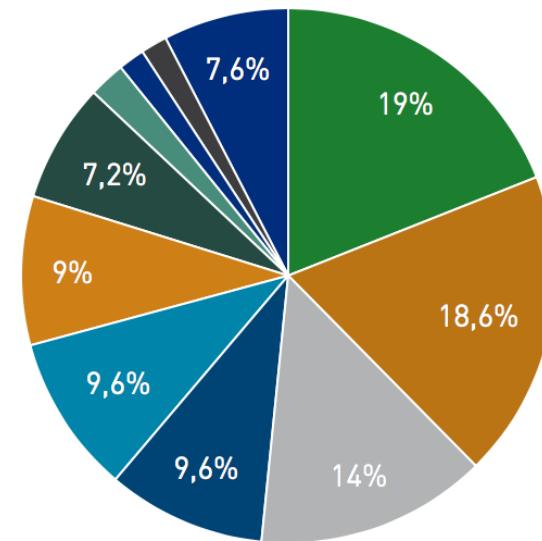
<https://en.wikipedia.org/wiki/InfiniBand>

Interconnect Family System Share



- Gigabit Ethernet
- Infiniband
- Custom Interconnect
- Omnipath
- Proprietary Network

Interconnect System Share



- 10G Ethernet
- 40G Ethernet
- 25G Ethernet
- Infiniband FDR
- Intel Omni-Path
- Infiniband EDR
- Aries interconnect
- 100G Ethernet
- Mellanox InfiniBand EDR
- Custom Interconnect
- Others



Network Speed Acceleration with IB and HSE

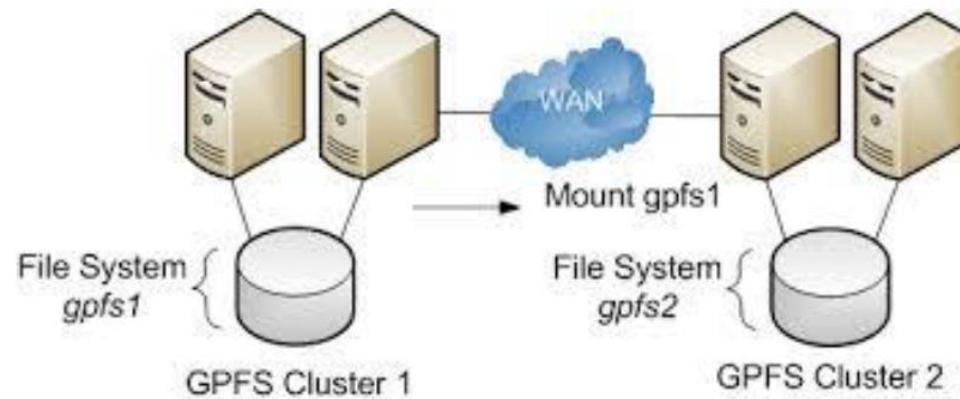
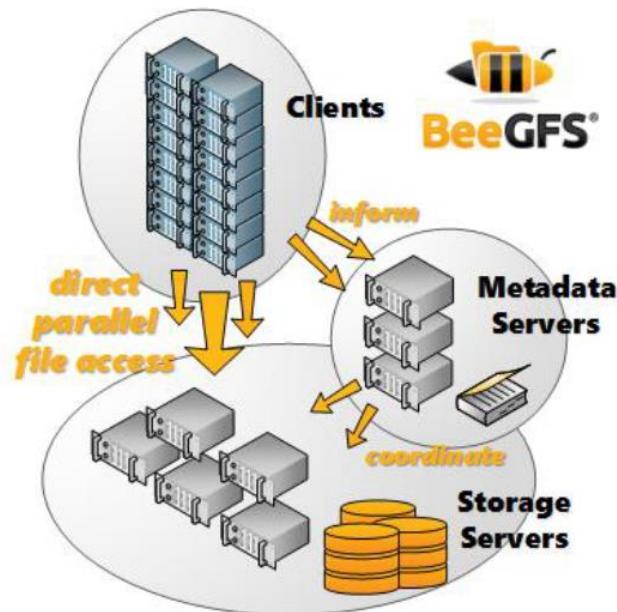
Ethernet (1979 -)	10 Mbit/sec
Fast Ethernet (1993 -)	100 Mbit/sec
Gigabit Ethernet (1995 -)	1000 Mbit /sec
ATM (1995 -)	155/622/1024 Mbit/sec
Myrinet (1993 -)	1 Gbit/sec
Fibre Channel (1994 -)	1 Gbit/sec
InfiniBand (2001 -)	2 Gbit/sec (1X SDR)
10-Gigabit Ethernet (2001 -)	10 Gbit/sec
InfiniBand (2003 -)	8 Gbit/sec (4X SDR)
InfiniBand (2005 -)	16 Gbit/sec (4X DDR)
	24 Gbit/sec (12X SDR)
InfiniBand (2007 -)	32 Gbit/sec (4X QDR)
40-Gigabit Ethernet (2010 -)	40 Gbit/sec
InfiniBand (2011 -)	54.6 Gbit/sec (4X FDR)
InfiniBand (2012 -)	2 x 54.6 Gbit/sec (4X Dual-FDR)
25-/50-Gigabit Ethernet (2014 -)	25/50 Gbit/sec
100-Gigabit Ethernet (2015 -)	100 Gbit/sec
Omni-Path (2015 -)	100 Gbit/sec
InfiniBand (2015 -)	100 Gbit/sec (4X EDR)
InfiniBand (2016 -)	200 Gbit/sec (4X HDR)

100 times in the last 15 years

Parallel filesystems

The filesystem manages how files are stored on disks and how they can be retrieved or written.

In a parallel architecture, with many simultaneous accesses to the disks, important to use a *parallel filesystem* technology such as GPFS, LUSTRE, BeeGFS etc.



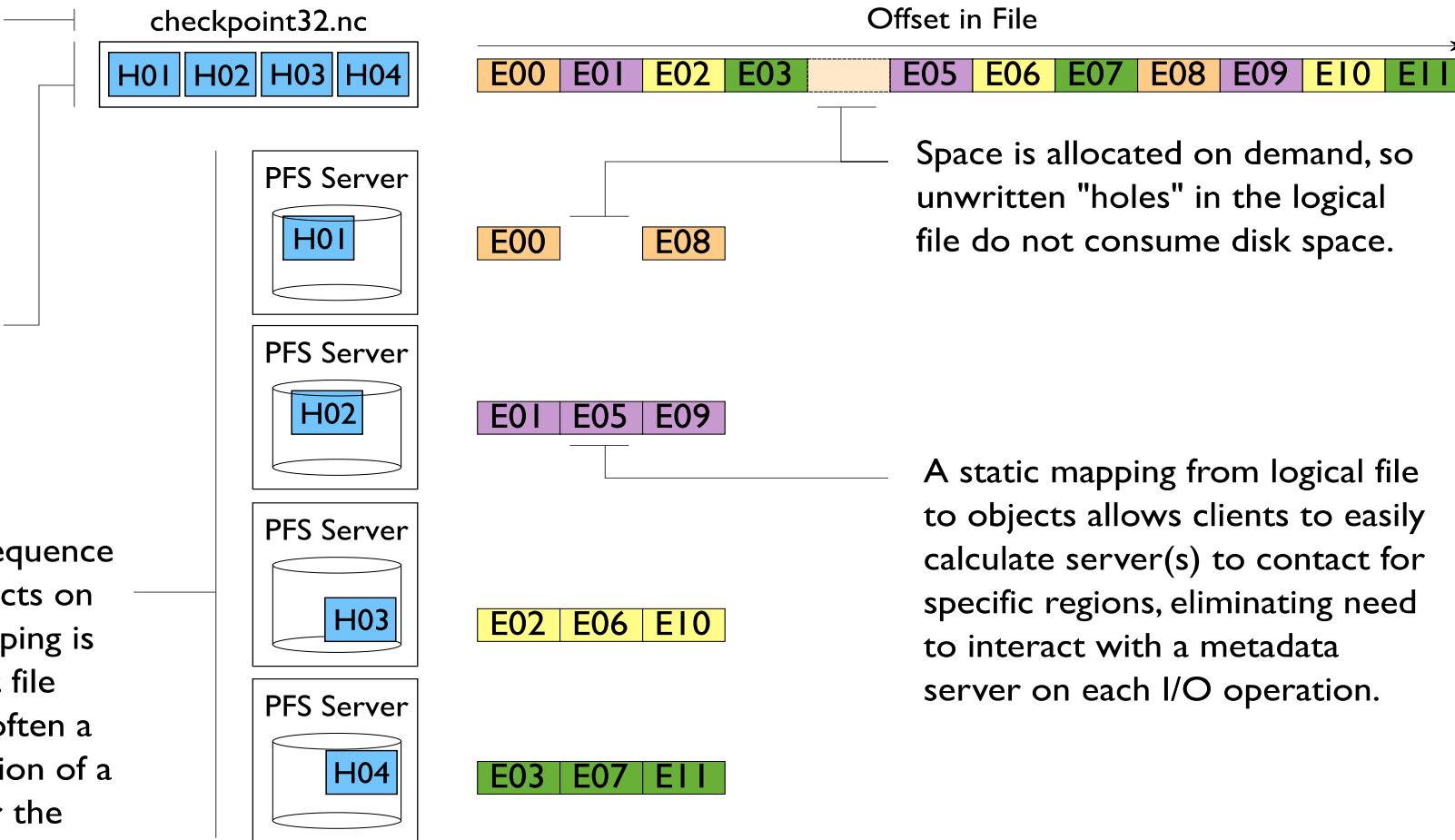
Data Distribution in Parallel File Systems

Distribution across multiple servers allows concurrent access.

Logically a file is an extendable sequence of bytes that can be referenced by offset into the sequence.

Metadata associated with the file specifies a mapping of this sequence of bytes into a set of objects on PFS servers.

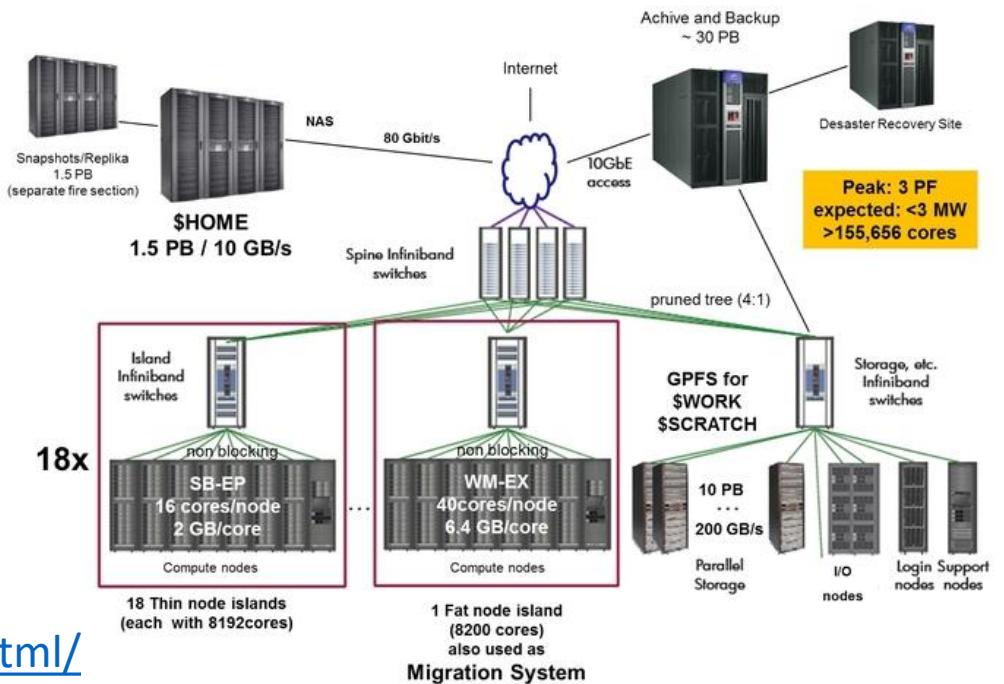
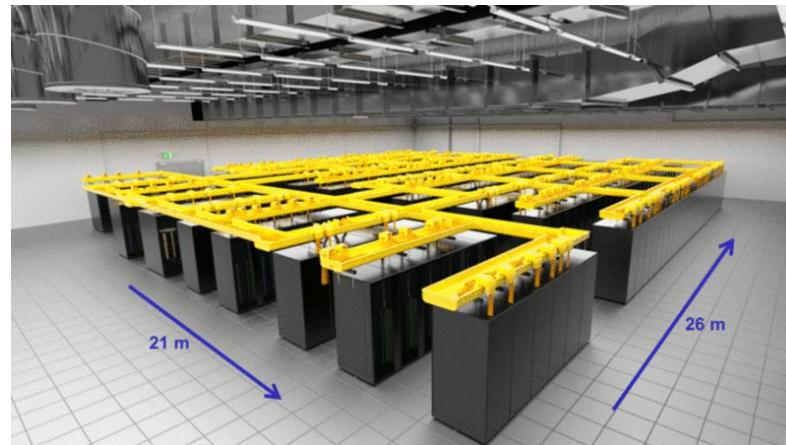
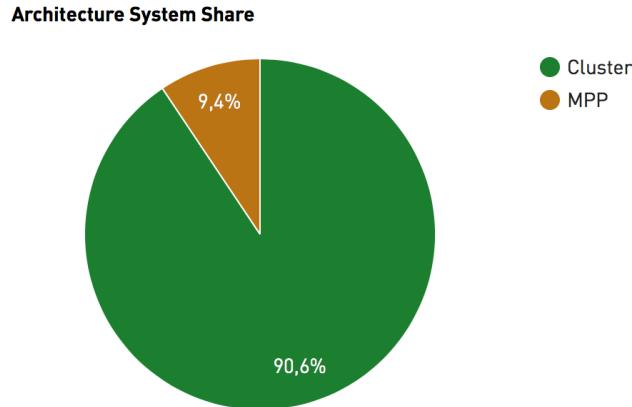
Extents in the byte sequence are mapped into objects on PFS servers. This mapping is usually determined at file creation time and is often a round-robin distribution of a fixed extent size over the allocated objects.



See also RAID <https://en.wikipedia.org/wiki/RAID> (here a schema, but in Italian <https://it.wikipedia.org/wiki/RAID>)

Clusters

- Parallel computer systems comprising an integrated collection of independent nodes, each of which is a system in its own right, capable of independent operation and derived from products developed and marketed for other stand-alone purposes
- E.g. SuperMUC

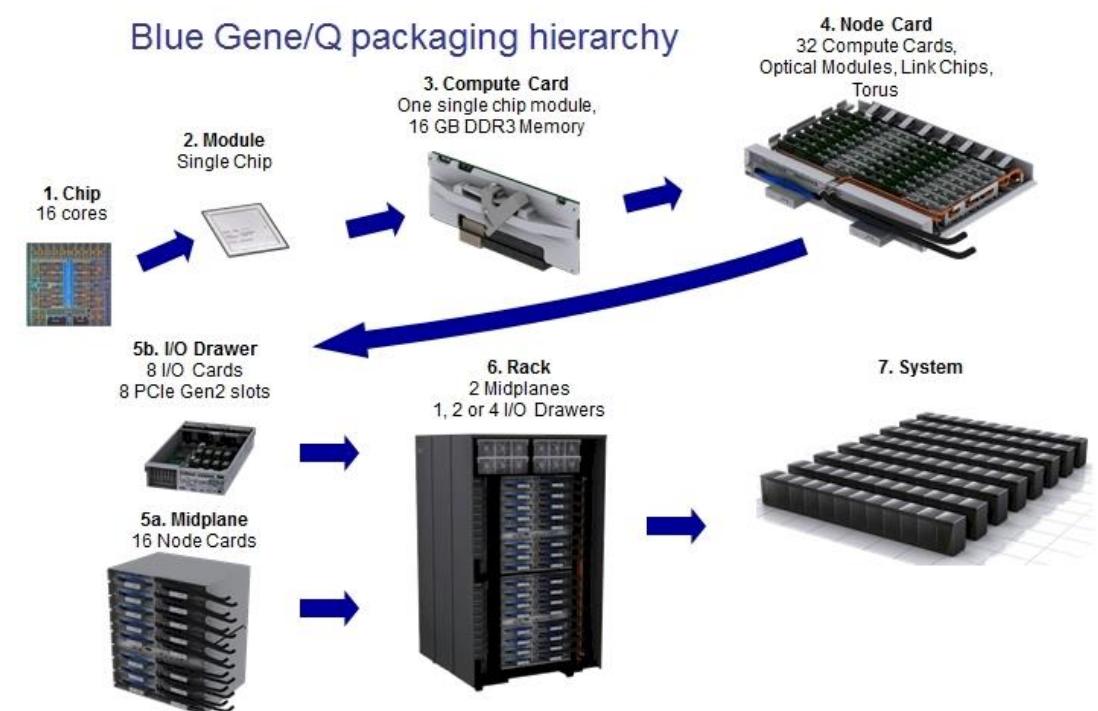


<https://doi.org/10.1109/MCSE.2005.34>

<http://www.prace-ri.eu/best-practice-guide-supermuc-html/>

Massively Parallel Processors

- These are more tightly-integrated systems: individual nodes cannot run on their own and they are connected by a custom network (like a multidimensional torus).
- But, similarly to a cluster, there is no single, shared memory spanning all the nodes
- The five-dimensional torus (10 links) is the Blue Gene/Q internal network used for all MPI communications and also for all the I/O between the compute nodes and the I/O nodes.



Linux Cluster

- An LC production cluster has four types of nodes, based upon function, which can differ in configuration details:
 - Login, Interactive/debug (with queues) , Batch, I/O and service nodes (not for users)

<https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide#UG3.1:MARCONIUserGuide-Productionenvironment>

- **Login nodes:**

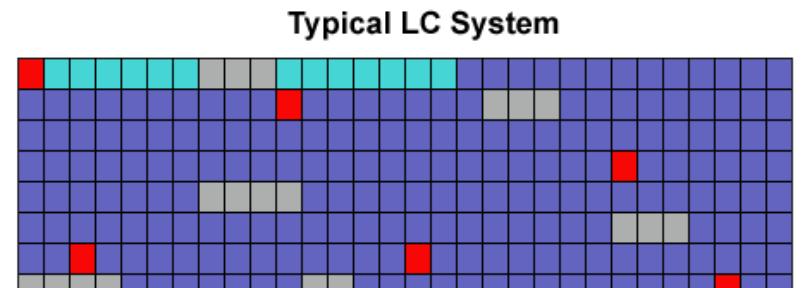
- Every system has a designated number of login nodes - depends upon the size of the system. Some examples:
 - agate = 2
 - sierra = 5
 - quartz = 14
 - zin = 20
- Login nodes are shared by multiple users
- Primarily used for interactive work such as editing files, submitting batch jobs, compiling, running GUIs, etc.
- Interactive use exclusively - login only nodes do not permit any batch jobs.
- DO NOT run production jobs on login nodes! Remember, you are sharing login nodes with other users.

- **Interactive/debug (pdebug) nodes:**

- Most LC systems have nodes that are designated for interactive work.
- Meant for testing, prototyping, debugging, and small, short jobs
- Cannot be logged into unless you already have a job running on them
- Nodes run one job at a time - not shared like login nodes
- Can also be used through the batch system

- **Batch (pbatch) nodes:**

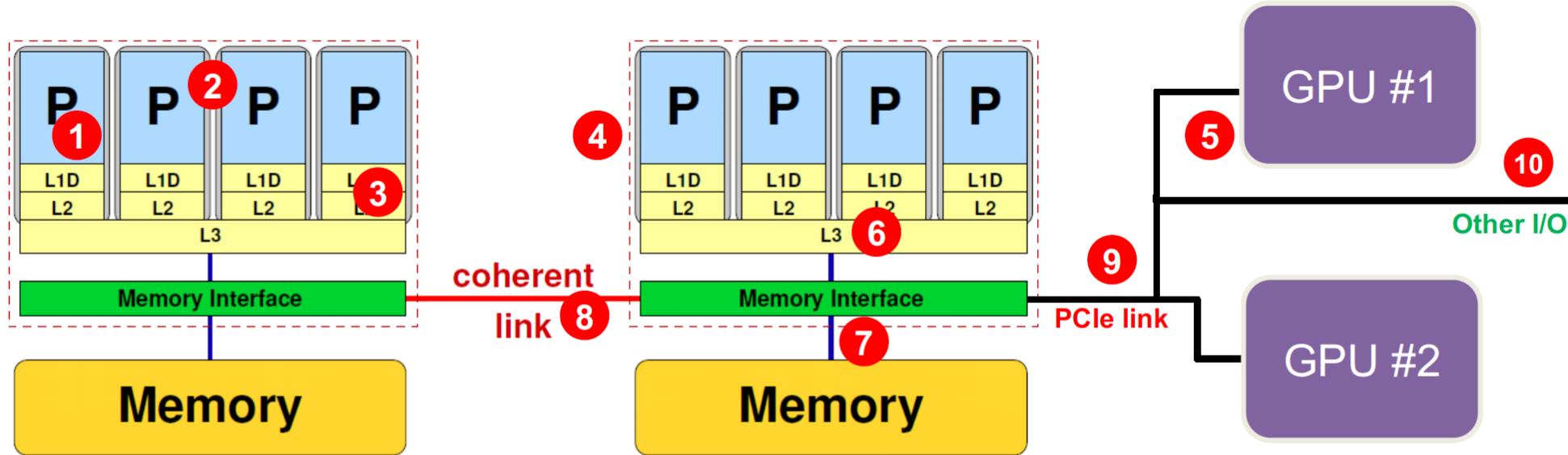
- Comprise the majority of nodes on each system
- Meant for production work
- Work is submitted via a batch scheduler (Slurm, Moab)
- Cannot be logged into unless you already have a job running on them
- Nodes run one job at a time - not shared like login nodes



https://computing.llnl.gov/tutorials/linux_clusters

Parallelism in a modern computer node

Parallel and shared resources within a shared-memory node



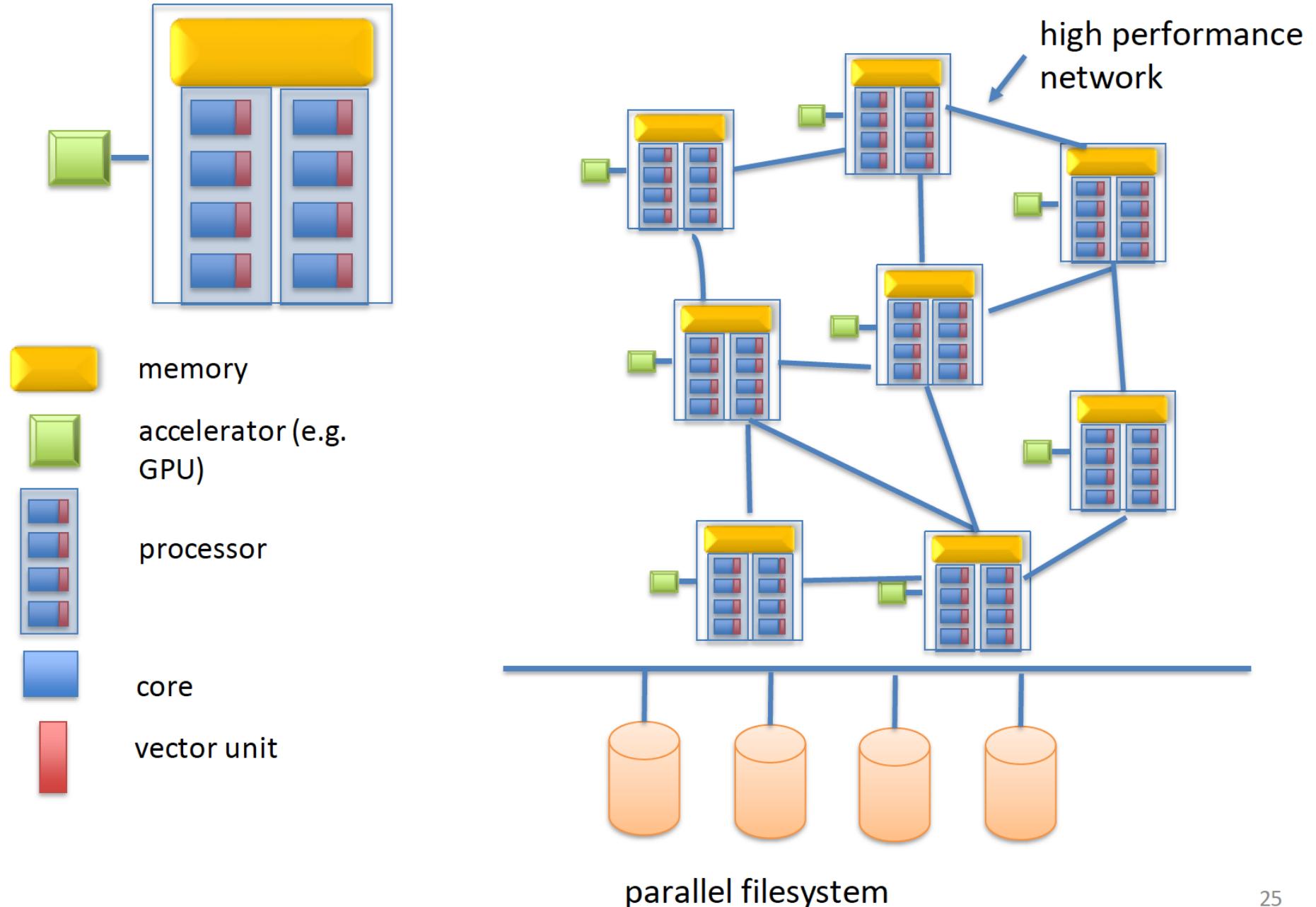
Parallel resources:

- Core 1
- CPU 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

Shared resources:

- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

Putting it all together



Which factors drive the evolution in HPC architecture?

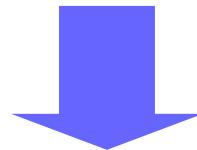
- ❑ The first (super) computers were mainly used by defence organisations and the US Govt (esp. Department of Energy) still makes significant investments. Later they were used for scientific research in a small number of centres.
- ❑ But the high cost of the dedicated components, and the fact that **HPC is not a strong market**, has caused a shift into using commodity or off-the-shelf devices such as processors, disks, memories, networks, etc.
- ❑ This shift has had a number of consequences:
 - ❑ Some manufacturers have changed business or no longer make supercomputers (e.g. SUN Microsystems).
 - ❑ Other supercomputer vendors (e.g. CRAY and SGI) no longer make microprocessors so the market is dominated by one or two brands, i.e. Intel and, to a lesser extent, IBM PowerPC.
 - ❑ Since microprocessors were not designed for HPC, the programmer must work harder to get maximum performance.
- ❑ But porting has become easier as only a few processor types are available and Linux has replaced all the other operating systems. It is now also possible for smaller organisations such as university departments to run small clusters.

Introduction to Parallel Computing

Main Concepts

How faster can we run?

- 12 tasks, each one requiring 1s
- Total serial time: 12s



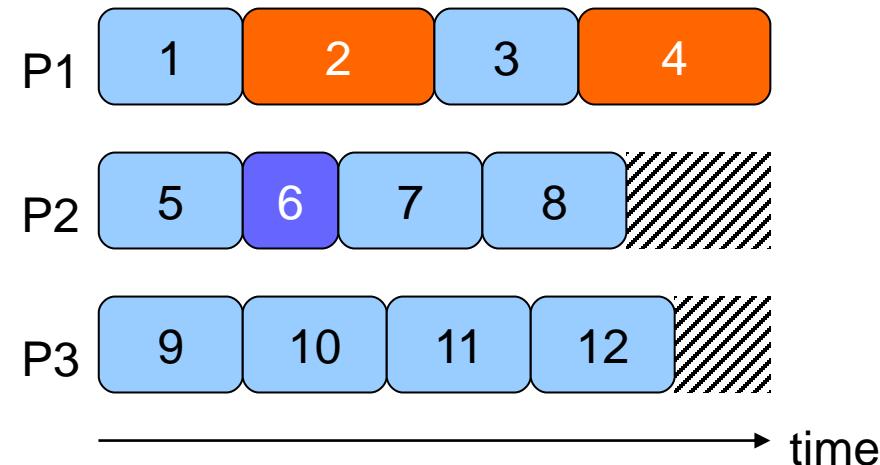
- If we have 3 processors and independent tasks execution time CAN become 4s



→ time

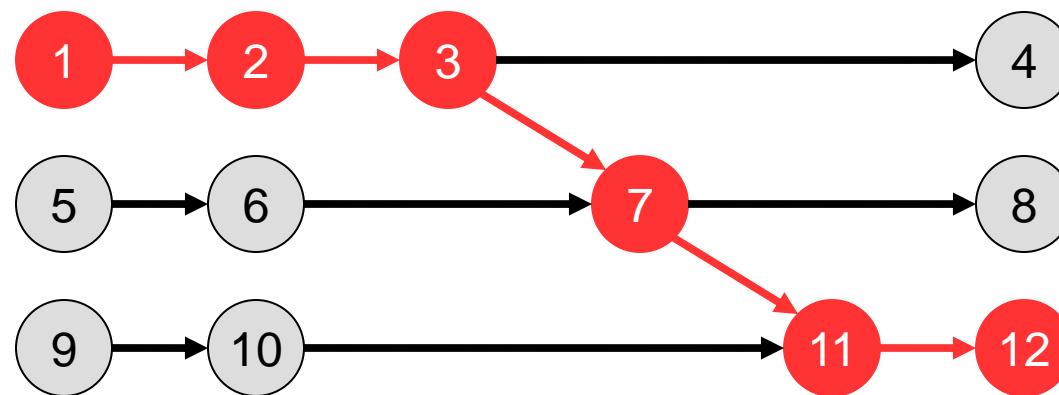
But

- What if the processors can not execute the tasks with the same speed?
I.e., a heterogeneous cluster.
- Or the tasks do not require the same amount of operations?
➤ Load imbalance (ending part of P2 and P3)



And

- What if tasks are dependent?
- Execution time grows from 4s to 6s



*Dependency
graph with
highlighted
critical path*



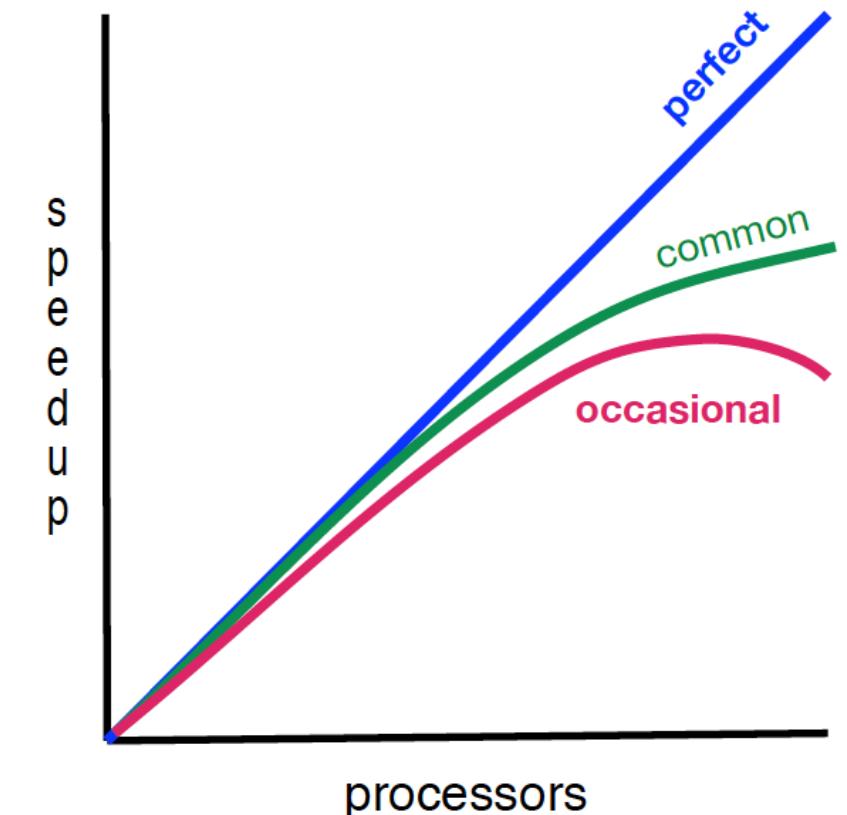
→ time

Scalability

- How much faster can a given problem be solved with p parallel processes instead of one?
- How much more work can be done with p parallel processes instead of 1?
- What impact for the communication requirements of the parallel application have on performance?
- What fraction of the resources is actually used productively for solving the problem?

Speedup

- Speedup $S_p = T_1/T_p$
- In the ideal case, the parallel program requires $1/p$ the time of the sequential program, i.e. $T_p = T_1/p$
- $S_p = p$ is the ideal case of linear speedup
- Realistically, $S_p \leq p$ due to the need of coordination the works and to communicate
- Overhead is the difference $T_o = pT_p - T_1$
- Is it possible to observe $S_p > p$, i.e. a SUPERLINEAR speedup?
 - Yes, mainly for Memory vs Disk

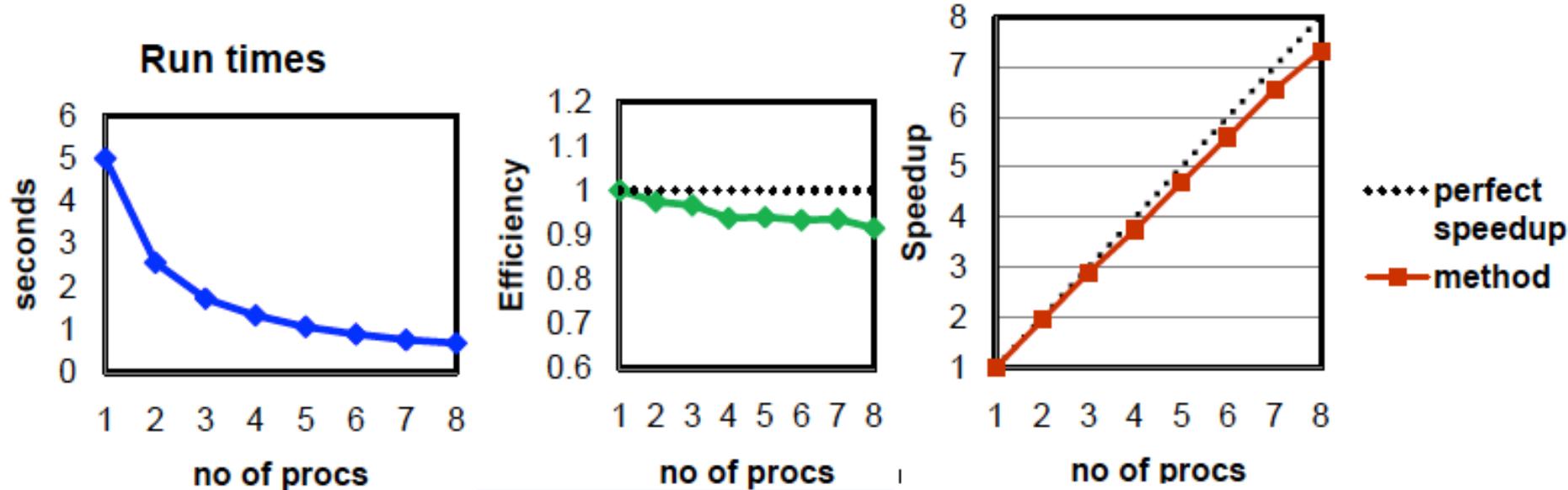


Performance Metrics

The size of the problem is important too

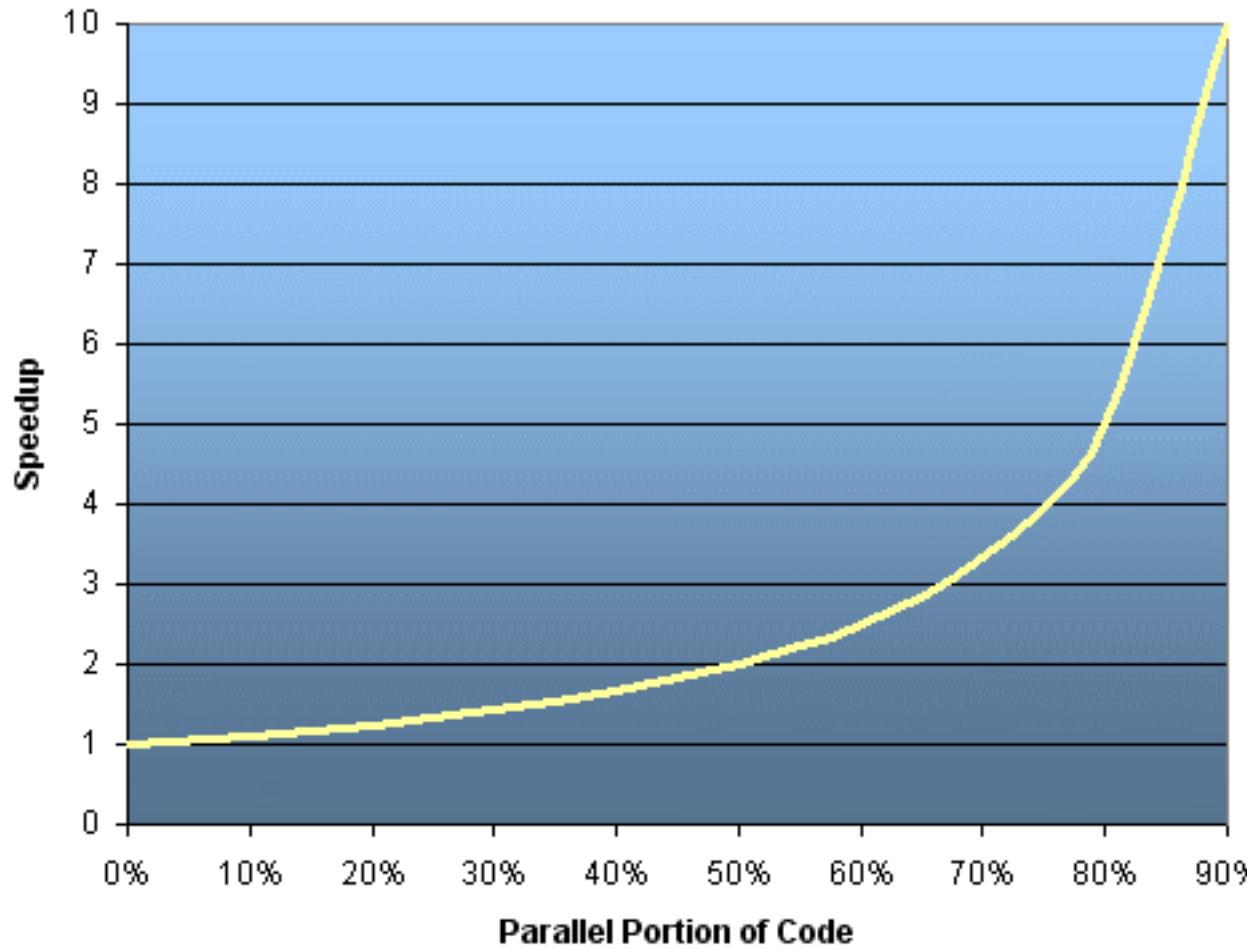
Let $T(n,p)$ be the time to solve a problem of size n using p processors

- Speedup: $S(n,p) = T(n,1)/T(n,p)$
- Efficiency: $E(n,p) = S(n,p)/p$



Amdahl's Law

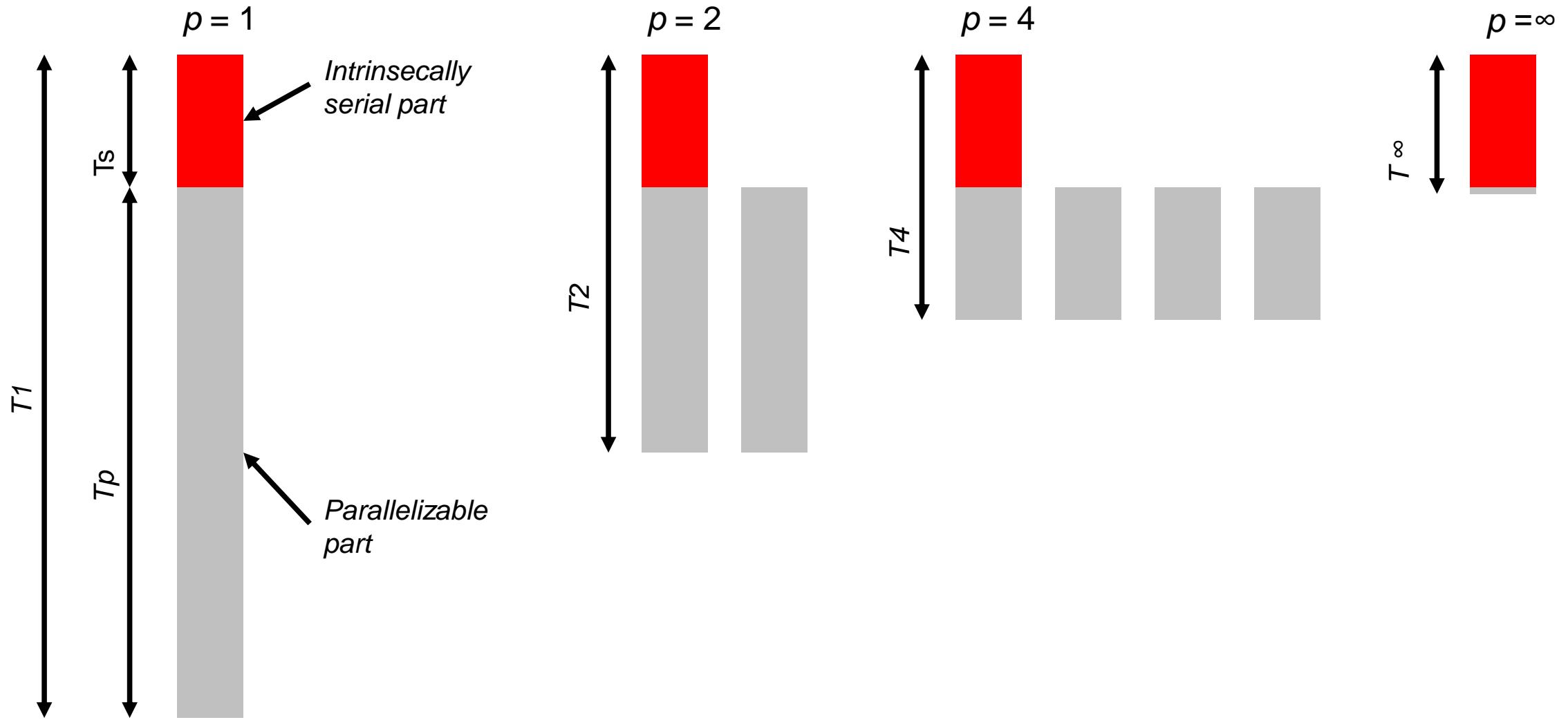
F_p is the parallelizable fraction of the code, with $F_s + F_p = 1$



$$T_p \geq T_1(F_s + \frac{F_p}{p})$$

$$\text{Speedup} = \frac{1}{F_s + \frac{F_p}{p}}$$

Example



Example

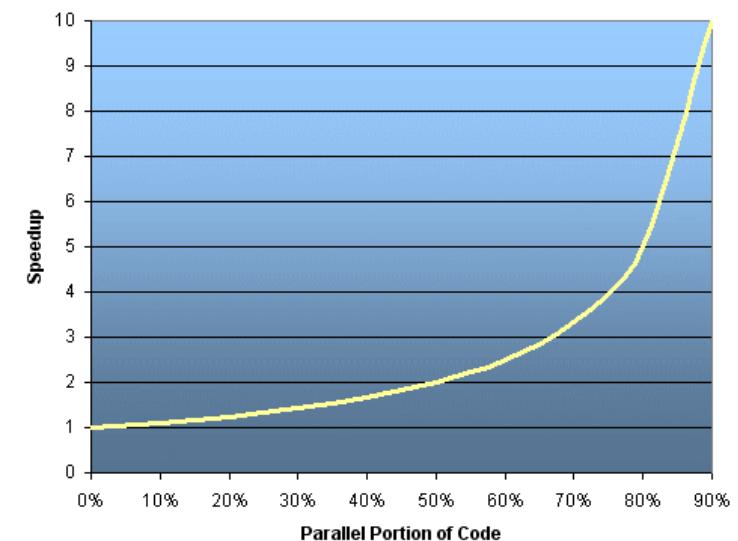
- Suppose that a program has $T_1 = 20\text{s}$
- Assume that 10% of the times spent in a serial portion of the program
- Therefore, the execution time of a parallel version with p processors is

$$20*(0.1 + 0.9 / p)$$

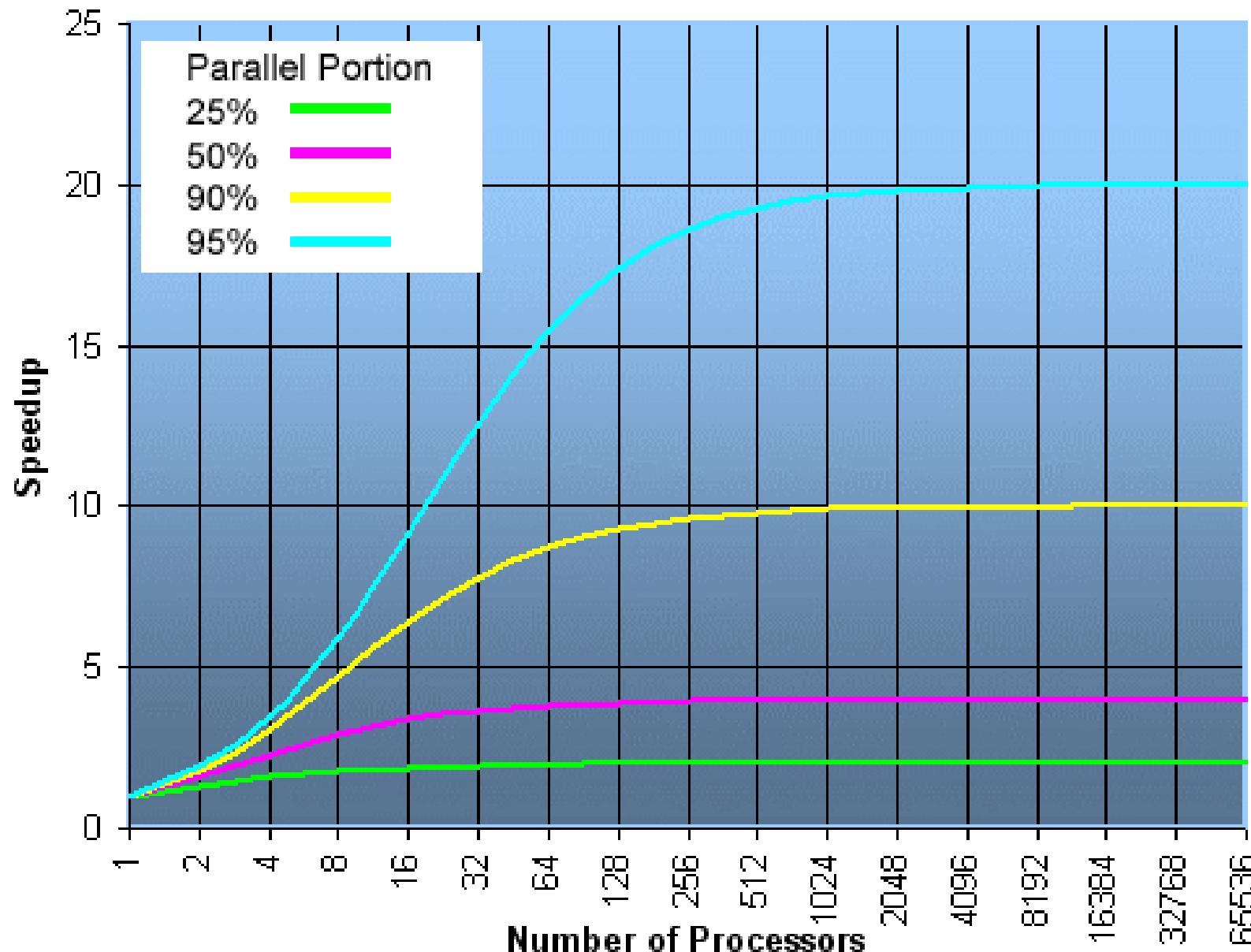
$$20*0.1 + (20*0.9 / p) = 2 + 18/p$$

$$T_p \geq 2$$

$$\text{Best possible speedup} = 20/2 = 10$$



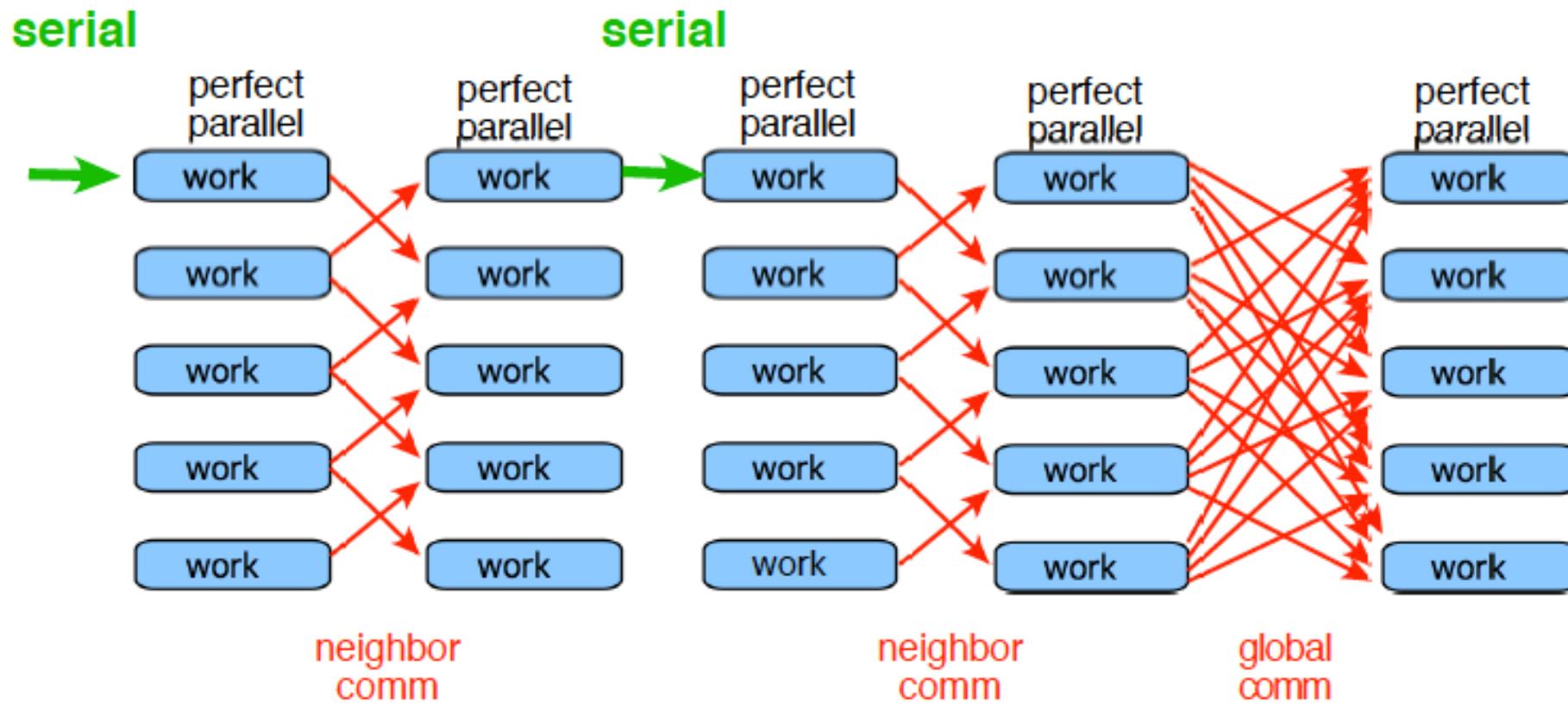
Amdahl's Law



Amdahl Was an Optimist

Parallelization usually adds communications/overheads

$$T_p \geq T_1(F_s + \frac{F_p}{p}) + T_{Ovh}$$



Critical paths

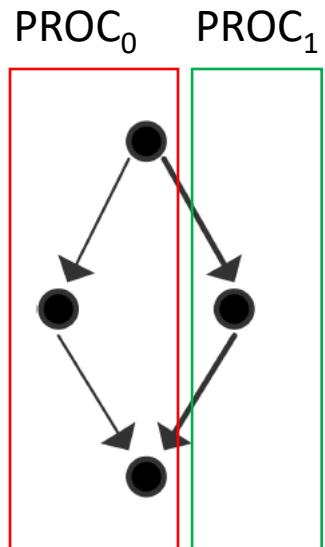
We define the critical path as a (possibly non-unique) chain of dependencies of maximum length. Since the tasks on a critical path need to be executed one after another, the length of the critical path is a lower bound on parallel execution time.

T_1 : the time the computation takes on a single processor

T_p : the time the computation takes with p processors

T_∞ : the time the computation takes if unlimited processors are available

P_∞ : the value of p for which $T_p = T_\infty$



$$T_1 = 4, \quad T_\infty = 3 \quad \Rightarrow T_1/T_\infty = 4/3$$

$$T_2 = 3, \quad S_2 = 4/3, \quad E_2 = 2/3 \quad \text{Actually } 1.333/3$$

$$P_\infty = 2$$

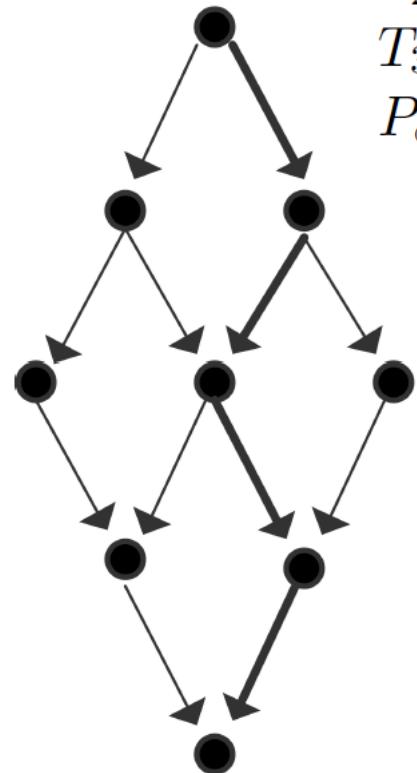
Critical paths

$$T_1 = 9, \quad T_\infty = 5 \quad \Rightarrow T_1/T_\infty = 9/5$$

$$T_2 = 6, \quad S_2 = 3/2, \quad E_2 = 3/4$$

$$T_3 = 5, \quad S_3 = 9/5, \quad E_3 = 3/5$$

$$P_\infty = 3$$



T_1 : the time the computation takes on a single processor

T_p : the time the computation takes with p processors

T_∞ : the time the computation takes if unlimited processors are available

P_∞ : the value of p for which $T_p = T_\infty$

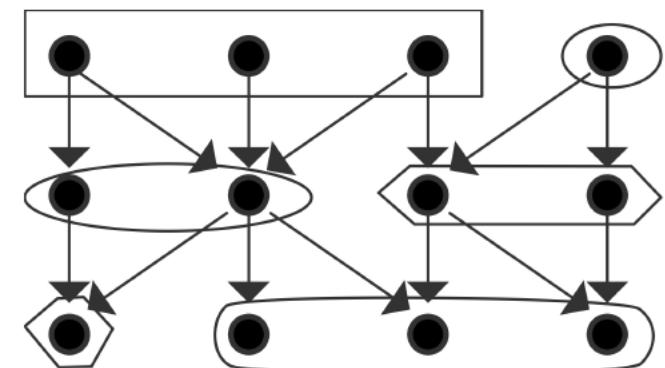
$$T_1 = 12, \quad T_\infty = 4 \quad \Rightarrow T_1/T_\infty = 3$$

$$T_2 = 6, \quad S_2 = 2, \quad E_2 = 1$$

$$T_3 = 4, \quad S_3 = 3, \quad E_3 = 1$$

$$T_4 = 3, \quad S_4 = 4, \quad E_4 = 1$$

$$P_\infty = 4$$



Amdahl was a Pessimist

Superlinear speedup is very rare but possible

- Parallel computer has p times as much RAM so higher fraction of program memory in RAM instead of disk.

An important reason for using parallel computers

- You can use more complex algorithms , where F_s is lower.

A useful side-effect of parallelization

- In general, the time spent in serial portion of code is a decreasing fraction of the total time as problem size increases.
- In the previous example $T_s = 2$, $T_p = 18$.

What happens if $T_1 = (2 + 180)$? $F_p = 0.99$, $S = 1/0.01 = 100$

Scaling efficiency

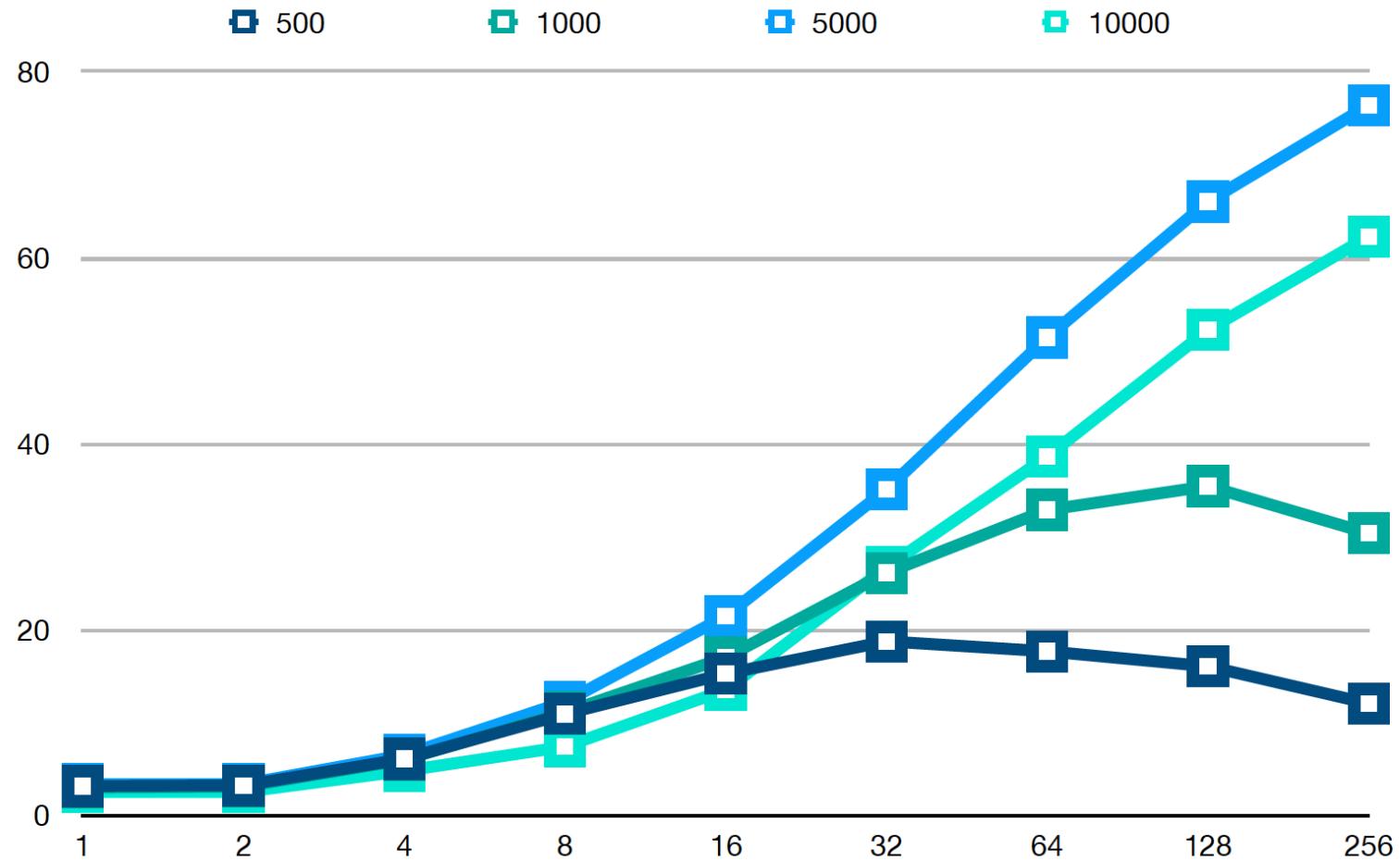
- Strong Scaling: increase the number of processors p keeping the *total problem size* fixed
 - The total amount of work remains constant
 - The amount of work for a single processor decreases as p increases
 - Goal: reduce the total execution time by adding more processors
- Weak Scaling: increase the number of processors p keeping the *per-processor work* fixed
 - The total amount of work grows as p increases
 - Goal: solve larger problems within the same amount of time
- To present the performance of a parallel program you must present the strong scaling with proper p values.
 - Es for a single node of the INFN cluster you should consider at least 256 (4 threads x core), 64 (actual cores) and the sequential time.
 - For the full cluster consider that, in principle, you can use 256 (single node) \times 12 (nodes) processors

A good example

Threads or cpu number

V2	seq	1	2	4	8	16	32	64	128	256	
Data size	500	2,450	3,125	3,194	6,050	10,889	15,218	18,703	17,627	16,014	12,010
	1000	9,099	2,972	3,070	5,966	11,056	16,975	26,071	32,848	35,404	30,329
	5000	251,723	3,334	3,335	6,475	12,189	21,416	35,054	51,404	66,034	76,395
	10000	754,078	2,491	2,490	4,761	7,370	13,483	26,680	38,599	52,221	62,274

OpenMP, single node

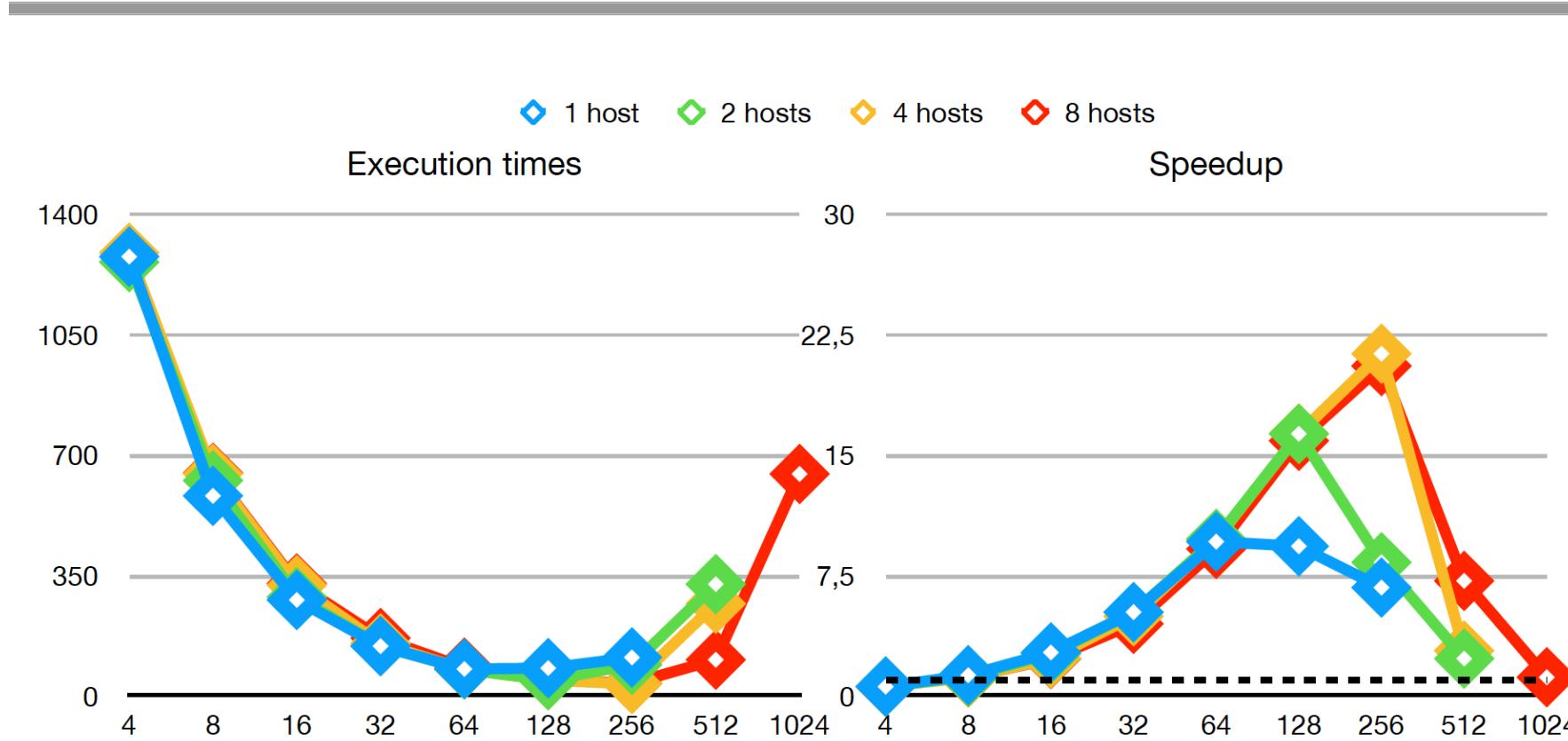


A good example

	10000	seq	2	4	8	16	32	64	128	256	512	1024
1	760,510	-	0,596	1,307	2,721	5,231	9,619	9,333	6,762	-	-	-
2	760,510	-	0,603	1,214	2,640	5,218	9,792	16,335	8,335	2,338	-	-
4	760,510	-	0,590	1,177	2,356	4,969	9,740	16,342	21,317	2,832	-	-
8	760,510	-	-	1,172	2,323	4,501	9,165	15,900	20,553	7,176	1,178	-

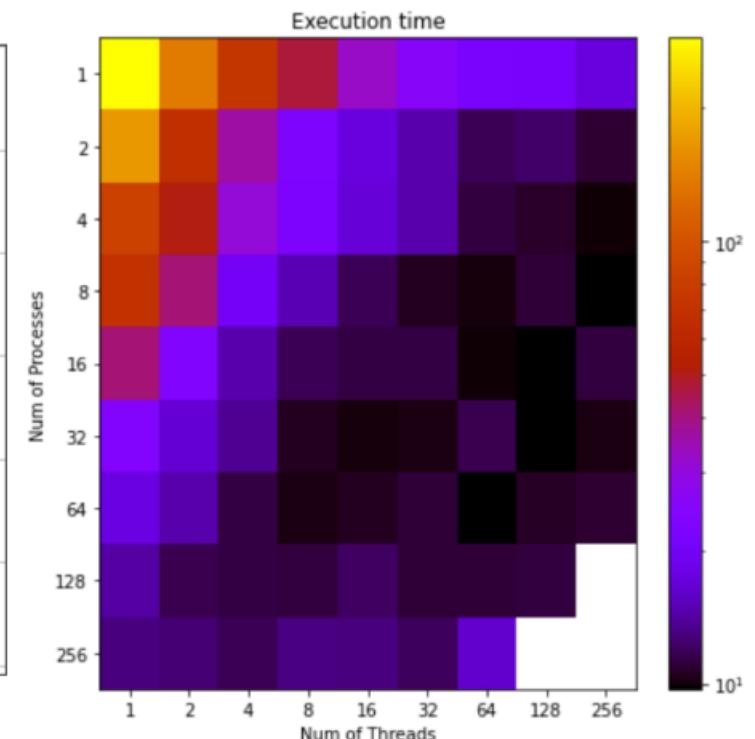
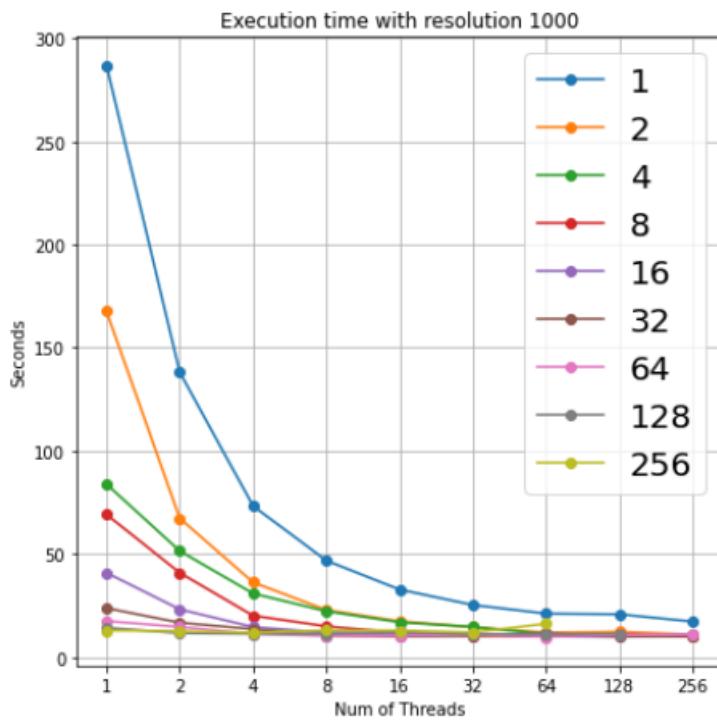
nodes

MPI, cluster

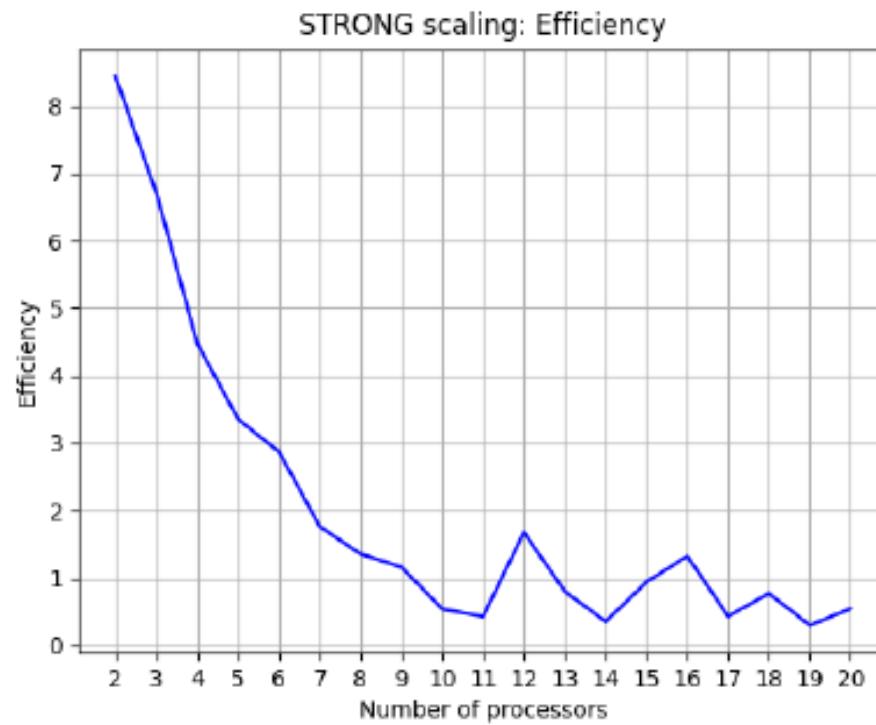
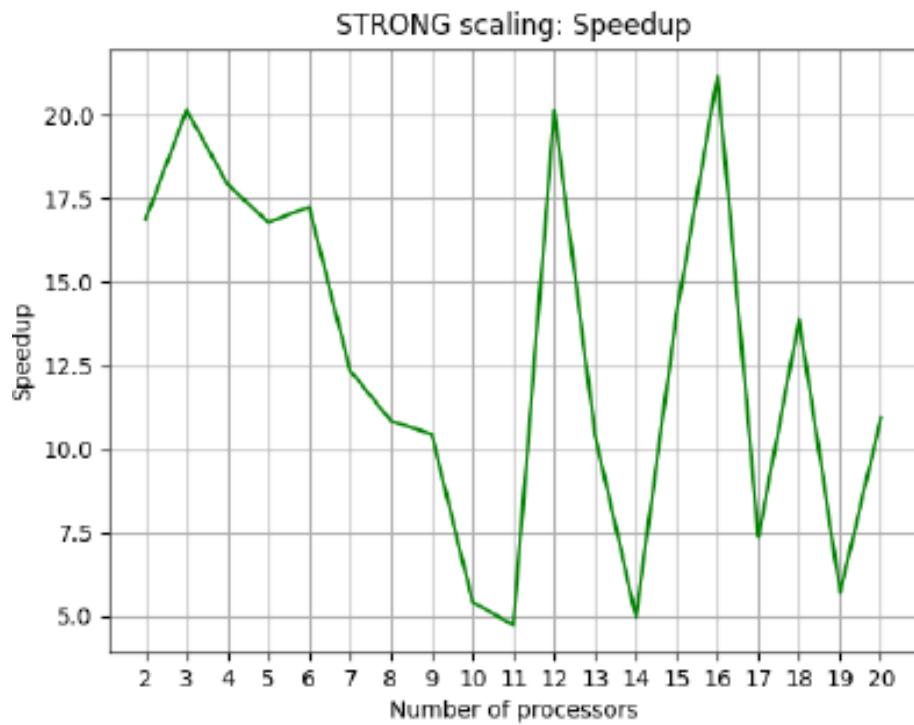


Another good example: MPI+OMP

Resolution 1000: Execution time on 8 Machine									
Processes / Threads	1	2	4	8	16	32	64	128	256
1	286.97	138.035	73.258	46.77	32.72	25.265	21.073	20.75	17.195
2	167.646	67.229	36.229	22.812	17.362	14.583	11.851	12.232	10.921
4	83.845	51.415	30.828	22.211	16.886	14.581	11.199	10.718	9.983
8	69.128	40.837	20.067	14.915	11.753	10.401	10.008	11.04	9.78
16	40.857	23.134	14.554	11.849	11.297	11.317	9.971	9.777	11.112
32	23.73	16.669	13.55	10.526	9.99	10.212	11.664	9.84	10.218
64	17.452	14.62	11.277	10.231	10.4	11.021	9.726	10.641	10.881
128	14.209	11.583	11.251	11.162	12.148	11.009	11.058	11.218	nan
256	12.892	12.6	11.833	13.012	12.89	11.986	16.172	nan	nan

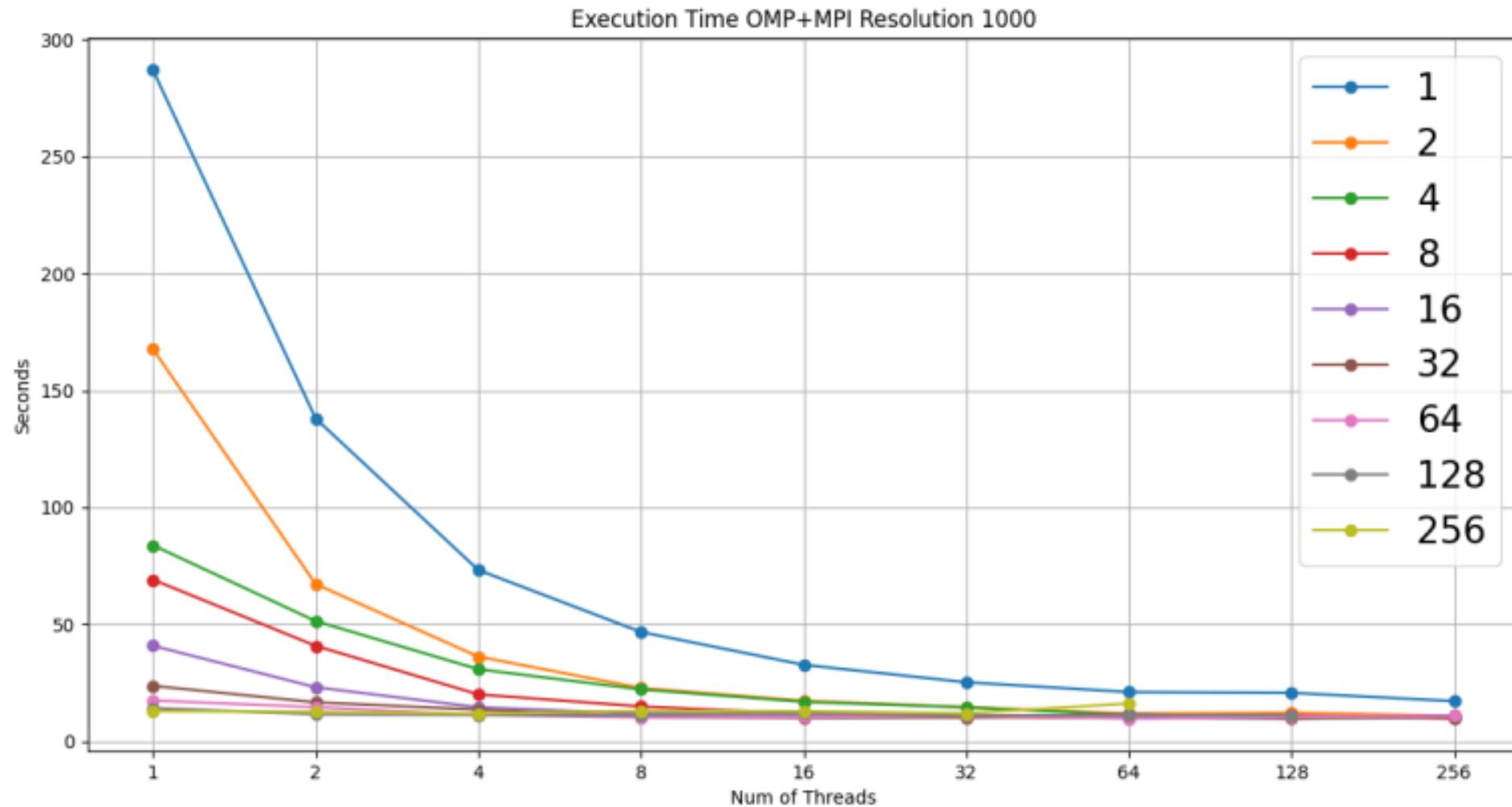


Poor presentation



At least if you don't want to discuss IN DETAILS why $S_{11} = 5$ and $S_{12} = 20$

Useless results



Possible misleading representation

The ideal speedup is
Number of node * threads

i.e. 430 vs 64×8

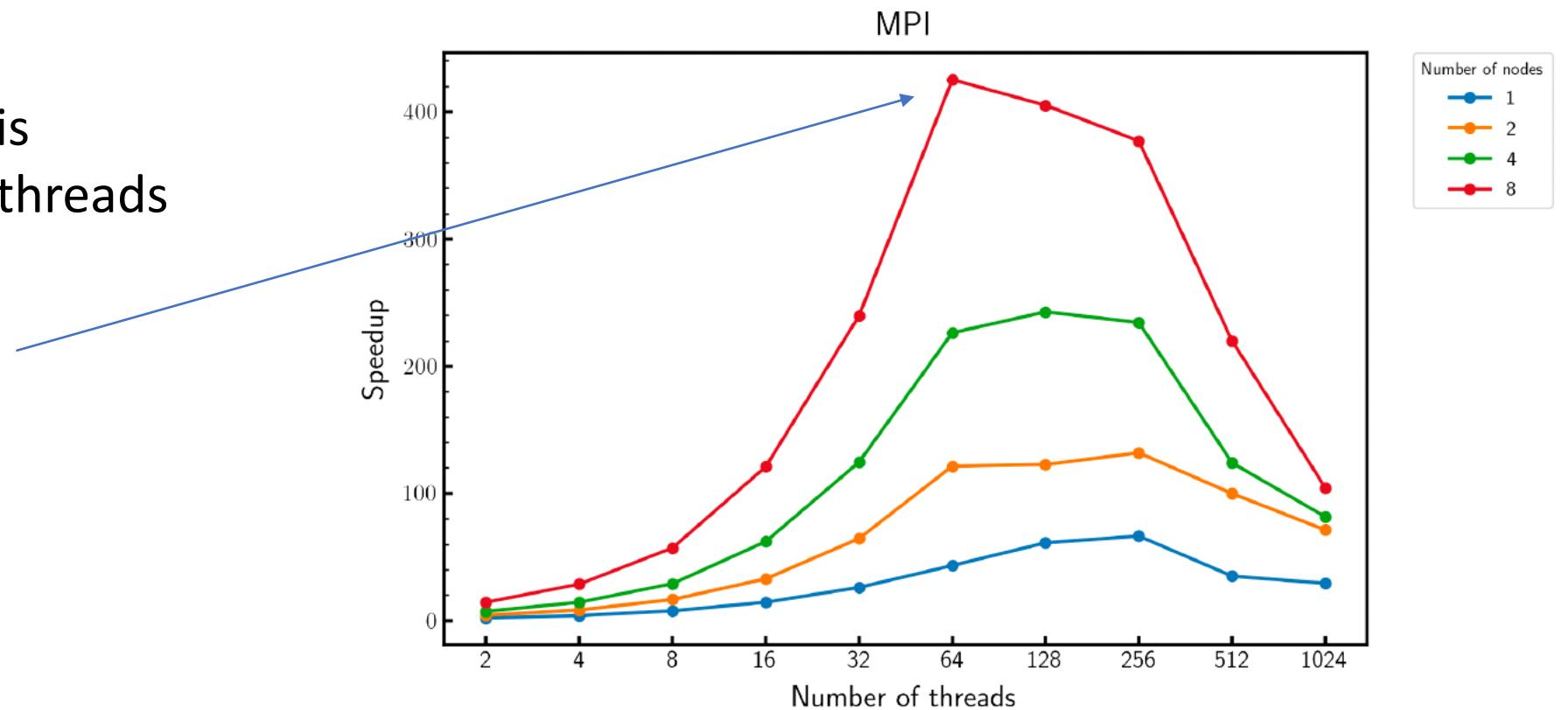


Fig. 11: MPI Speedup, size of the game board 10000x10000

Remember the main lesson

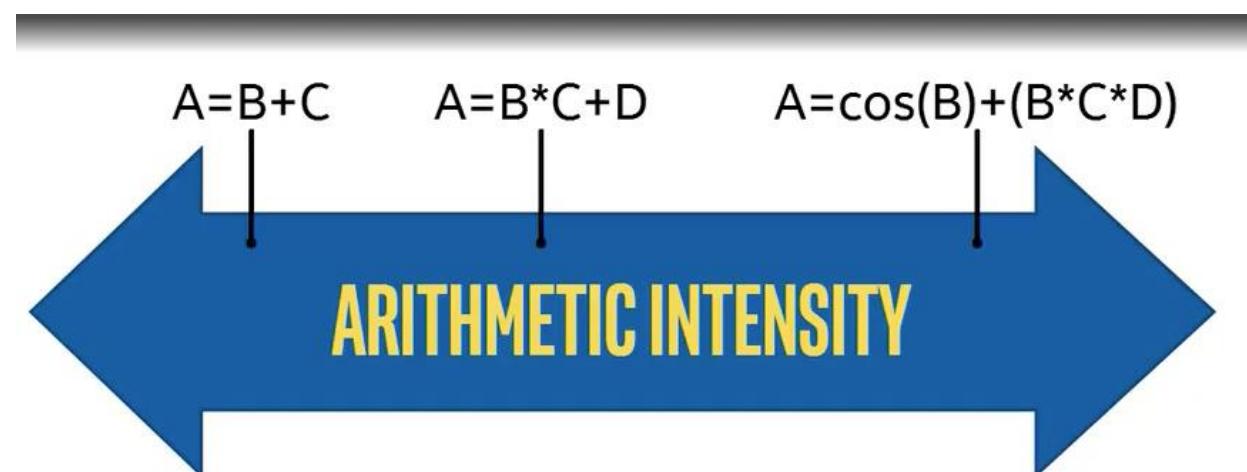
- Linear speedup is rare, due to communication overhead, load imbalance, algorithm/architecture mismatch, etc.
- Further, essentially nothing scales to arbitrarily many processors.
- However, for most users, the important question is:

**Have I achieved acceptable performance on
my software for a suitable range of data
and the resources I'm using?**

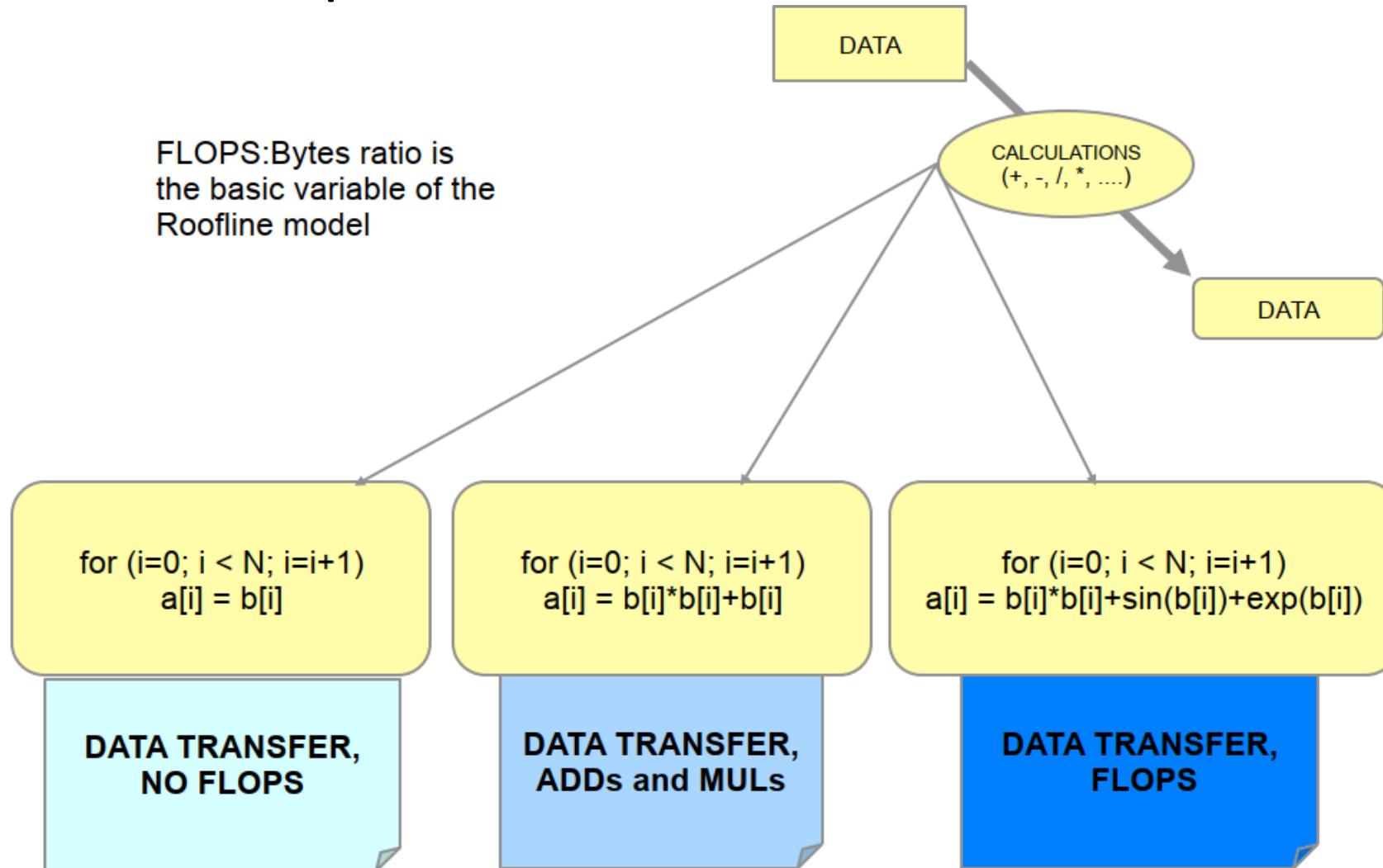
- **In the final project is important not only to achieve acceptable performance but also to be able to present and discuss them.**
The exam mark is proportional to this...

The Roofline performance model

- The Amdahl's Law is an example of bound and bottleneck analysis, i.e. rather than try to predict performance, it provides valuable insight into the primary factors affecting the performance of computer systems. In particular, the critical influence of the system bottleneck is highlighted and quantified
- Here we analyse a complementary model that relates processor performance to off-chip memory traffic
- If n is the number of data items that an algorithm operates on, and $f(n)$ the number of operations it takes, then the **operation/arithmetic intensity** is $f(n)/n$.



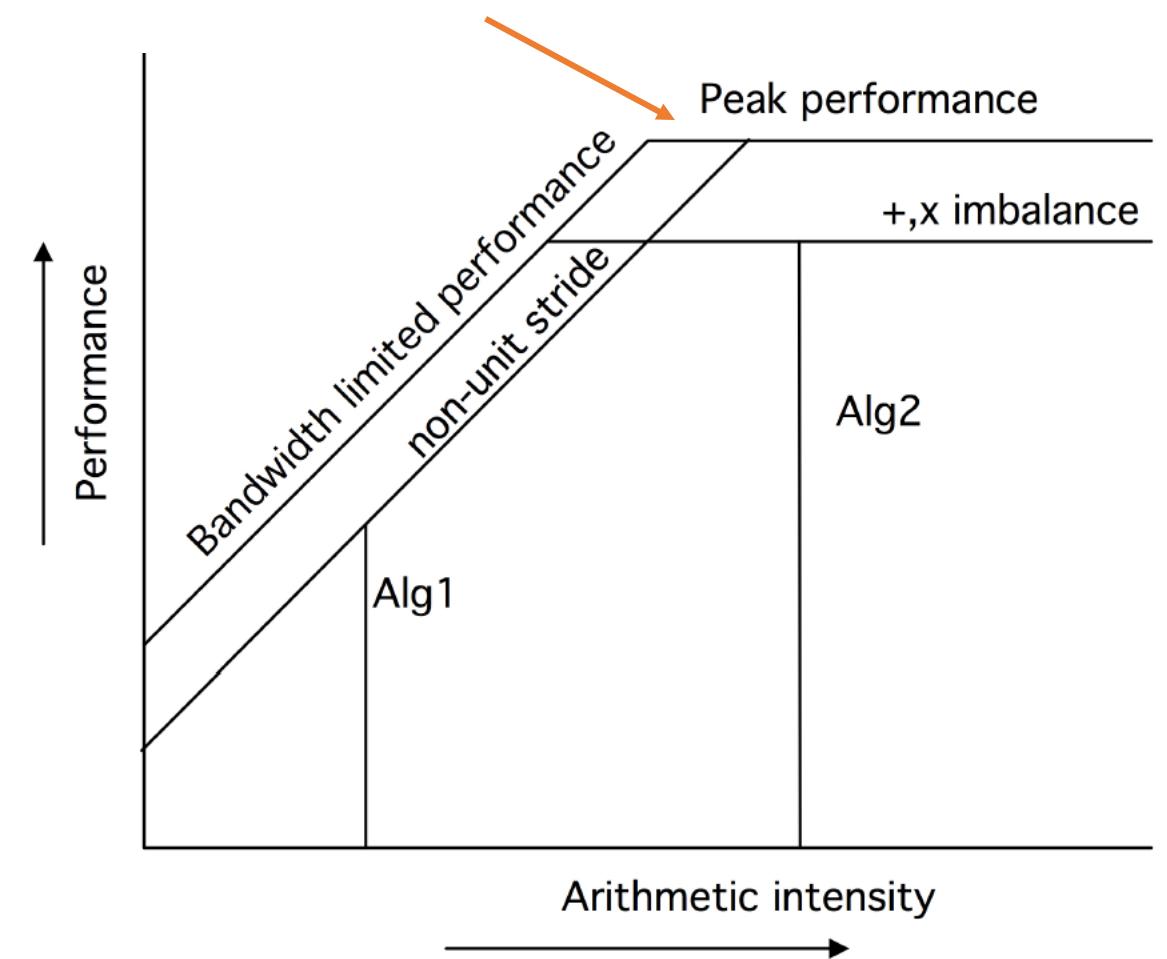
Basic concepts



The roofline performance model

- The peak performance of a system is an **absolute bound** on the achievable performance
- It is achieved **only if** every aspect of a CPU is **perfectly used**.
- The calculation of this number is purely based on CPU properties and clock cycle
- Limiting factor: load imbalance, bad use of the cache...

$$\frac{\text{operations}}{\text{second}} = \frac{\text{operations}}{\text{data item}} \cdot \frac{\text{data items}}{\text{second}}$$



An example

- Machine parameter #1:
- Machine parameter #2:
- Code characteristic:

Peak performance: $P_{peak} \left[\frac{F}{s} \right]$

Memory bandwidth: $b_S \left[\frac{B}{s} \right]$

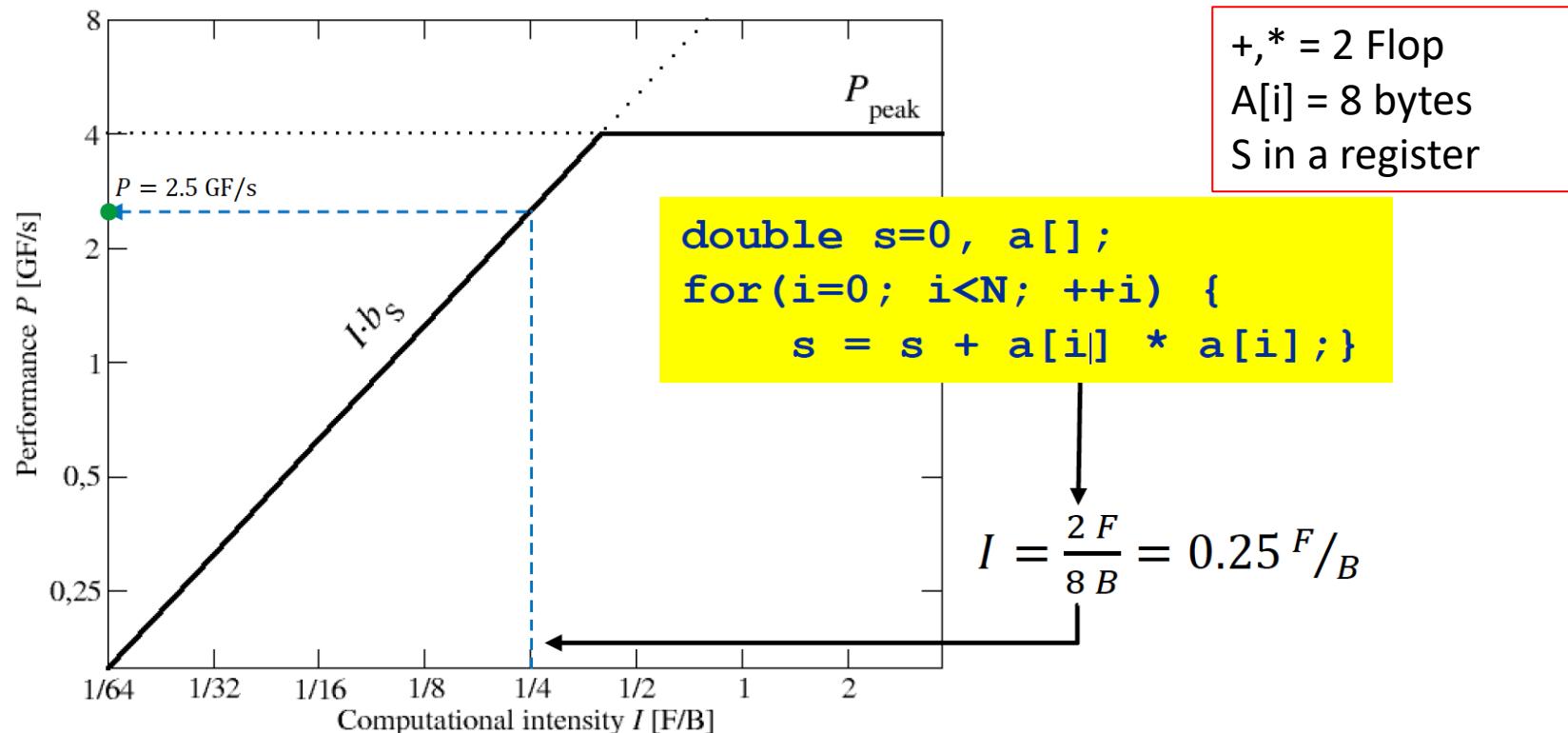
Computational Intensity: $I \left[\frac{F}{B} \right] \frac{\text{Flops}}{\text{Bytes}}$

Machine properties:

$$P_{peak} = 4 \frac{\text{GF}}{\text{s}}$$

$$b_S = 10 \frac{\text{GB}}{\text{s}}$$

Application property: I



Arithmetic intensity

```
double a[], b[];  
for(i=0; i<N; ++i) {  
    a[i] = a[i] + b[i];}
```

$$B_C = 24B / 1F = 24 \text{ B/F}$$
$$I = 0.042 \text{ F/B}$$

```
double a[], b[];  
for(i=0; i<N; ++i) {  
    a[i] = a[i]+ s * b[i];}
```

$$B_C = 24B / 2F = 12 \text{ B/F}$$
$$I = 0.083 \text{ F/B}$$

```
float s=0, a[];  
for(i=0; i<N; ++i) {  
    s = s + a[i] * a[i];}
```

Scalar – can be kept in register

$$B_C = 4B/2F = 2 \text{ B/F}$$
$$I = 0.5 \text{ F/B}$$

```
float s=0, a[], b[];  
for(i=0; i<N; ++i) {  
    s = s + a[i] * b[i];}
```

Scalar – can be kept in register

$$B_C = 8B / 2F = 4 \text{ B/F}$$
$$I = 0.25 \text{ F/B}$$

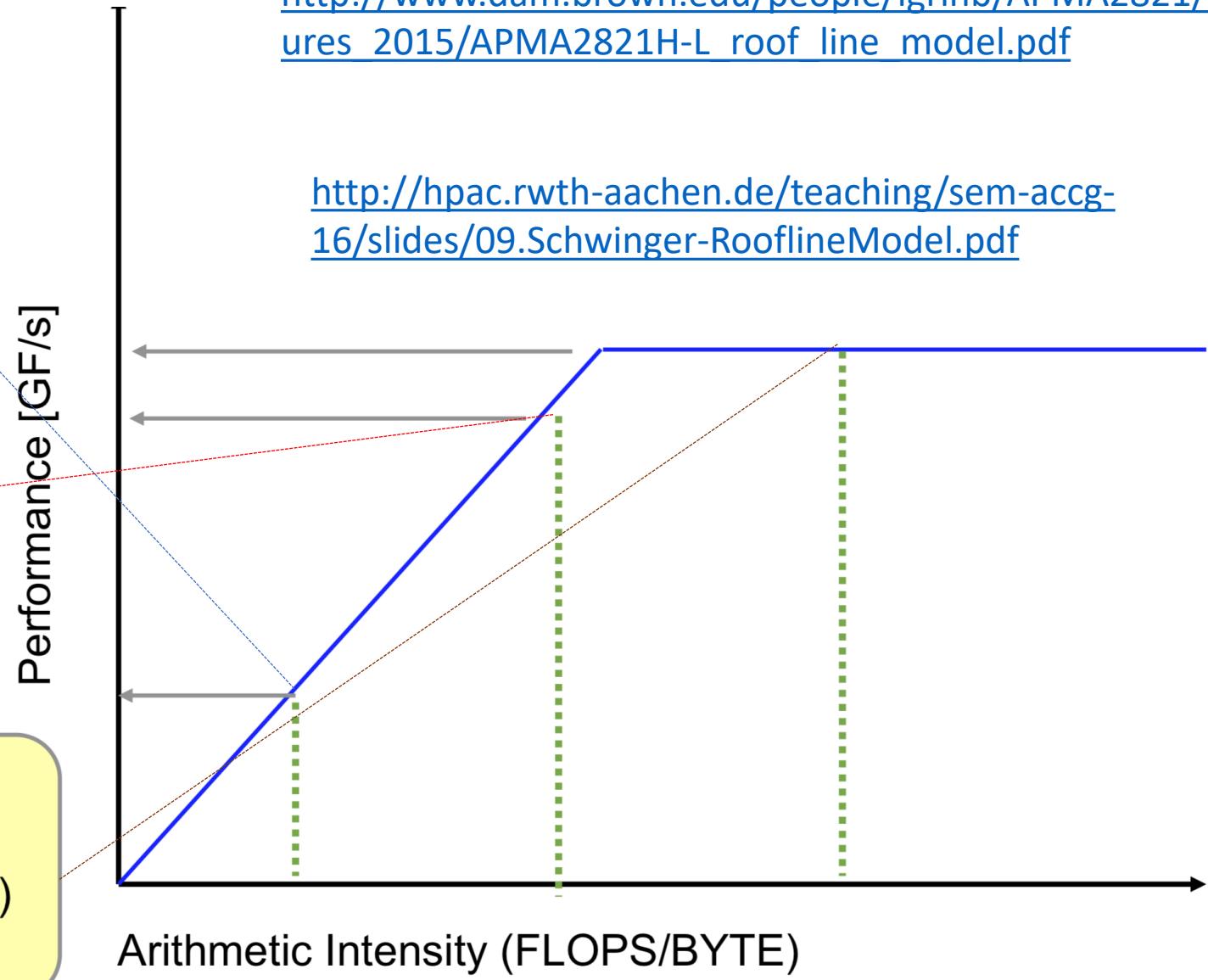
Scalar – can be kept in register

Other examples

```
for (i=0; i < N; i=i+1)  
    a[i] = 2.3*b[i]
```

```
for (i=0; i < N; i=i+1)  
    a[i] = b[i]*b[i]+b[i]
```

```
for (i=0; i < N; i=i+1)  
    a[i] = b[i]*b[i]+sin(b[i])+exp(b[i])
```



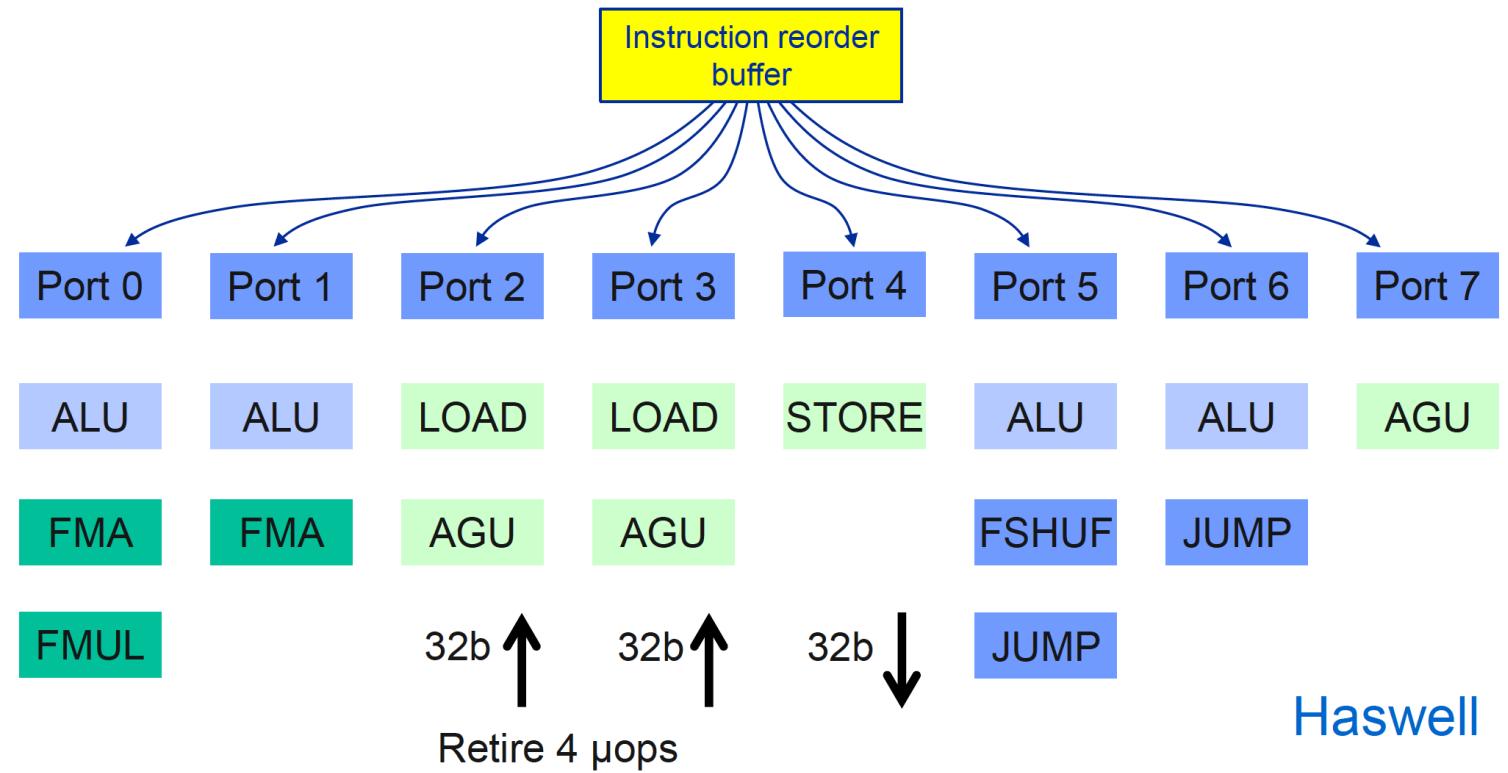
http://www.dam.brown.edu/people/lgrinb/APMA2821/Lectures_2015/APMA2821H-L_roof_line_model.pdf

<http://hpac.rwth-aachen.de/teaching/sem-accg-16/slides/09.Schwinger-RooflineModel.pdf>

Estimating P_{peak}

- Per cycle with AVX, SSE, or scalar it can execute
 - 2 LOAD AND 1 STORE
 - 2 instructions from 2 FMA, 2 MULT, 1 ADD
- Overall maximum of 4 instructions per cycle
- 1 AVX instruction = 4 DP or 8 SP scalar operations

Haswell port scheduler model:



```

double *A, *B, *C, *D;
for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}

```

One AVX iteration (1.5 cycles) does $4 \times 2 = 8$ flops

$$2.3 \cdot 10^9 \text{ cy/s} \cdot \frac{8 \text{ flops}}{1.5 \text{ cy}} = 12.27 \frac{\text{Gflops}}{\text{s}} \quad P_{\text{peak}}$$

$$12.27 \frac{\text{Gflops}}{\text{s}} \cdot 16 \frac{\text{bytes}}{\text{flop}} = 196 \frac{\text{Gbyte}}{\text{s}} \quad B_s$$

Assembly code (AVX2+FMA, no additional unrolling):

```

.B2.9:
vmovupd    (%rdx,%rax,8), %ymm2    # LOAD
vmovupd    (%r12,%rax,8), %ymm1    # LOAD
vfmadd213pd (%rbx,%rax,8), %ymm1, %ymm2 # LOAD+FMA
vmovupd    %ymm2, (%rdi,%rax,8)    # STORE
addq        $4, %rax
cmpq        %r11, %rax
jb          .B2.9
# remainder loop omitted

```

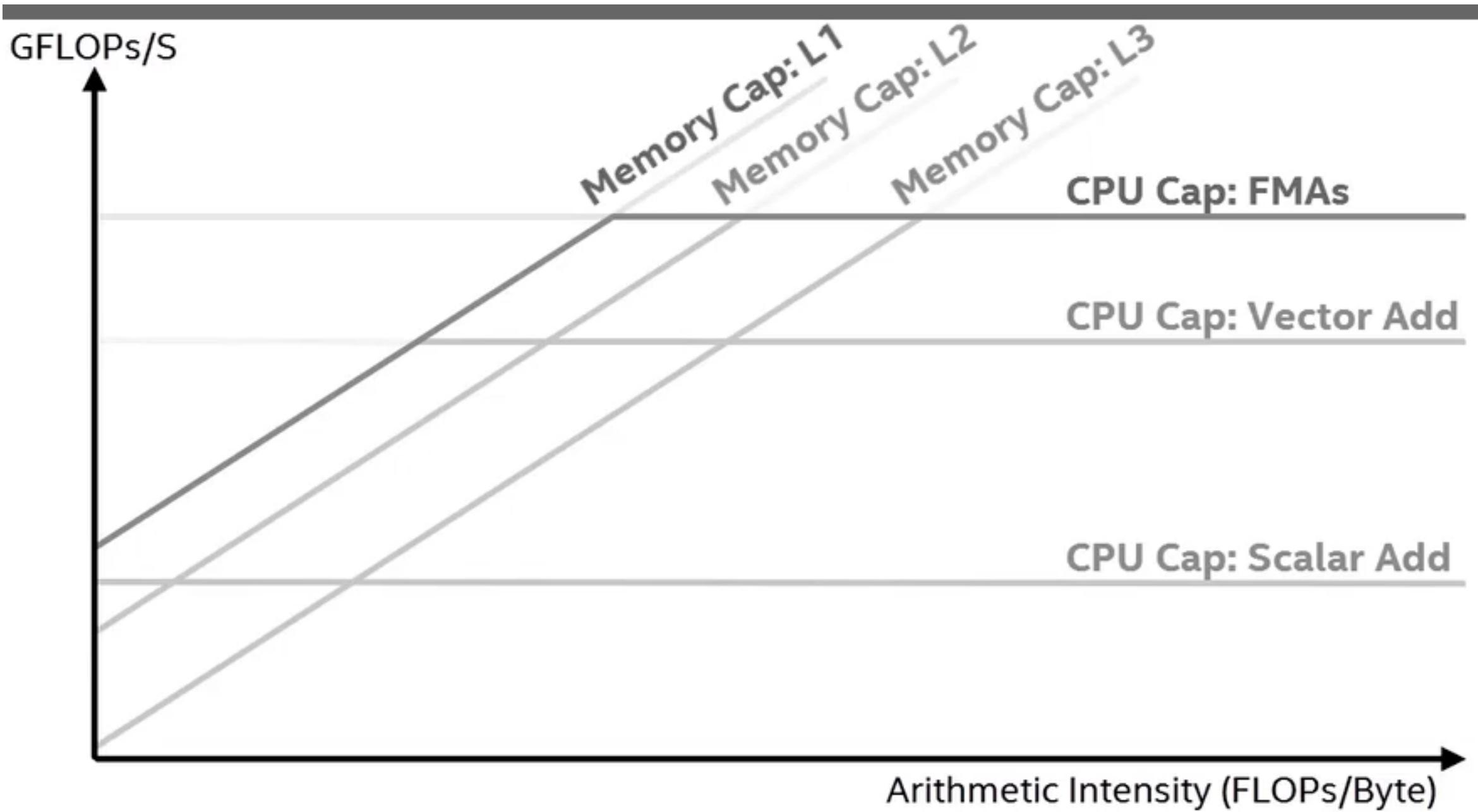
Minimum number of cycles to process one AVX-vectorized iteration
(equivalent to 4 scalar iterations) on one core?

Cycle 1: LOAD + LOAD + STORE

Cycle 2: LOAD + LOAD + FMA + FMA

Cycle 3: LOAD + LOAD + STORE

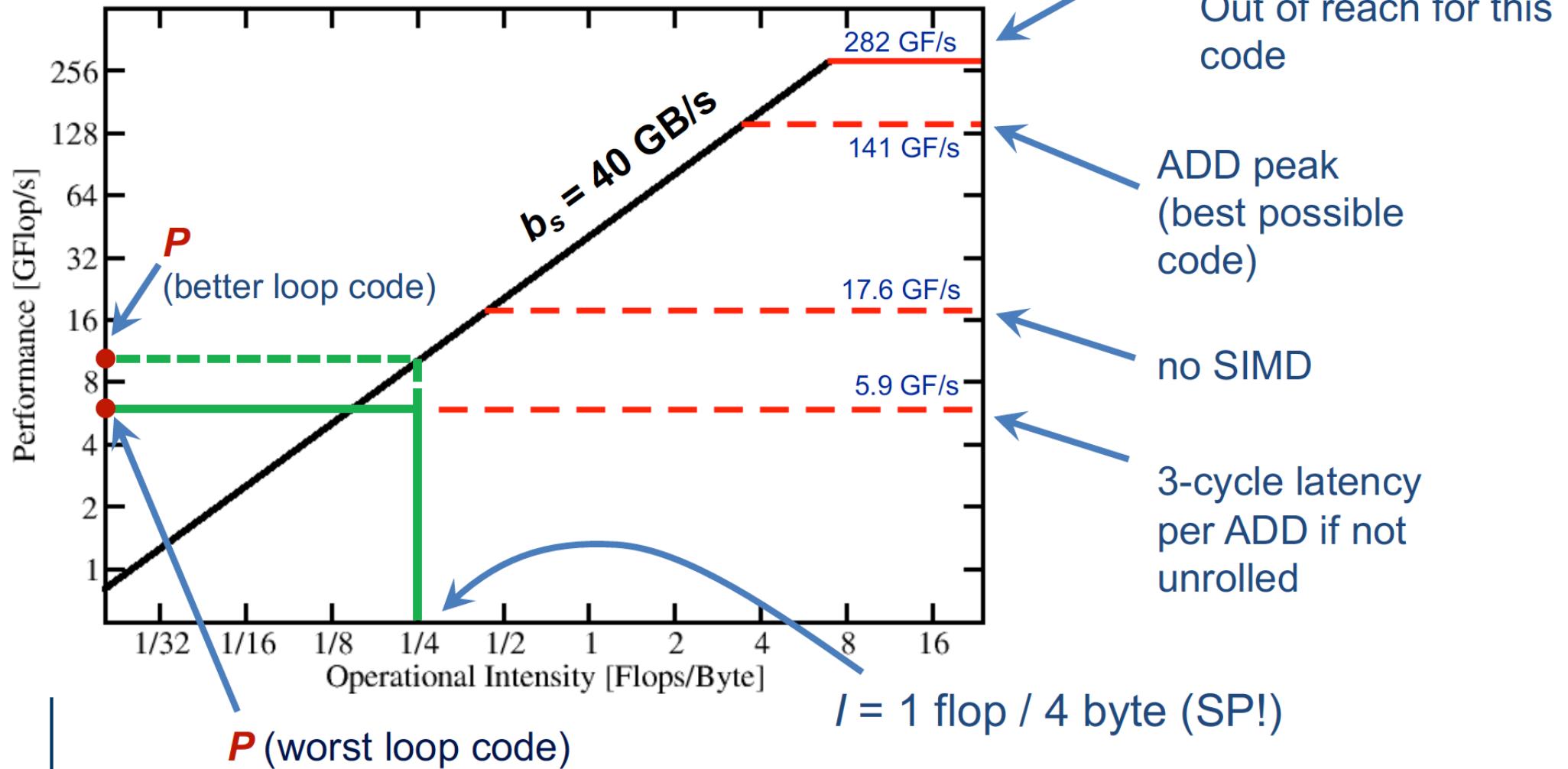
Answer: 1.5 cycles



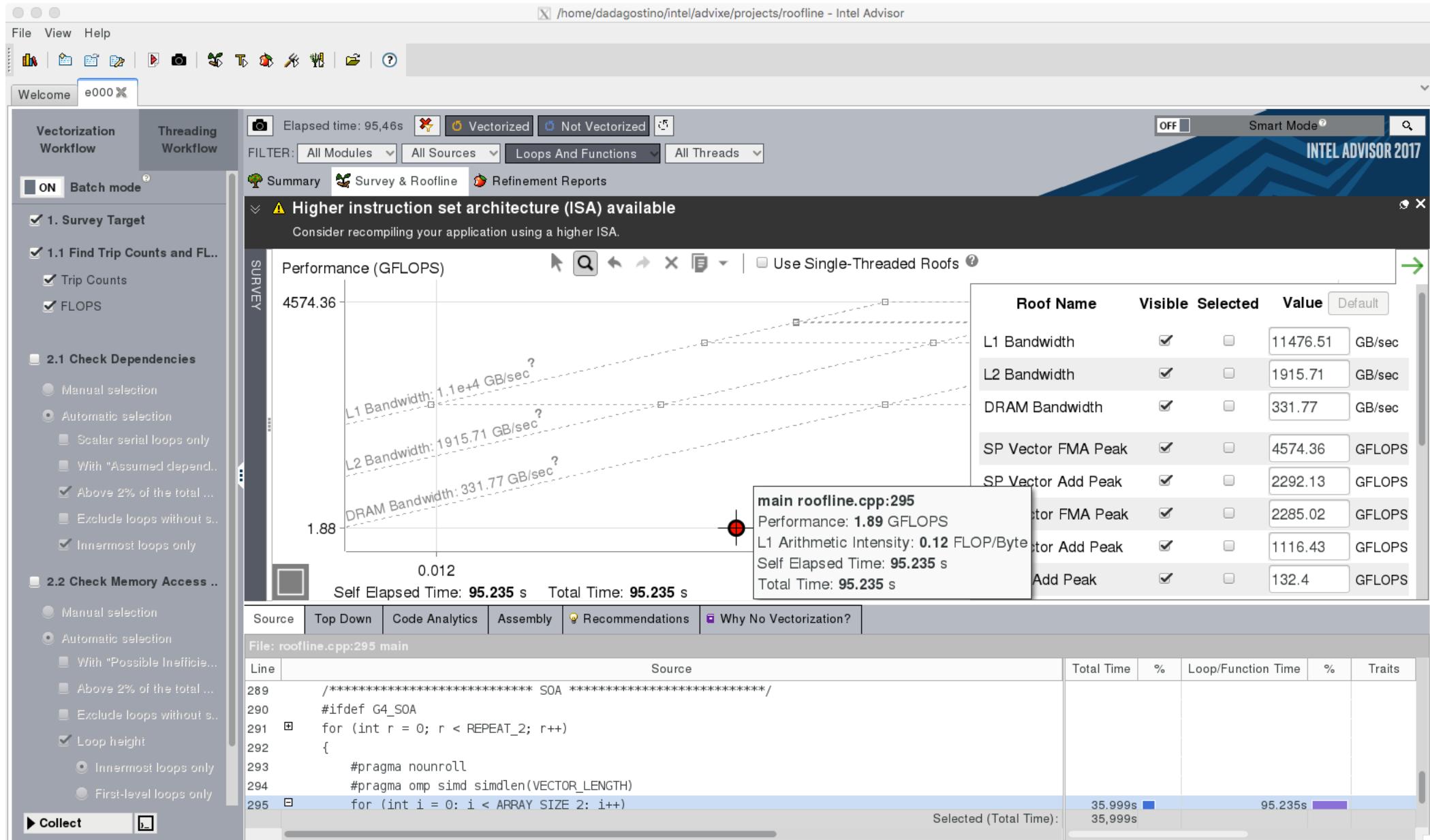
Example: `do i=1,N; s=s+a(i); enddo`

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_S)$$



Roofline model and Intel



Parallel Programming Models

Message Passing Interface (MPI).

- Allows parallel processes to communicate via sending “messages” (i.e. data). Most standard way of communication between nodes, but can also be used within a node.

OpenMP

- Allows parallel processes to communicate via shared memory in a node. Cannot be used between shared memory nodes.

Hybrid MPI+OpenMP

- Combines both MPI+OpenMP. A situation could be to use OpenMP within a shared memory node and MPI between nodes.

OpenAcc

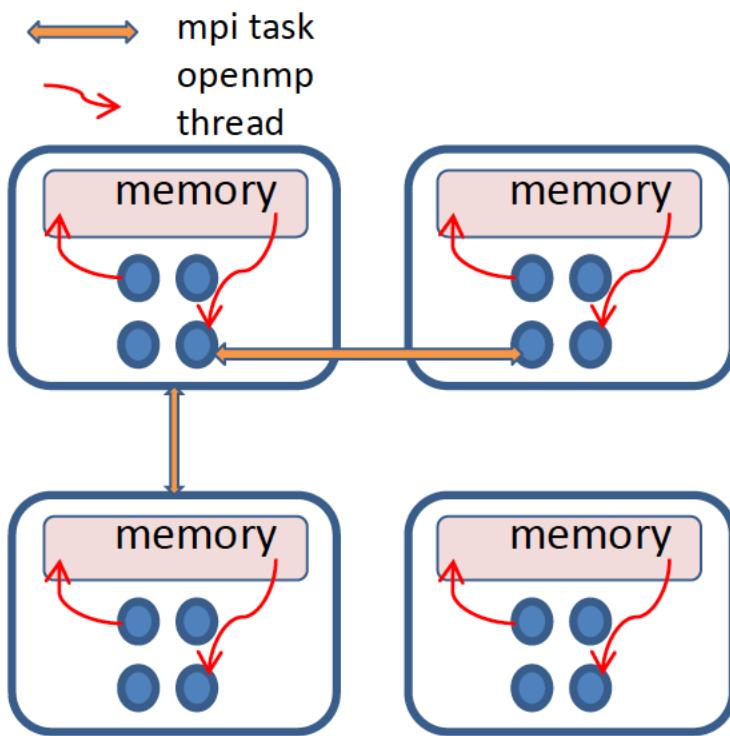
- Similar to OpenMP but used to program devices such as GPUs.

CUDA

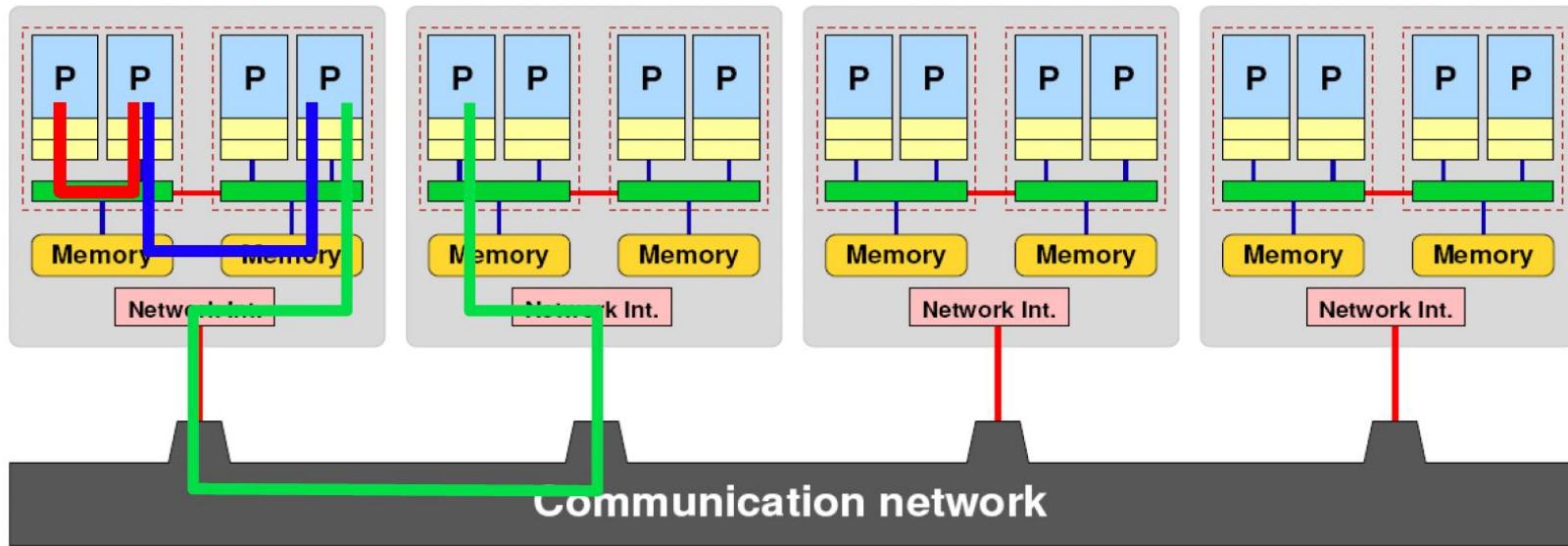
- Nvidia extension to C/C++ for GPU programming.

OpenCL

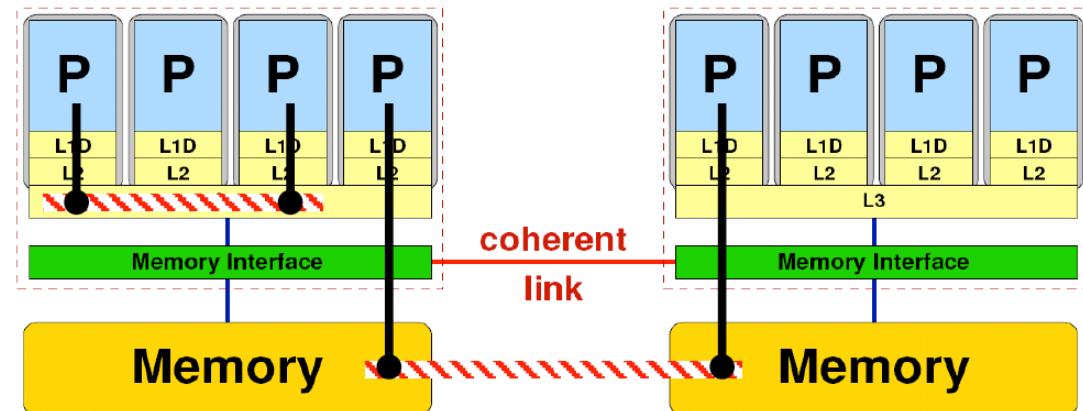
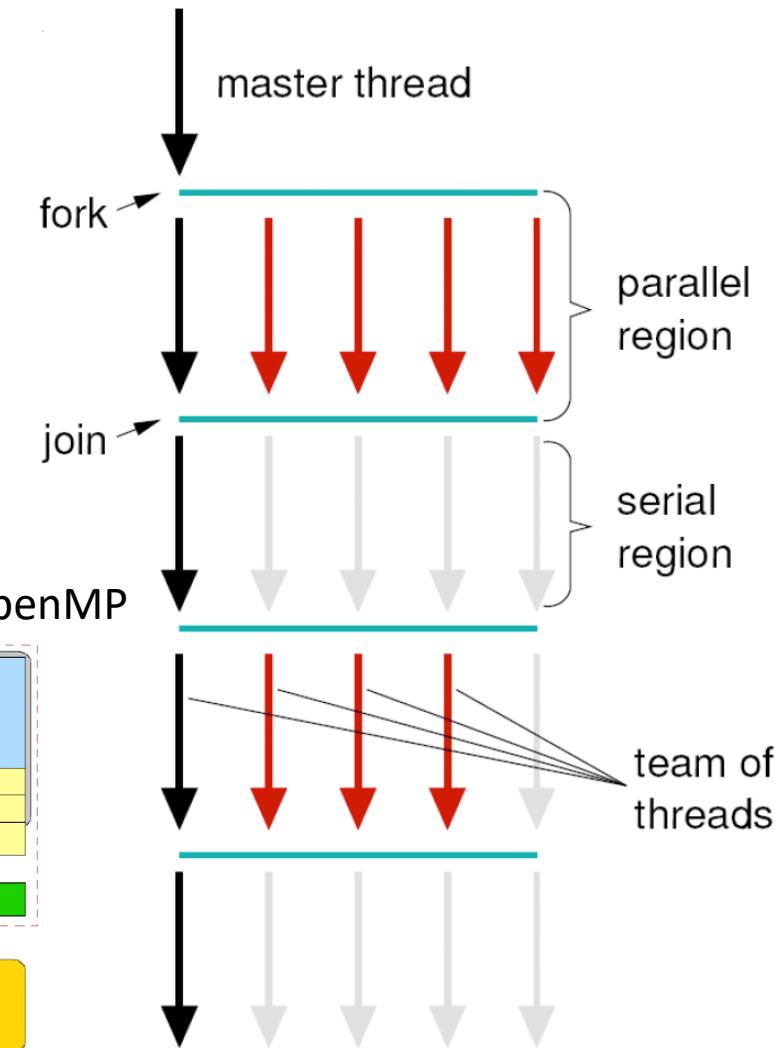
- Non-Nvidia alternative to programming GPUs.



MPI / OpenMP (OpenACC)



MPI

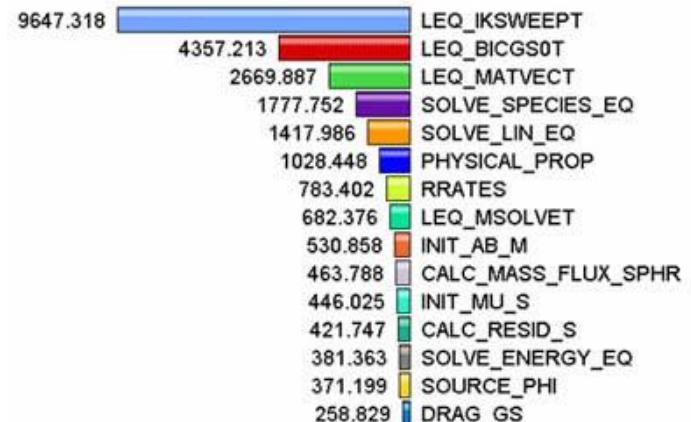


Design parallel algorithm

- We have introduced parallel hardware and software libraries, now we need to map our problem into a parallel algorithm.
- It is a (mostly) manual, time consuming, complex, error-prone and iterative process
- For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs, i.e. parallelizing compiler (loops) or pre-processor (openMP).
- However
 - Works on shared memory architectures
 - Possible wrong results
 - Less flexible than manual parallelization
 - Lower performance

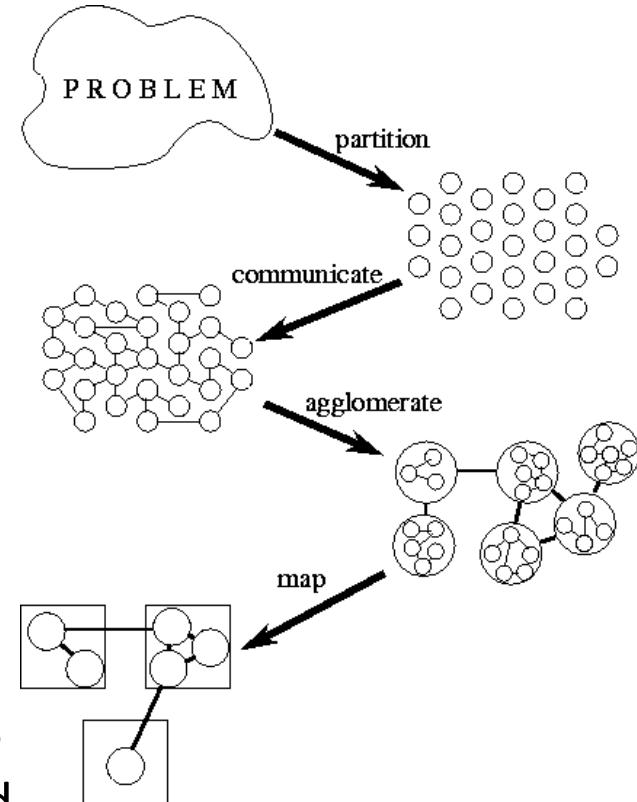
Designing parallel algorithms

- Identify the program's hotspots
 - Most programs accomplish most of their work in a few places. (profilers and performance analysis tools)
 - Focus on parallelizing the hotspots
- Identify bottlenecks in the program
 - Mainly I/O
- Identify inhibitors to parallelism
 - Data dependence, ...
- Investigate other algorithms if possible
- Then start designing the solution



PCAM Methodology

- *Partitioning.* The computation and the data are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution (i.e. concurrency and scalability).
- *Communication.* The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
- *Agglomeration.* The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
- *Mapping.* Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.



DESIGNING and BUILDING
PARALLEL PROGRAMS

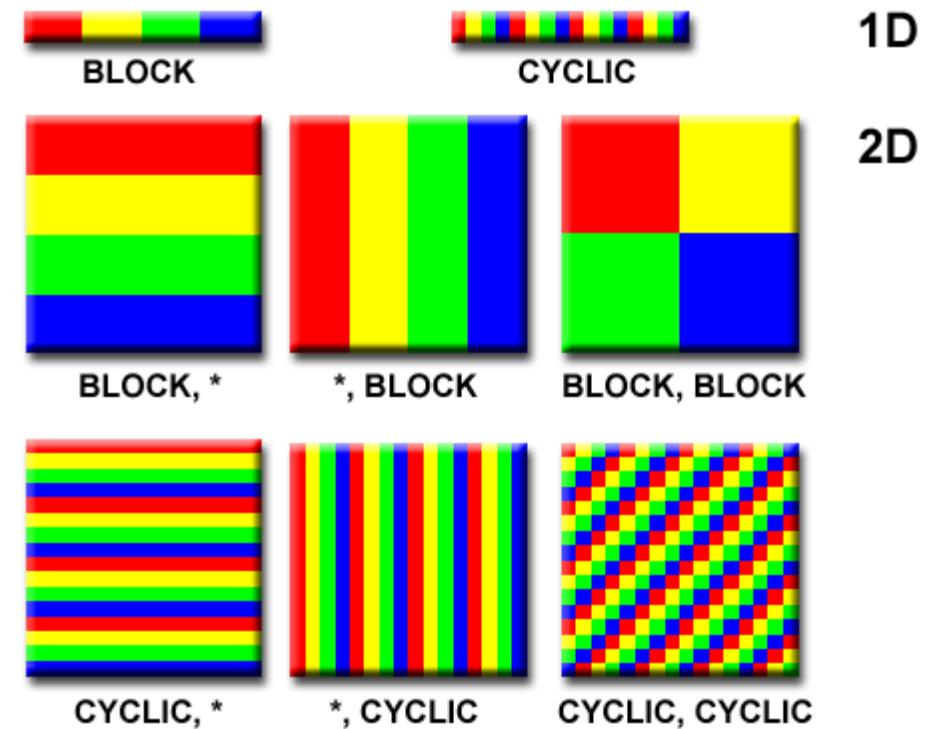
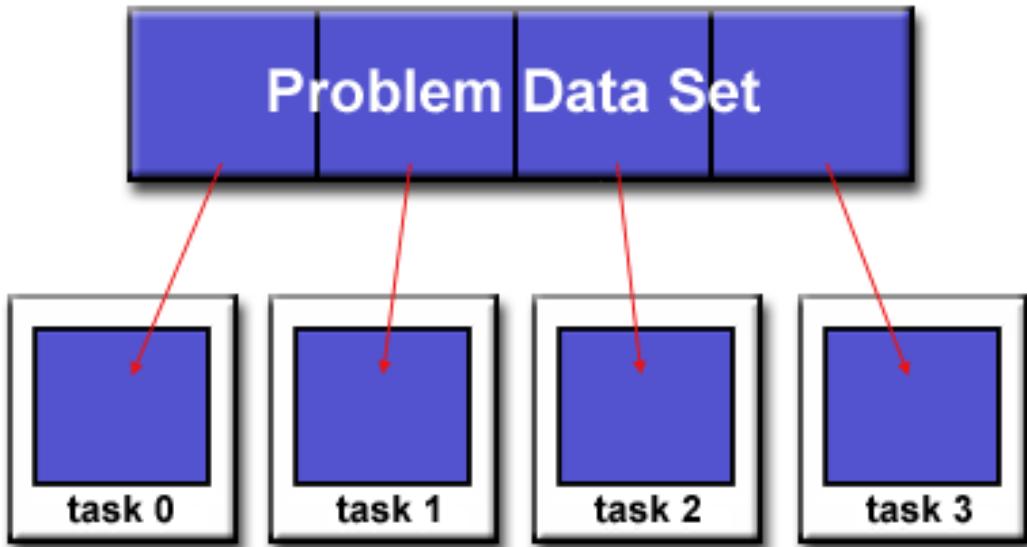
Concepts and Tools for
Parallel Software Engineering



Ian Foster

Partitioning – Domain decomposition

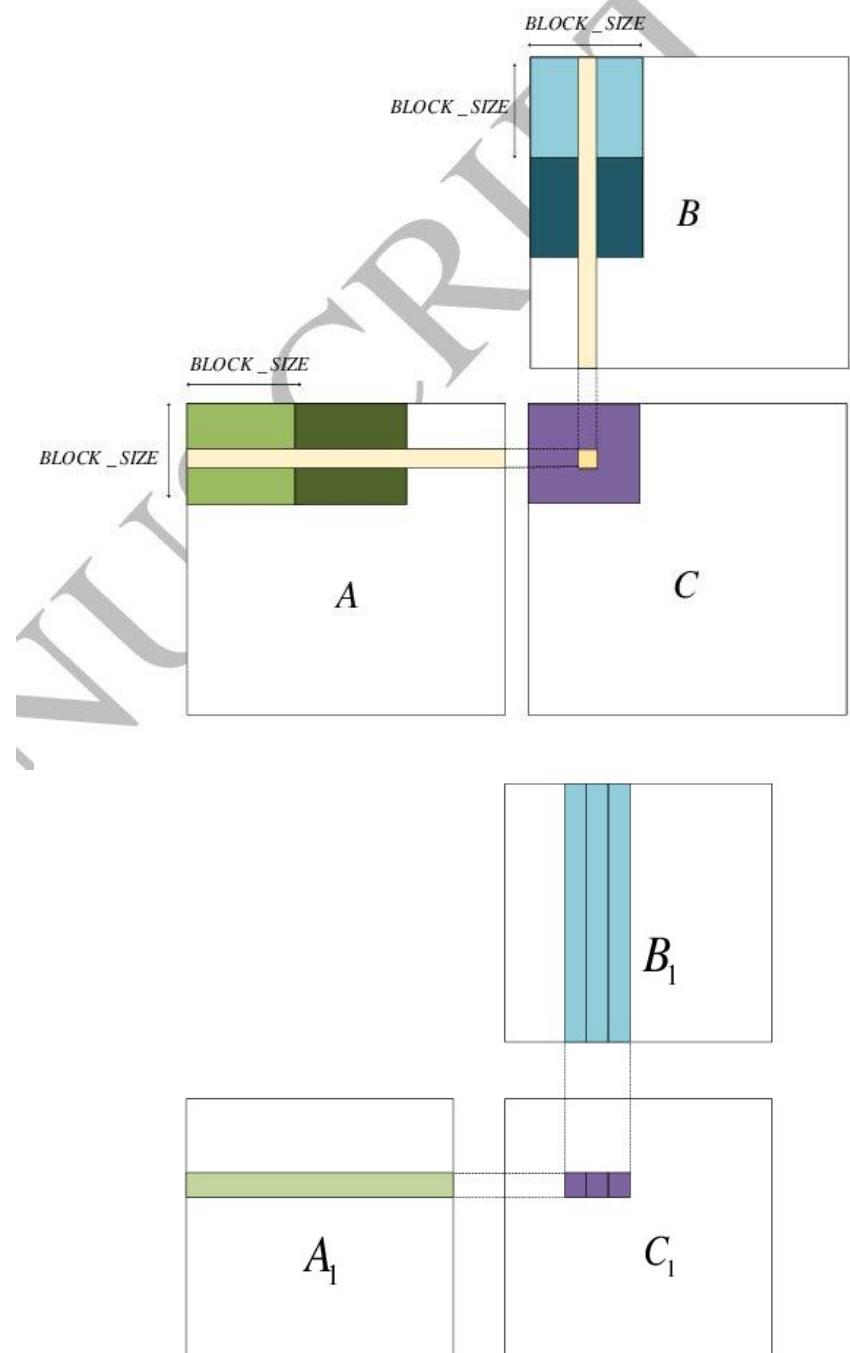
- Break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
- Domain or functional decomposition



Example: $C = AxB$

Different partitioning strategies

$$\begin{aligned}
 & \begin{pmatrix} -1 & 2 & 4 \\ 1 & 0 & -1 \\ 2 & -1 & 3 \end{pmatrix} \begin{pmatrix} -2 & 2 \\ 0 & 1 \\ -2 & -1 \end{pmatrix} \\
 &= \begin{pmatrix} (-1 & 2 & 4) \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} & (-1 & 2 & 4) \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \\ \hline (1 & 0 & -1) \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} & (1 & 0 & -1) \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \\ \hline (2 & -1 & 3) \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} & (2 & -1 & 3) \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \end{pmatrix} \\
 &= \begin{pmatrix} -6 & -4 \\ 0 & 3 \\ -10 & 0 \end{pmatrix}
 \end{aligned}$$

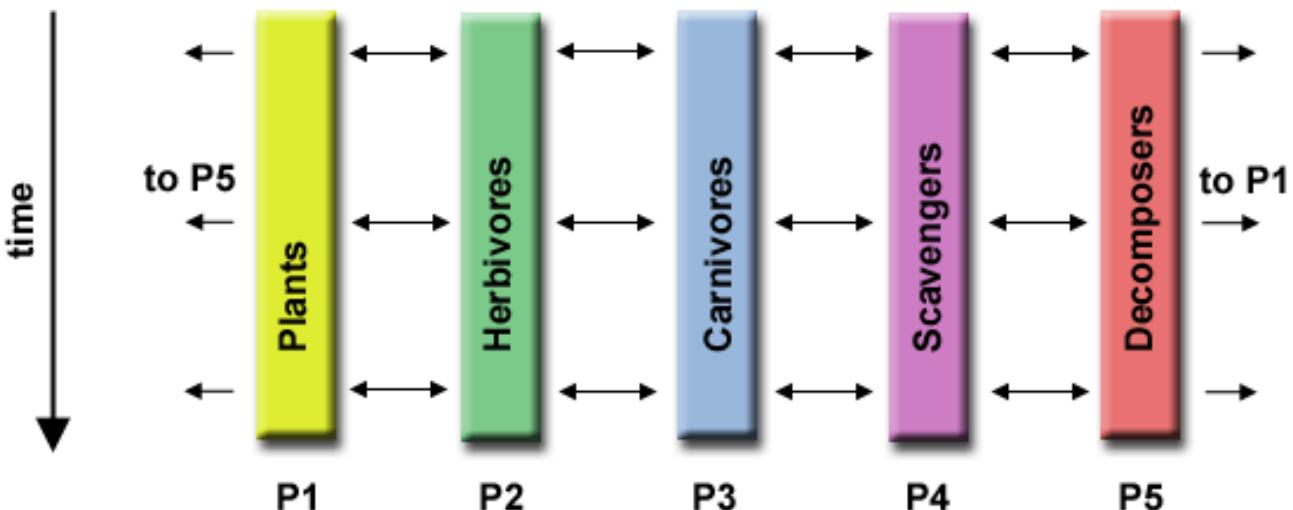
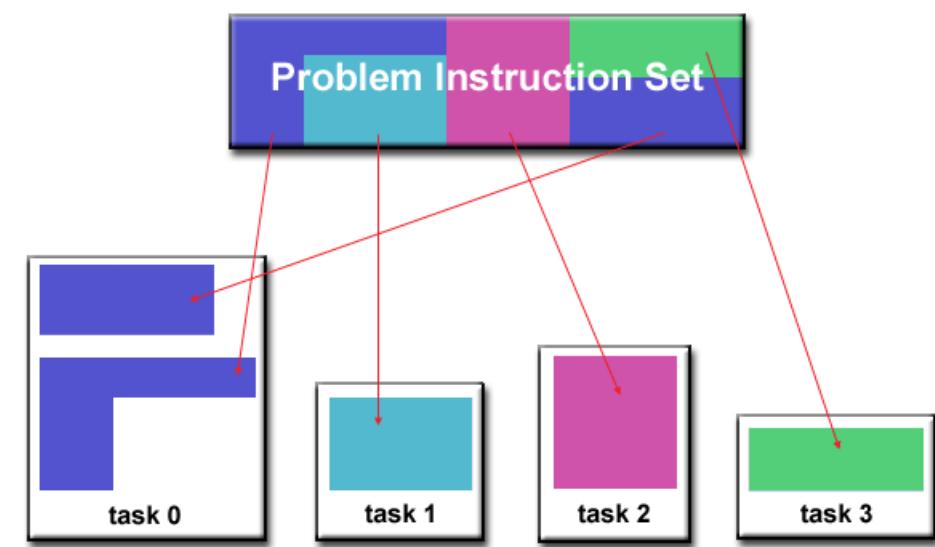


Partitioning – Functional decomposition

- The focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Example 1: Each program calculates the population of a given group, where each group's growth depends on that of its neighbors.

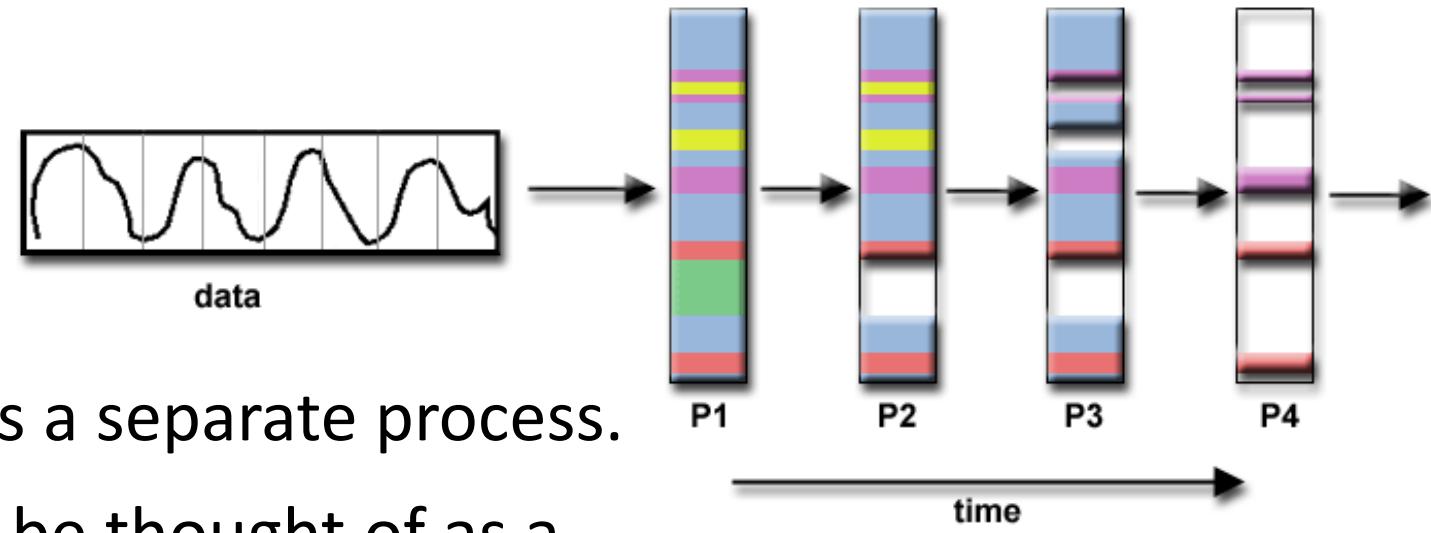
For each time stamp, each process calculates its current state, then exchanges information with the neighbor populations.

- Ex 2: nbody, with each body associated with a process.

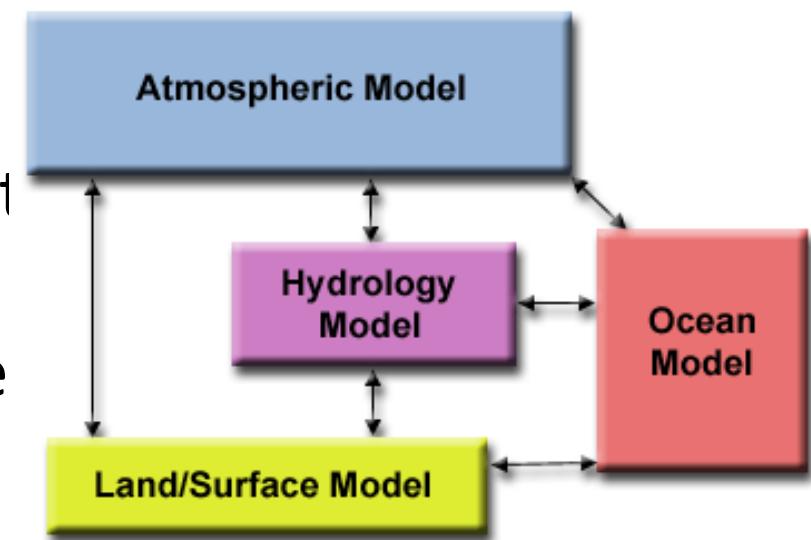


Partitioning – Functional decomposition

- Ex 3: Signal/image processing
An audio signal data set is passed through four distinct computational filters. Each filter is a separate process.



- Ex 4: Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



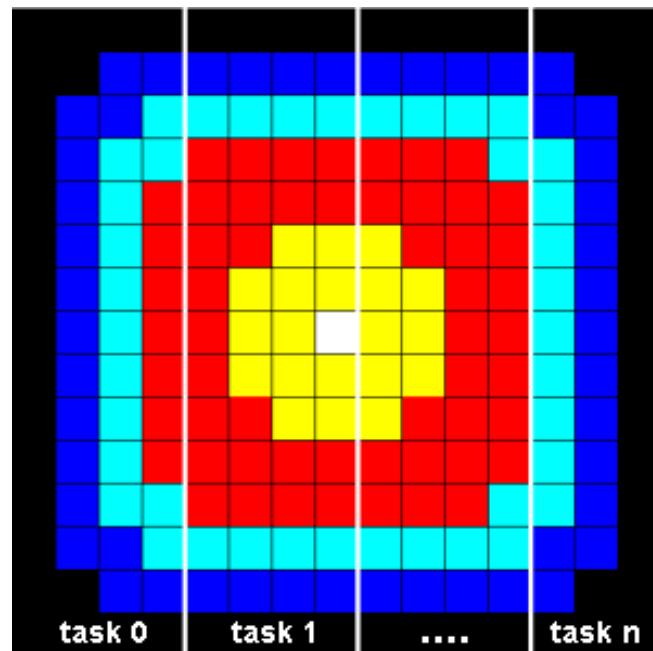
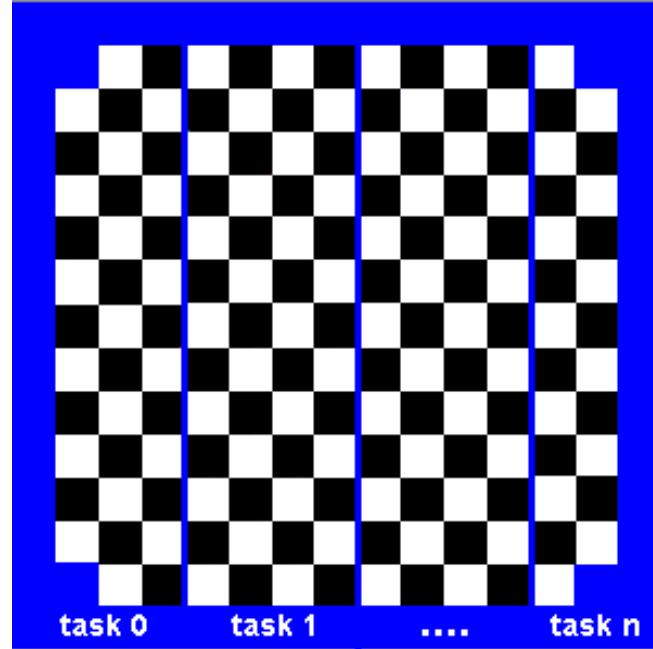
Partitioning design checklist

Before proceeding to evaluate communication requirements, you can use the following checklist to ensure that the partitioning you designed has no obvious flaws. Generally, all these questions should be answered in the affirmative.

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.
- Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
- Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
- Have you identified several alternative partitions? You should do it now. And remember to investigate both domain and functional decompositions.

Communications

- Communications between tasks depends upon the problem
- No Need for communications
 - Problems that can be decomposed and executed in parallel with virtually no need for tasks to share data.
 - Often called embarrassingly parallel because they are so straightforward. E.g. reverse the colors of a chessboard.
- Need for communication
 - Most parallel applications require tasks to share data
 - Example: a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.



What Factors to Consider?

Cost of Communications

- Inter-task communication always implies overhead
- Resources are used to package/transmit data instead of computation
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems

Latency vs. Bandwidth

- latency is the time it takes to send a minimal (0 byte) message from point A to point B.
- bandwidth is the amount of data that can be communicated per unit of time (bytes/sec).
- Sending many small messages can cause latency to dominate communication overheads.
- Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth

What Factors to Consider?

Visibility of communications

- With the Message Passing Model, communications are explicit and under the control of the programmer
- With the Data Parallel Model, communications (e.g. shared data access) often occur transparently to the programmer.

Synchronous vs. asynchronous communications

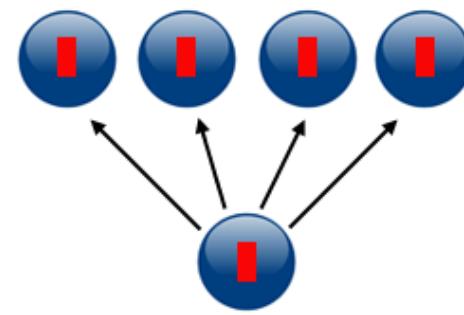
- Synchronous communications require handshaking between tasks that are sharing data.
- Synchronous communications are blocking since other work must wait until the communications have completed. E.g. phone call.
- Asynchronous communications allow tasks to transfer data independently from one another.
- Asynchronous communications are non-blocking since other work can be done while the communications are taking place. E.g. whatsapp
- Interleaving computation with communication is the single greatest benefit for using asynchronous communications

Scope of the Communication

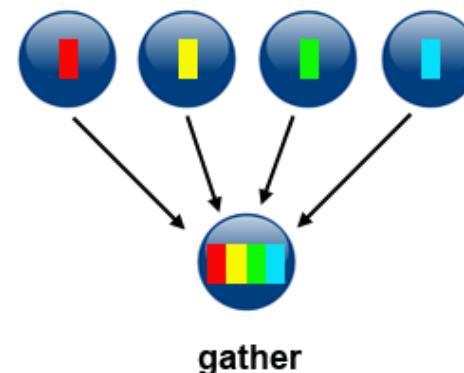
Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.

Point-to-point - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.

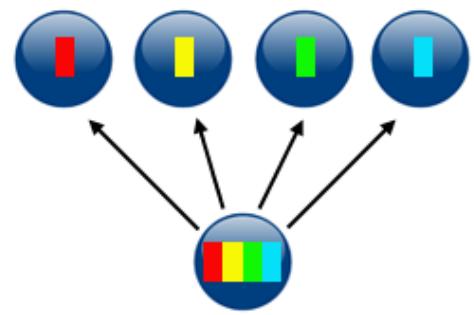
Collective - involves data sharing between more than two tasks, which are often specified as being members in a common group.



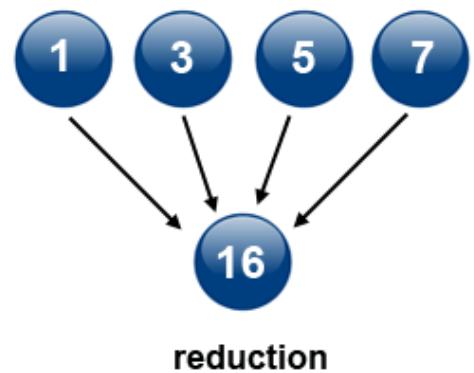
broadcast



gather



scatter



reduction

Efficiency of communications

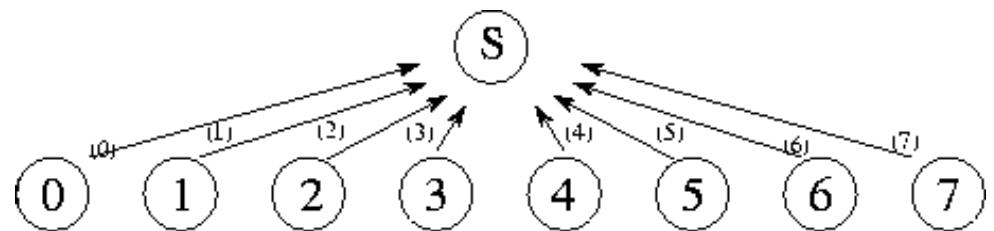
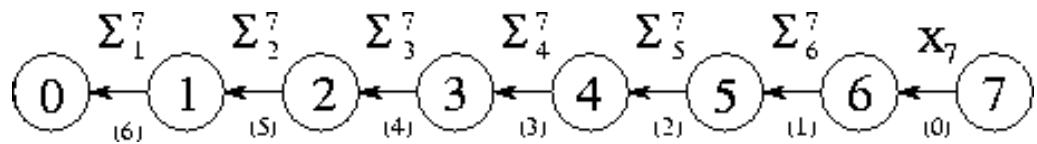
- Oftentimes, the programmer has choices that can affect communications performance. Only a few are mentioned here.
- Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
- What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
- Network fabric - different platforms use different networks. Some networks perform better than others. Choosing a platform with a faster network may be an option.

Communication design checklist

- Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggest a nonscalable construct. For example, if a frequently accessed data structure is encapsulated in a single task, consider distributing or replicating this data structure.
- Does each task communicate only with a small number of neighbors? If each task must communicate with many other tasks, evaluate the possibility of formulating this global communication in terms of a local communication structure or use optimized, collective operations (e.g. MPI_Reduce).
- Are communication operations able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Try to use divide-and-conquer techniques.
- Is the computation associated with different tasks able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Consider whether you can reorder communication and computation operations.

Example

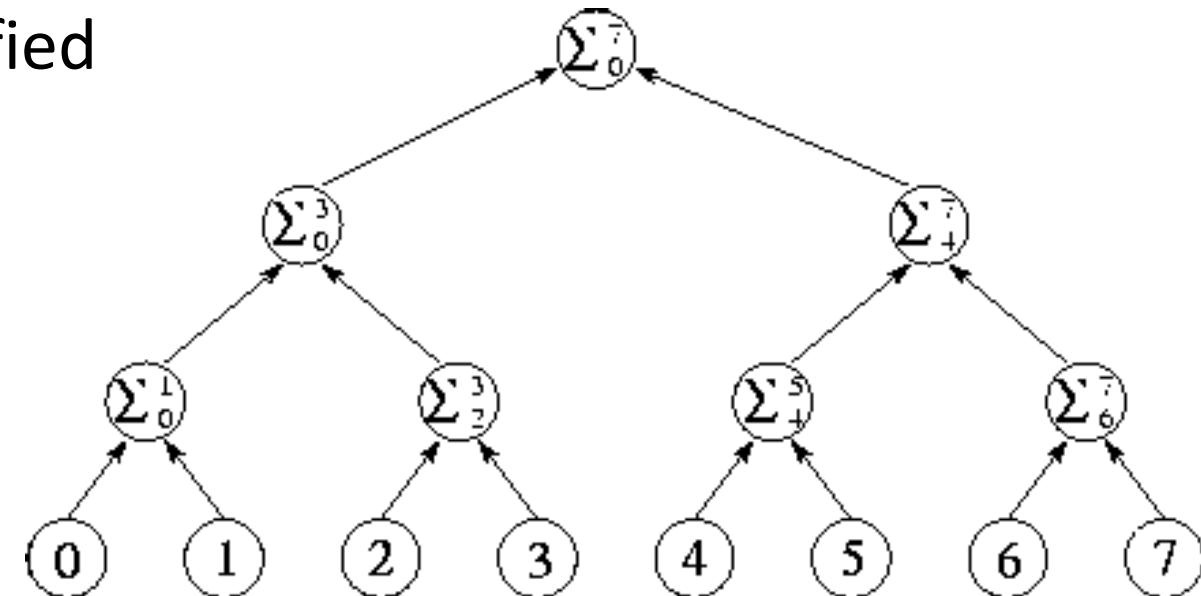
- A centralized summation algorithm that uses a central manager task (S) to sum N numbers distributed among N tasks.
- We can distribute the summation of the N numbers by making each task compute
$$S_i = X_i + S_{i-1}$$



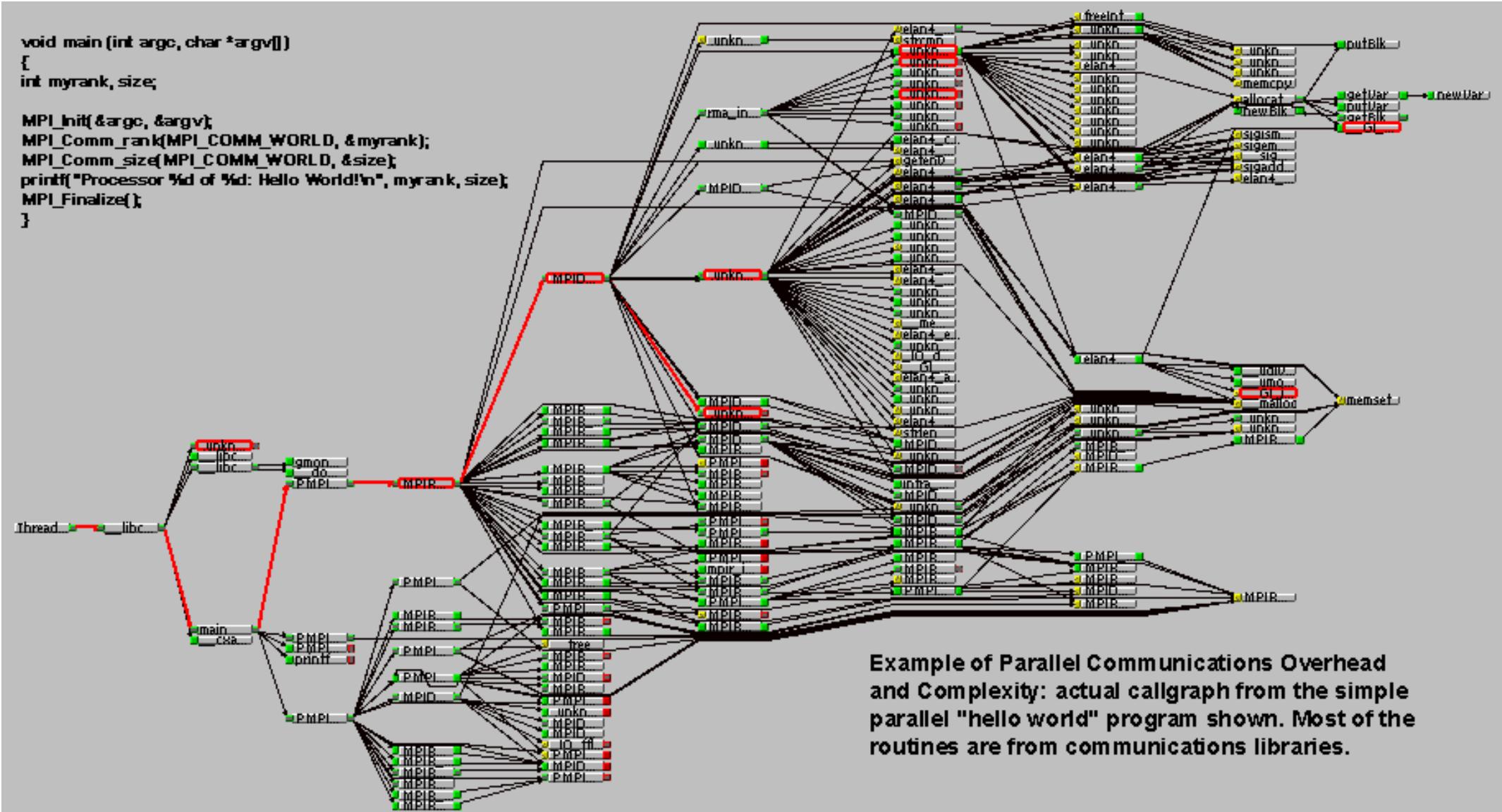
- This algorithm distributes the $N-1$ communications and additions, but permits concurrent execution only if multiple summation operations are to be performed. A single summation still takes $N-1$ steps.

Example

- The *divide and conquer* problem-solving strategy: we seek to partition recursively a complex problem into two or more simpler problems of roughly equivalent size
- we have distributed the $N-1$ communication and computation operations required to perform the summation and have modified the order in which these operations are performed so that they can proceed concurrently
 - Regular communication structure
 - Each task communicates with a small set of neighbors
 - The solution scales



Overhead and complexity



Synchronization

Barrier

- Usually implies that all tasks are involved
- Each task performs its work until it reaches the barrier, then it "blocks"
- When the last task reaches the barrier, all tasks are synchronized before the next operation

Lock/Semaphore

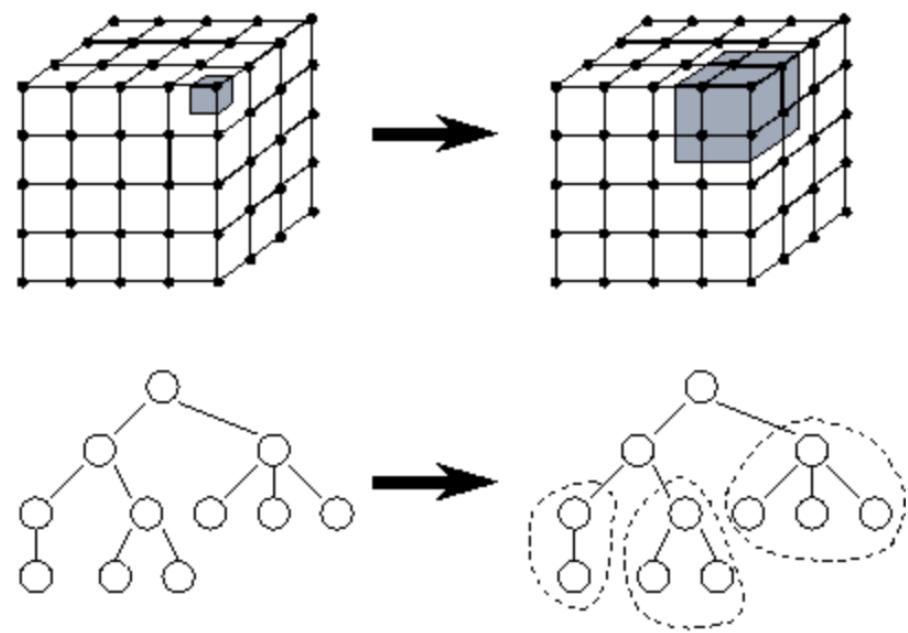
- Can involve any number of tasks
- Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use the lock/semaphore/flag
- Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.

Synchronous communication operations

- Involves only those tasks executing a communication operation
- When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication.

Agglomeration

- In the first two stages of the design process, we partitioned the computation to be performed into a set of tasks and introduced communication to exchange data required by these tasks.
- The resulting algorithm is still abstract in the sense that it is not specialized for efficient execution on any particular parallel computer.
 - it may be highly inefficient if we create many more tasks than there are processors on the target computer and this computer is not designed for efficient execution of small tasks.
- Here we move from the abstract toward the concrete
- it is useful to combine tasks in a smaller number of greater size tasks?
- It is worthwhile to *replicate* data and/or computation wrt to communications?



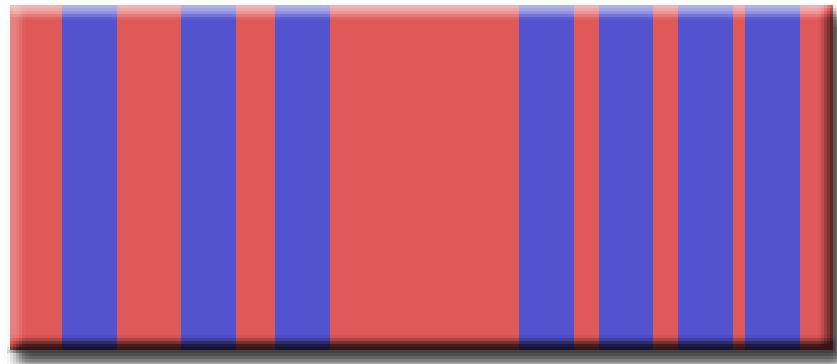
Increasing Granularity

- In the partitioning phase we focus on defining as many tasks as possible.
- But such a large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.
 - Communication costs: we have to stop computing in order to communicate
 - Task creation cost
 - Vectorization and cache exploitation
- We can sometimes trade off replicated computation for reduced communication requirements and/or execution time.

Granularity

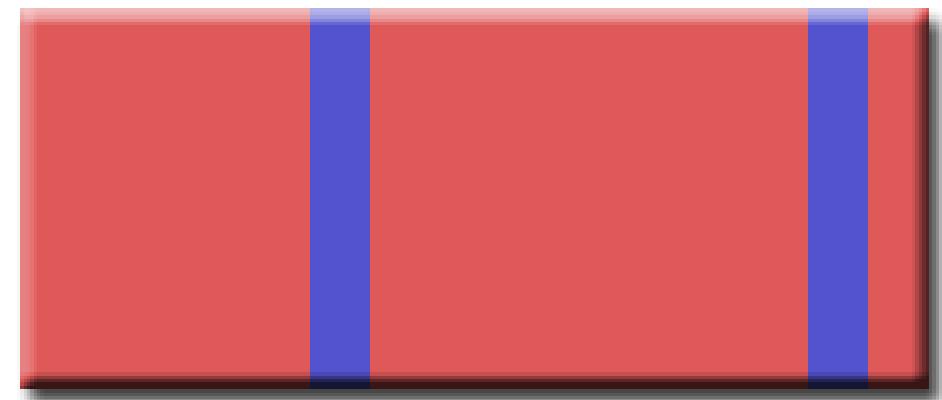
It is a qualitative measure of the ratio of computation to communication.

Fine-grain parallelism



communication
 computation

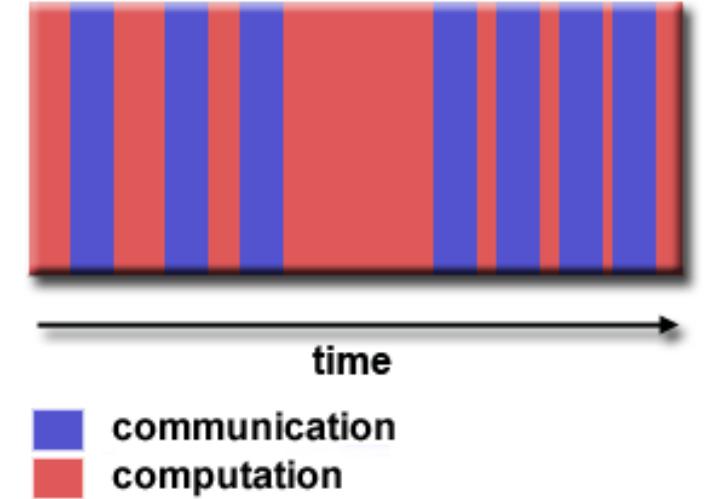
Coarse-grain parallelism



communication
 computation

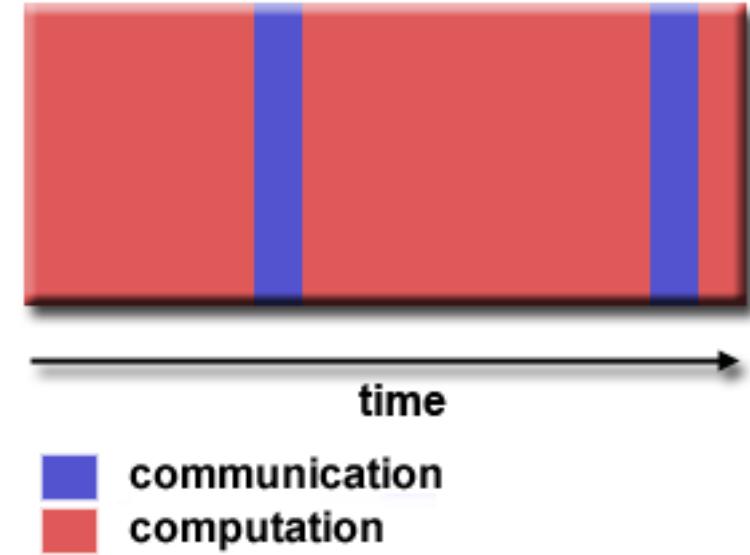
Fine-grain Parallelism

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



Coarse-grain Parallelism

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently

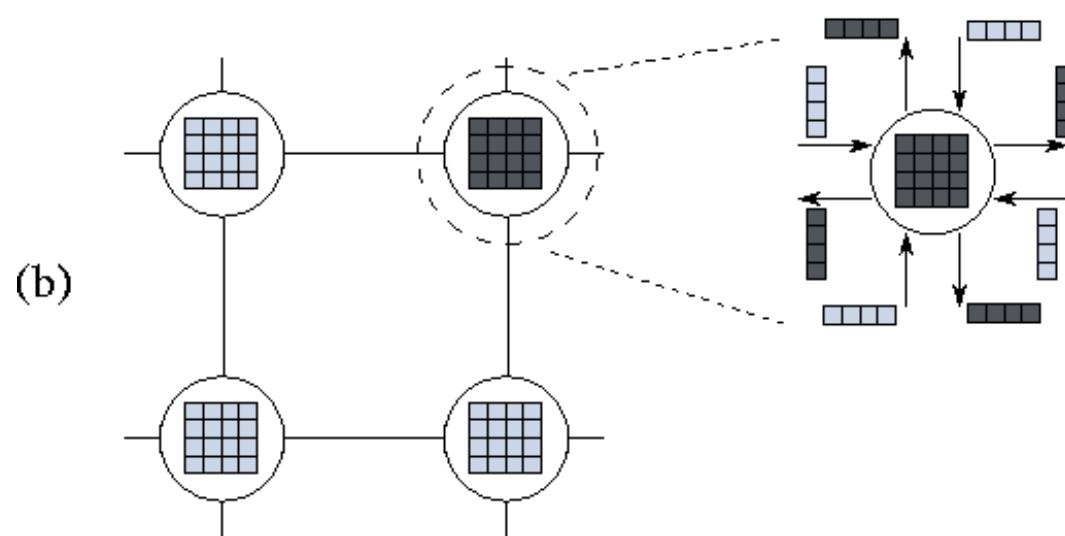
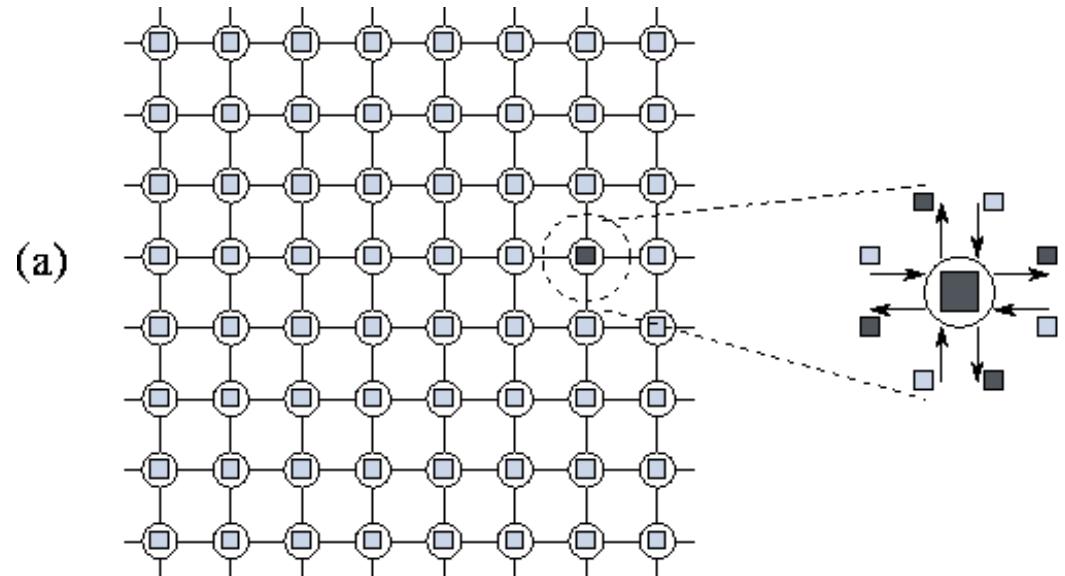


Which is Best?

- It depends on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

Examples

- 4 send/rec for updating 1 point
(5 point stencil)
- A) 64 task, about 224 communications
($36*4 + 24*3 + 4*2$)
- B) 4 tasks, 8 communications

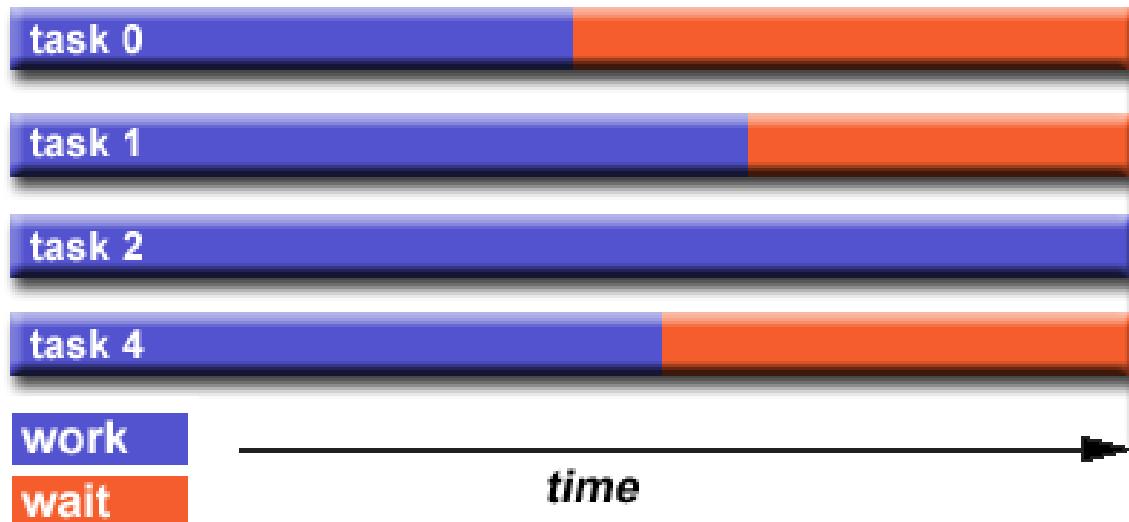


Agglomeration Design Checklist

- Has agglomeration reduced communication costs by increasing locality? If not, examine your algorithm to determine whether this could be achieved using an alternative agglomeration strategy.
- Has agglomeration yielded **tasks with similar computation and communication costs**? The larger the tasks created by agglomeration, the more important it is that they have similar costs. If we have created just one task per processor, then these tasks should have nearly identical costs.
- Does the number of tasks still scale with problem size? If not, then your algorithm is no longer able to solve larger problems on larger parallel computers.
- Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability? Other things being equal, algorithms that create fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks.

Load Balancing

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



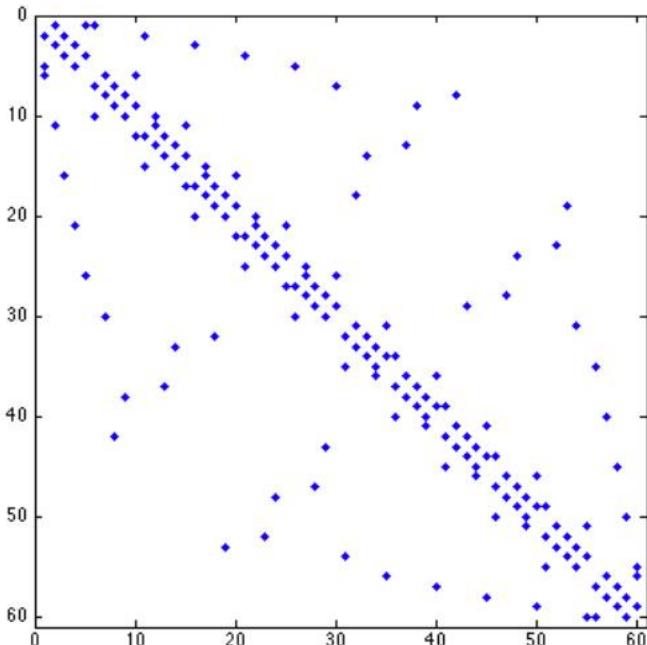
How to Achieve Load Balance

Equally partition the work each task receives

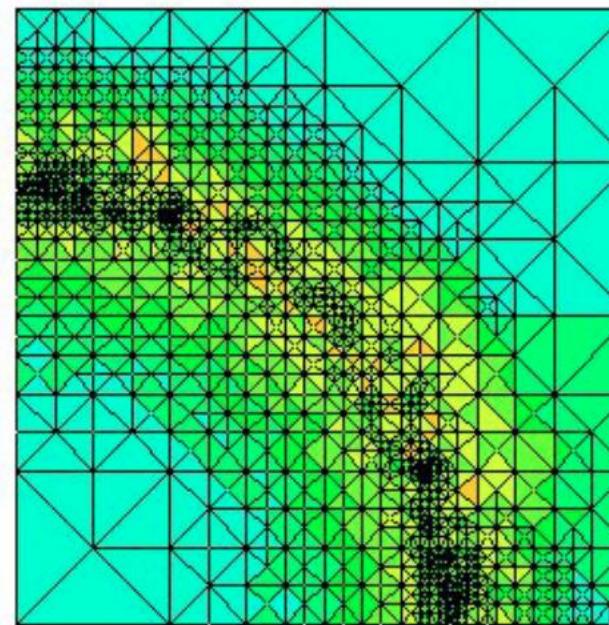
- For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
- For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
- If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool.
 - For example you can perform benchmarks to weight the machines and assign the workload accordingly. It is a static (pre-defined) load balancing strategy.

How to Achieve Load Balance

Certain classes of problems result in load imbalances even if data is evenly distributed among tasks



Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".

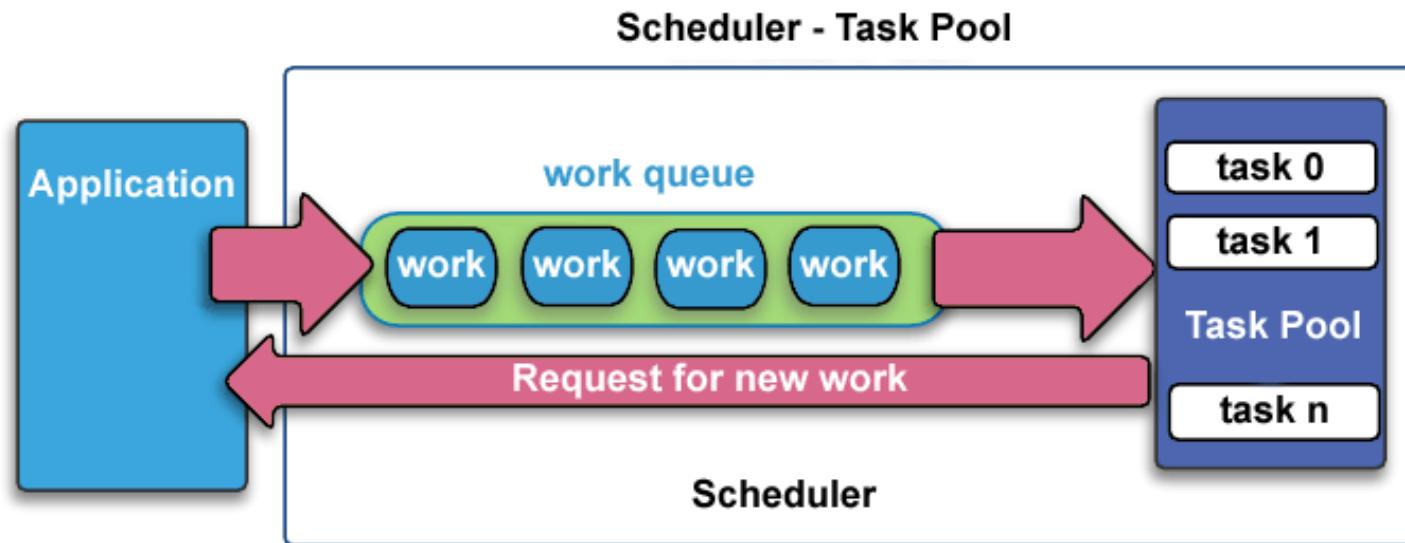


Adaptive grid methods - some tasks may need to refine their mesh while others don't.

Use Dynamic Work Assignment

When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler - task pool approach. As each task finishes its work, it queues to get a new piece of work.

- It is the **master-slave/worker paradigm** where the scheduler is a master process (not necessarily process 0)



Mapping

- In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute.
- Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:
 - We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
 - We place tasks that communicate frequently on the *same* processor, so as to increase locality.
- Two important aspects
 - which cores our threads/processes are running on (affinity)
 - where memory has been allocated (NUMA effects)

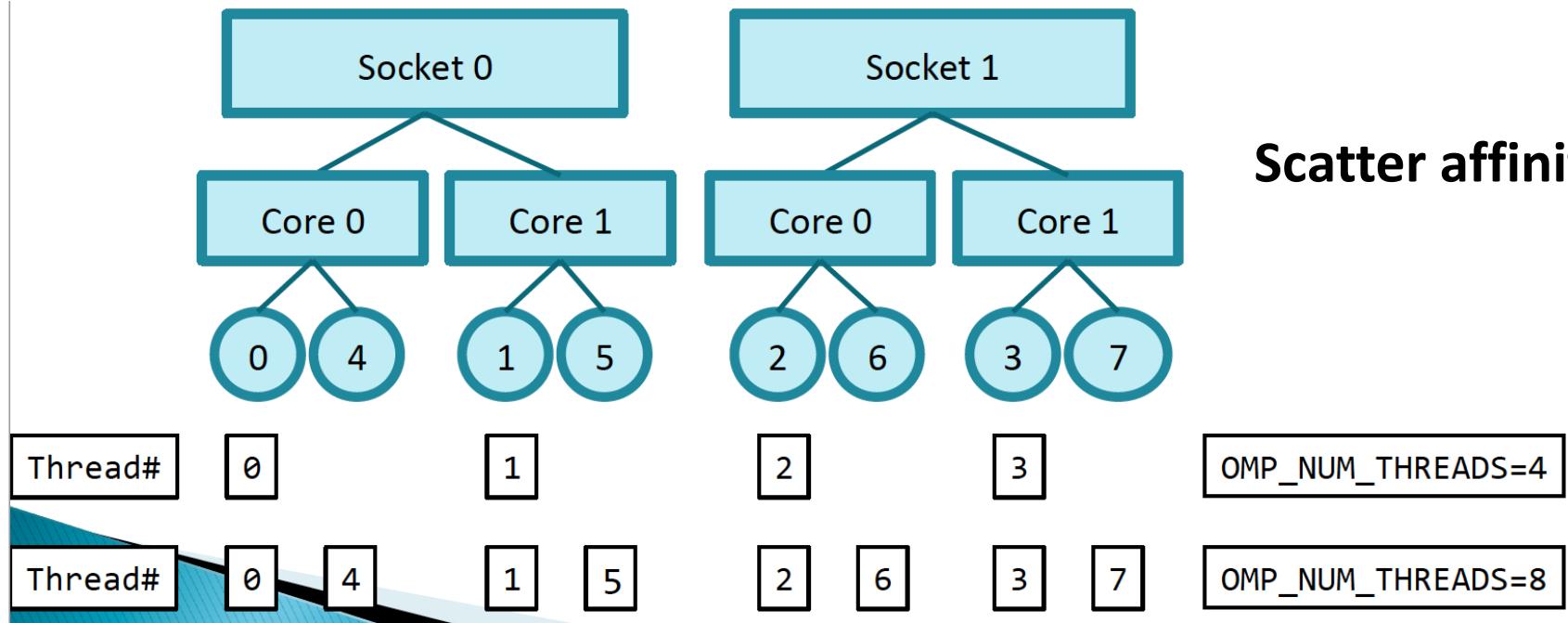
NUMA

- ▶ In an OpenMP application running on a node, the threads running on any socket see one unified memory space, and can read and write to memory that is local to other sockets/dies.
 - The memory is shared between the different sockets on a node.
 - The time taken to access memory on a different socket (non-local access) is slower than time taken to access local memory.
- ▶ This memory architecture is called **Non-Uniform Memory Access (NUMA)**
- ▶ “**NUMA effects**” arise when threads excessively access memory on a different NUMA domain.
- ▶ Numa effects are to be avoided!

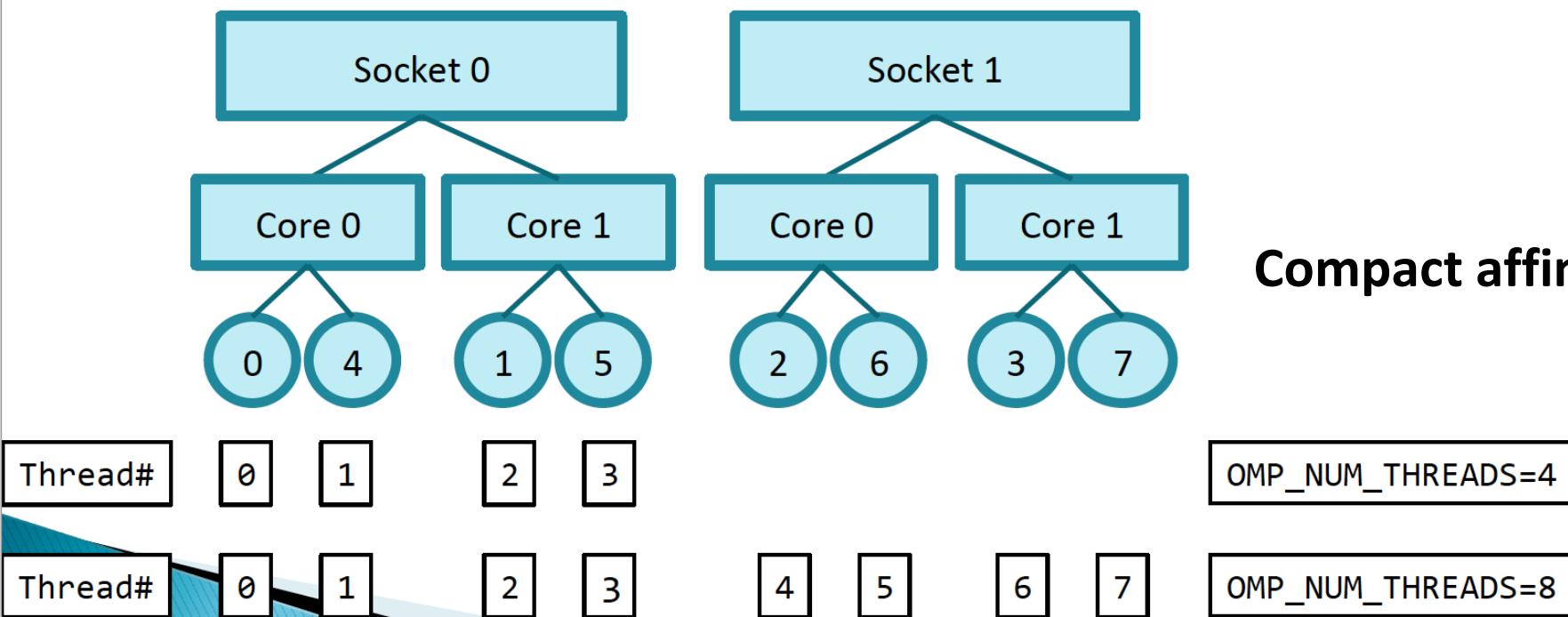
Affinity

- ▶ The concept of **affinity** is important to reduce NUMA-effects.
- ▶ **CPU affinity** is the pinning of a process or thread to a particular core
 - If the operating system interrupts the task, it doesn't migrate it to another core, but waits until the core is free again
 - For most HPC scenarios where only one application is running on a node, these interruptions are short
- ▶ **Memory affinity** is the allocation of memory as close as possible to the core on which the task that requested the memory is running
- ▶ Both CPU affinity and memory affinity are important if we are to maximise memory bandwidth on NUMA nodes
 - If memory affinity is not enabled then bandwidth will be reduced as we go off-socket to access remote memory
 - If CPU affinity is not enabled then allocating memory locally is of no use when the task that requested the memory might no longer be running on the same socket

Scatter affinity



Compact affinity



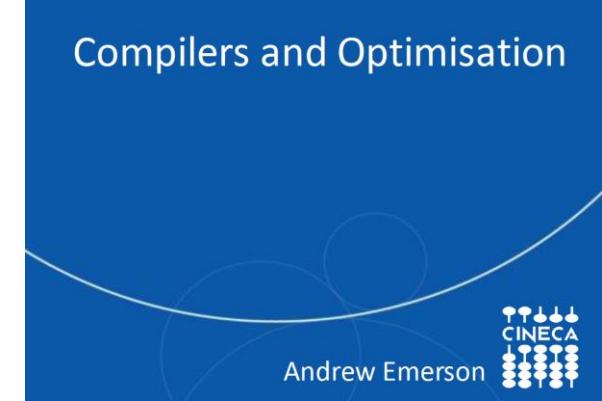
Parallel examples

- See here
https://computing.llnl.gov/tutorials/parallel_comp/#Examples
- My PhD Thesis – see the paper on Aulaweb

Vectorization

And compilers

Introduction



- Perhaps, together with the operating system, the compiler is THE most important HPC software tool.
- Compilers are very sophisticated software tools but cannot replace human understanding of what the code should do.
 - They can attempt to optimise the code.
- There are many compilers available and for all computer operating systems, both free and commercial
- The Intel Compiler
<https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference>

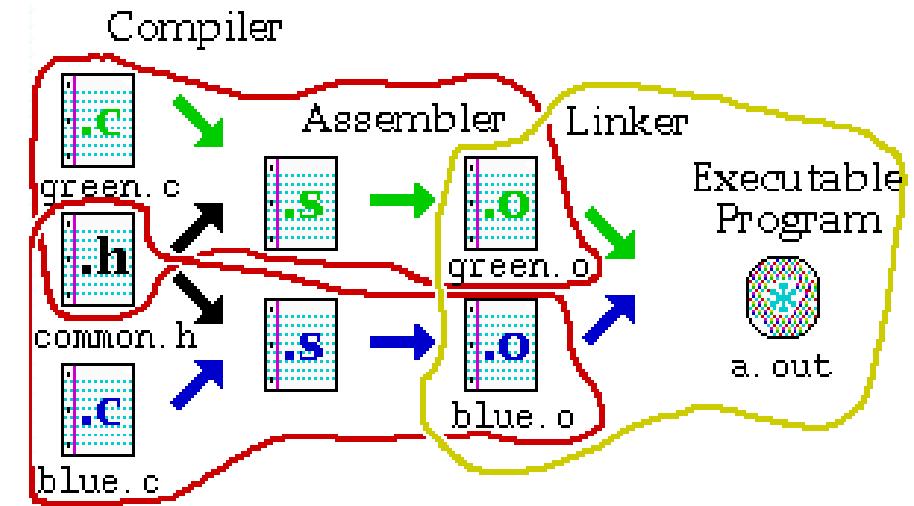
Compilers and interpreters

- Interpreted languages
 - The code is “translated” statement-by-statement during the execution
 - Easier on the programmer, modifications can be made quickly but optimisations between different statements (almost) impossible
 - Used for scripting languages (bash, Perl, PHP, ..)
- Compiled languages
 - Entire program is translated before execution
 - Optimizations between different parts of the program possible.
 - HPC languages include FORTRAN, C and C++
 - Python? https://sea.ucar.edu/sites/default/files/python_hpc.pdf
http://www.training.prace-ri.eu/uploads/tx_pracetmo/pythonHPC.pdf

Compilers stages

“Compiling” a program is actually a three stage process:

1. Pre-processing to replace MACROs (**#define**), code insertions (**#include**), code selections (**#ifdef**, **#if**).
 - <https://www.howtoforge.com/linux-nm-command/>
 - Code optimisations are mainly done during compilation.
2. Compilation of the source code into object files – organised collections of symbols referring to variables and functions.
 - <https://www.howtoforge.com/linux-nm-command/>
 - Code optimisations are mainly done during compilation.
3. Linking of the object files, together with any external libraries to create the executable (if all referred objects are resolved).
 - For large projects usual to separate the compiling and linking phases.



Common compiler options –icc

Functionality	Linux	Functionality	Linux *
Disable optimization	-O0	Optimize for current machine	-xHOST
Optimize for speed (no code size increase)	-O1	Generate SSE v1 code	-xSSE1
Optimize for speed (default)	-O2	Generate SSE v2 code (default, may also emit SSE v1 code)	-xSSE2
High-level optimizer (e.g. loop unroll)	-O3	Generate SSE v3 code (may also emit SSE v1 and v2 code)	-xSSE3
Aggressive optimizations (e.g. -ipo, -O3, ...)	-fast	Generate SSE v3 code for Atom-based processors	-xSSE_ATOM
Create symbols for debugging	-g	Generate SSSE v3 code (may also emit SSE v1, v2, and v3 code)	-xSSSE3
Generate assembly files	-S	Generate SSE4.1 code (may also emit (S)SSEE v1, v2, and v3 code)	-xSSE4.1
Optimization report generation	-opt-report	Generate SSE4.2 code (may also emit (S)SSE v1, v2, v3, and v4 code)	-xSSE4.2
OpenMP support	-openmp	Generate AVX code	-xAVX

* For Intel processors use -x, for non-Intel processors use -m

<https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/alphabetical-list-of-compiler-options.html>

Source code and performance

- <https://www.intel.com/content/www/us/en/search.html?ws=text#q=code%20sample&t=Developers&sort=relevancy>
- <https://github.com/intel-system-studio/samples>
Do in your home git clone https://github.com/intel-system-studio/samples.git
- Local copy of files /opt/intel/composer_xe_2013.5.192/composer_xe_2013.5.192/Samples/en_US/C++ ,
/opt/intel/samples_2019/en/ /opt/intel/samples_2017/en/ ...

Example

- <https://software.intel.com/en-us/advisor-tutorial-vectorization-linux-cplusplus-simd-parallelism-use-case>
- `icc -std=c99 -O0 -D NOUNCCALL Multiply.c Driver.c -o MatVector` (*do not cut&paste the commands, rewrite them*)
- Let's try also O1 and O3

O0
ROW:101 COL: 101
Execution time is 331.700 seconds
GigaFlops = 0.061507
Sum of result = 195853.999899

O1
ROW:101 COL: 101
Execution time is 49.423 seconds
GigaFlops = 0.412806
Sum of result = 195853.999899

O2/O3
ROW:101 COL: 101
Execution time is 12.435 seconds
GigaFlops = 1.640643
Sum of result = 195853.999899

Compiler optimization

- Compilers give the possibility of specifying optimisation options at compile time, together with the other options.
 - -O0 : no optimisation, the code is translated literally
 - -O1, -O2: local optimisations, compromise between compilation speed, optimisation, code accuracy and executable size (in icc O2 is default, see icc --help)
 - -O3: high optimisation, can alter the semantics of the program (hence not used for debugging)
 - -O4 or higher: Aggressive optimisations, depending on hardware.
- These are either general optimisation levels or specific flags related to the underlying hardware.
- Compilers are generally conservative, i.e. will not optimise if strong risk of obtaining incorrect results *unless* forced to by the user either with compiler directives or options.

Pay attention

Increasing the optimisation level doesn't always increase performance. Must check each time!

- Example: matrix-matrix multiplication (1024x1024), double precision, FORTRAN.
- Two systems:
 - FERMI: (IBM BG/Q Power A2, 1.6Ghz)
 - PLX: (Xeon Westmere CPUs, 2.4 Ghz)

The compiler recognises the matrix-matrix product and substitutes the code with a call to a library routine `__xl_dgemm`, but it is quite slow. Intel on PLX uses a similar strategy, but uses instead the efficient MKL library

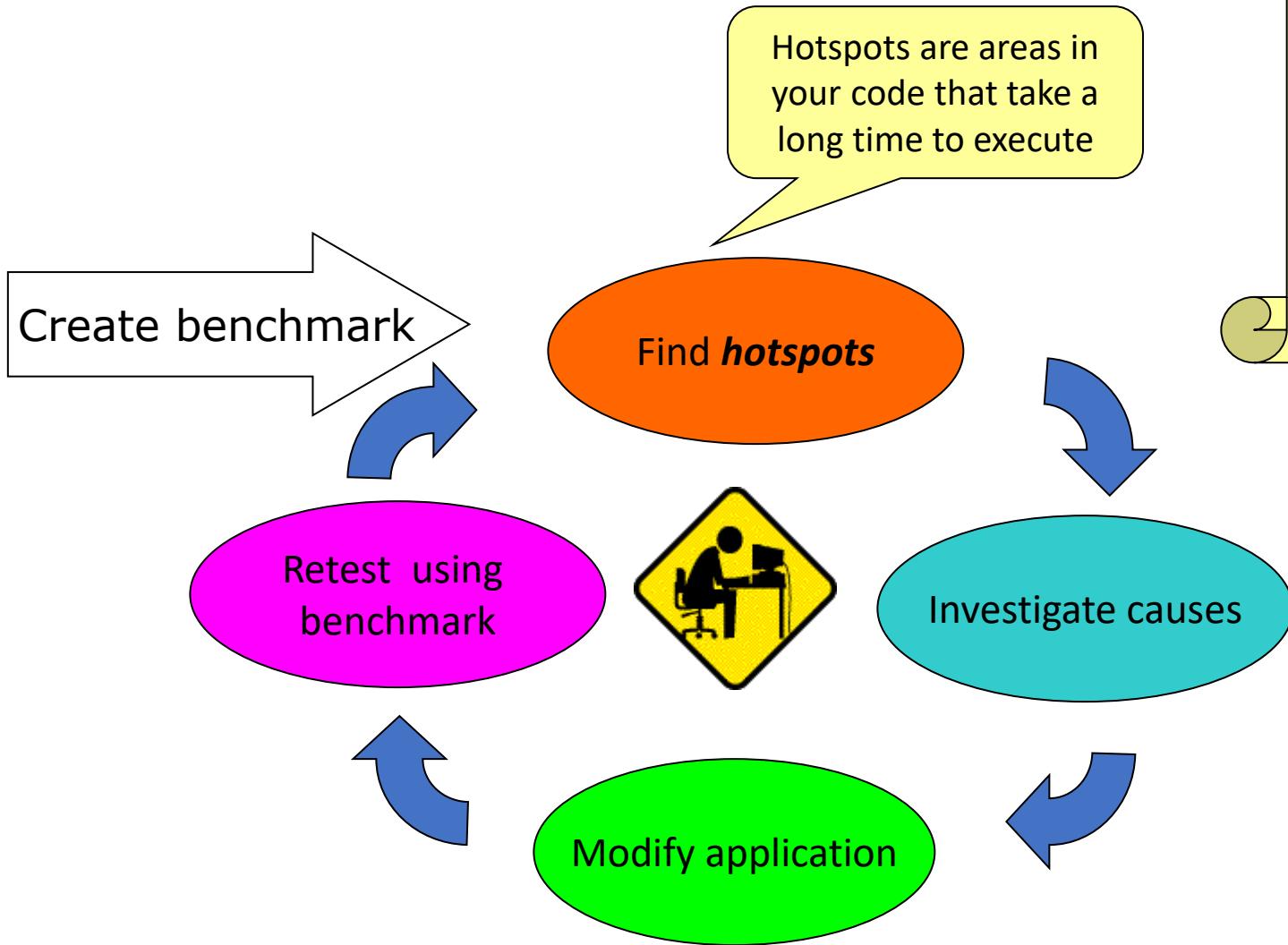
FERMI xlf

Option	Seconds	MFlops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX -ifort

Option	Seconds	MFlops
-O0	8.94	240
-O2	1.41	1514
-O3	0.72	2955
-O4	0.33	6392
-O5	0.32	6623

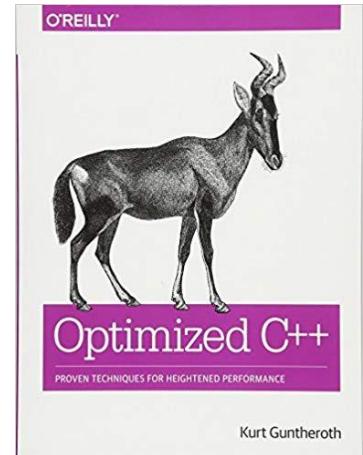
The Software Optimization Process



Software optimization does not begin where coding ends. It is an ongoing process that starts at design stage and continues all the way through development.

The Software Optimization Facts

- It's easier to optimize a slow correct program than to debug a fast incorrect one
 - *Nobody cares how fast you can compute a wrong answer...*
- Programs typically spend 80% of their time in 20% of the code
 - Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application
 - *Know when to stop!*
- Don't optimize what does not matter
 - *Make the common case fast!*



1.
Optimizing software in C++
An optimization guide for Windows, Linux, and Mac
platforms

By Agner Fog. Technical University of Denmark.
Copyright © 2004 - 2019. Last updated 2019-09-14.

A first optimization: the Vectorization

- From Intel: it is the unrolling of a loop combined with the generation of packed SIMD instructions by the compiler.
- Because the packed instructions operate on more than one data element at a time, the loop can execute more efficiently.
- It is sometimes referred to as auto-vectorization, to emphasize that the compiler identifies and optimizes suitable loops on its own, without requiring any special action by the programmer.

<https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf>

<https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-programming-guidelines-for-vectorization>

<https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf>

<https://software.intel.com/content/dam/develop/external/us/en/documents/dpd-vectorization-codebook-558870.pdf>

Recap

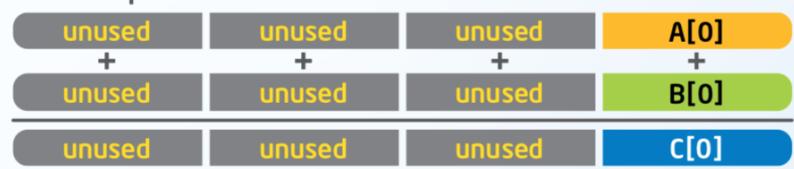
Scalar version works on one element at a time

$$a[i] = b[i] + c[i] \times d[i];$$

Vector version carries out the same instructions on many elements at a time

$$a[i:8] = b[i:8] + c[i:8] * d[i:8];$$

Scalar Implementation



```
for (i=0; i<N; i++) {  
    a[i]=b[i]+c[i];  
}
```



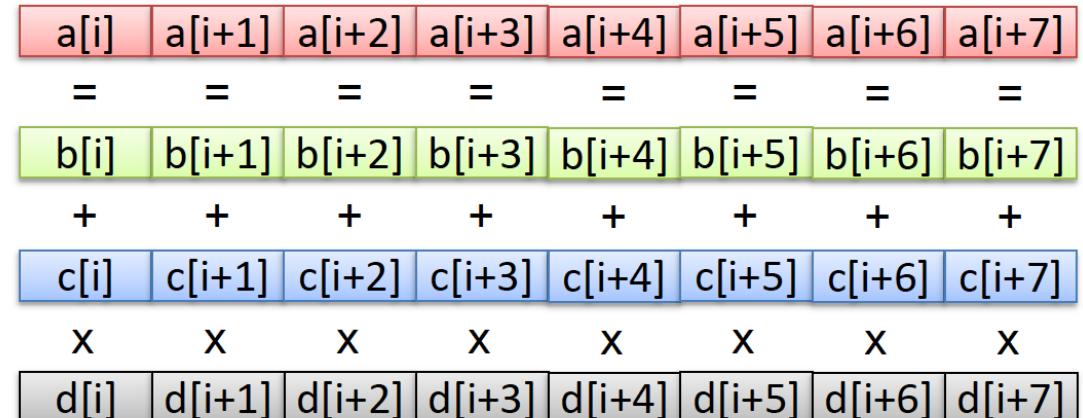
```
for (i=0; i<N; i+=4) {  
    a[i+0]=b[i+0]+c[i+0];  
    a[i+1]=b[i+1]+c[i+1];  
    a[i+2]=b[i+2]+c[i+2];  
    a[i+3]=b[i+3]+c[i+3];  
}
```

$$\begin{array}{l} a[i] \\ = \\ b[i] \\ + \\ c[i] \\ \times \\ d[i] \end{array}$$

Load b(i..i+3)
Load c(i..i+3)
Operate b+c->a
Store a



Made by the compiler – avoid to unroll loops by yourself



4 assembly ops wrt 4*4

Loop unrolling

- Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as space–time tradeoff. The transformation can be undertaken manually by the programmer or by an optimizing compiler.
- The goal of loop unwinding is to increase a program's speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration; reducing branch penalties; as well as hiding latencies, including the delay in reading data from memory. To eliminate this computational overhead, loops can be re-written as a repeated sequence of similar independent statements.

https://en.wikipedia.org/wiki/Loop_unrolling

An example using RISC-V assembler

The DAXPY cycle $Y = a \cdot X + Y$, where X and Y are vectors ($Y[i] = a \cdot X[i] + Y[i]$)

If AVX512 and X and Y have 8 elements, 66 ops wrt 6, speedup $11 > 8$!

Conventional processor

```
fld f0, a(x3)          // load scalar a
addi x5, x19, 512      // end of array X
loop: fld f1, 0(x19)   // load x[i]
fmul.d f1, f1, f0      // a * x[i]
fld f2, 0(x20)         // load y[i]
fadd.d f2, f2, f1      // a * x[i] + y[i]
fsd f2, 0(x20)         // store y[i]
addi x19, x19, 8        // increment index to x
addi x20, x20, 8        // increment index to y
bltu x19, x5, loop    // repeat if not done
```

Vectorised version

```
fld f0, a(x3)          // load scalar a
fld.v v0, 0(x19)       // load vector x
fmul.d.vs v0, v0, f0   // a * x
fld.v v1, 0(x20)       // load vector y
fadd.d.v v1, v1, v0    // vector-vector add
fsd.v v1, 0(x20)       // store vector y
```

How to achieve this goal?

```
__m128i add4(__m128i a, __m128i b) {  
    return _mm_add_epi32(a, b);  
}
```

2

- Let the compiler vectorize automatically is THE best choice (at least O2)
- But alternative strategies are

1. Write sections of a program in assembly language
<https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C>

2. Use higher level vector intrinsics

<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>

<https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/details-about-intrinsics.html>

https://chryswoods.com/vector_c++/immintrin.html (an example)

You obviously can do nothing if you have time and resources to waste...

```
#include <stdio.h>  
  
int main() {  
    /* Add 10 and 20 and store result into register %eax */  
    __asm__ ( "movl $10, %eax;"  
              "movl $20, %ebx;"  
              "addl %ebx, %eax;"  
    );  
  
    /* Subtract 20 from 10 and store result into register %eax */  
    __asm__ ( "movl $10, %eax;"  
              "movl $20, %ebx;"  
              "subl %ebx, %eax;"  
    );  
  
    /* Multiply 10 and 20 and store result into register %eax */  
    __asm__ ( "movl $10, %eax;"  
              "movl $20, %ebx;"  
              "imull %ebx, %eax;"  
    );  
    return 0 ;  
}
```

1

Alignment and Interprocedural optimizations

- The vectorizer can generate faster code when operating on aligned data. For example if you align the arrays a, b, and x in Driver.c on a 16-byte boundary the vectorizer can use aligned load instructions for all arrays rather than the slower unaligned load instructions and can avoid runtime tests of alignment.
- The compiler may be able to perform additional optimizations if it is able to **optimize across source line** boundaries. These may include function inlining.

```
icc -std=c99 -D NOFUNCCALL -D ALIGNED -ipo -O3 Multiply.c Driver.c -o MatVector
```

Execution time is 12.775 seconds - GigaFlops = 1.597076 (so , little or no improvements in this case)

The fastest execution? NO! Add **-xHost** to use the highest Instruction set available in every CPU (here AVX512). But the executable becomes less portable (only Linux systems with AVX512 cpus)

Execution time is 5.798 seconds - GigaFlops = 3.519013

Pay attention

-march= tells the compiler to generate code for processors that support certain features.

-x Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.

Options -x and -m are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

On Linux systems, if neither -x nor -m is specified, the default target architecture supports Intel(R) SSE2 instructions.

In case of AVX512 on a processor different from KNL it could be necessary to specify, instead, -xCORE-AVX512 | -xCOMMON-AVX512 AND -qopt-zmm-usage=high

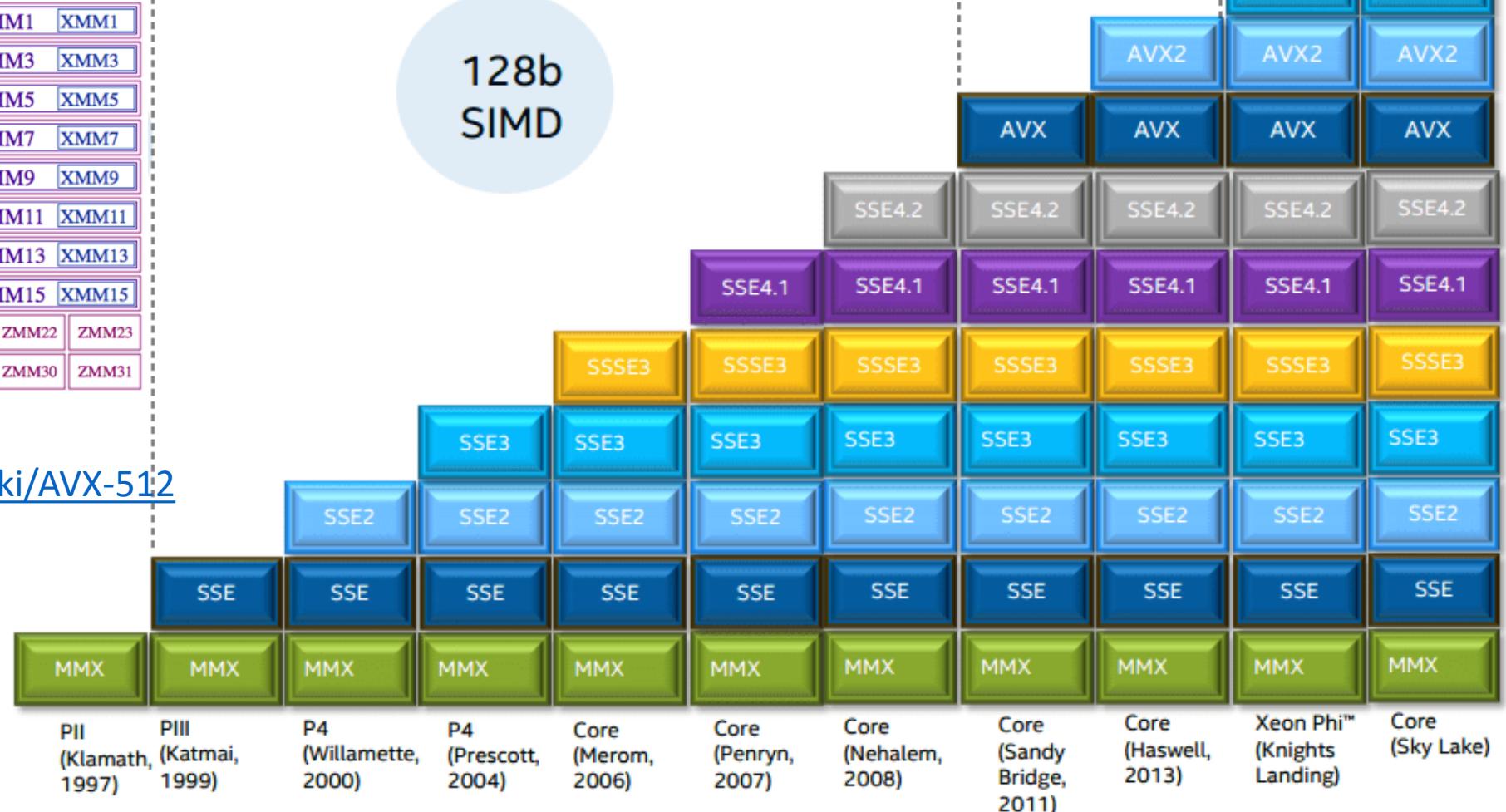
Recap

Intel SIMD ISA Evolution

SIMD extensions on top of x86/x87

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1		
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3		
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5		
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7		
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9		
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11		
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13		
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15		
ZMM16	ZMM17	ZMM18	ZMM19	ZMM20	ZMM21	ZMM22	ZMM23
ZMM24	ZMM25	ZMM26	ZMM27	ZMM28	ZMM29	ZMM30	ZMM31

<https://en.wikipedia.org/wiki/AVX-512>



Crash course in SIMD assembly

Register names

- SSE : xmm0 to xmm15 (128 bits)
- AVX2 : ymm0 to ymm15 (256 bits)
- AVX512 : zmm0 to zmm31 (512 bits)

In scalar mode, SSE registers are used

floating point instruction names

where $\langle\text{op}\rangle\langle\text{simd or not}\rangle\langle\text{raw type}\rangle$

- $\langle\text{op}\rangle$ is something like `vmul`, `vadd`, `vmov` or `vfmadd`
- $\langle\text{simd or not}\rangle$ is either '`s`' for scalar or '`p`' for packed (i.e. vector)
- raw type is either '`s`' for single precision or '`d`' for double precision

Typically :

`vmulss`, `vmovaps`, `vaddpd`, `vfmaddpd`

Istantanea Schermo

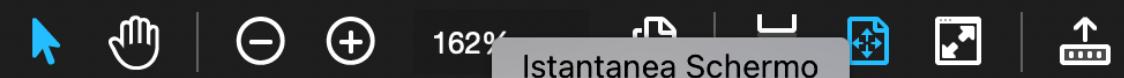
Practical look at assembly

Extract assembly code

- Run `objdump -d -C` on your executable or library
- Search for your function name **Or use Intel Advisor**

Check for vectorization

- For avx2, look for ymm
- For avx512, look for zmm
- Othersize look for instructions with ps or pd at the end
 - but ignore mov operations
 - only concentrate on arithmetic ones



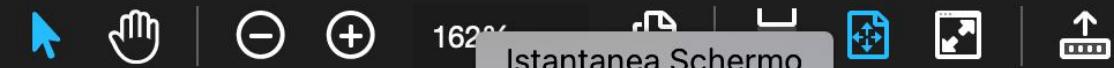
Exercise 1

Code

d18:	c5 fc 59 d8	vmulps %ymm0,%ymm0,%ymm3
d1c:	c5 fc 58 c0	vaddps %ymm0,%ymm0,%ymm0
d20:	c5 e4 5c de	vsubps %ymm6,%ymm3,%ymm3
d24:	c4 c1 7c 59 c0	vmulps %ymm8,%ymm0,%ymm0
d29:	c4 c1 64 58 da	vaddps %ymm10,%ymm3,%ymm3
d2e:	c4 41 7c 58 c3	vaddps %ymm11,%ymm0,%ymm8
d33:	c5 e4 59 d3	vmulps %ymm3,%ymm3,%ymm2
d37:	c4 c1 3c 59 f0	vmulps %ymm8,%ymm8,%ymm6
d3c:	c5 ec 58 d6	vaddps %ymm6,%ymm2,%ymm2

Solution

- Presence of ymm
- Vectorized, AVX level



Data alignment

- Data alignment is a method to force the compiler to create data objects in memory on specific byte boundaries.
- This is done to increase efficiency of data loads and stores to and from the processor.
- When using SSE intrinsics, you should align data to 16 bytes in memory operations.
With AVX512 you should consider 64 bytes.
- Contiguous memory locations are loaded in an easy way, otherwise "by hand"

```
// a is 16-byte aligned
float a[4] = {1.0, 2.0, 3.0, 4.0};
__m128 t = _mm_load_ps(a);
```

p: packed
s: single

LSB

1.0	2.0	3.0	4.0
-----	-----	-----	-----

- Same result as

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
```

How to align Static Arrays

- You can use an appropriate clause/attribute or option as described below to align the base-pointer.
- Once you do this, the data will be allocated at a good boundary AND the compiler will recognize alignment of the base-pointer for these arrays at all use sites.
- With Windows C/C++ `__declspec(align(64)) float A[1000];`
- With Linux/Mac C/C++ `float A[1000] __attribute__((aligned(64)));`

<https://www.intel.com/content/www/us/en/developer/articles/technical/data-alignment-to-assist-vectorization.html>

How to align Dynamic data

- Replace malloc() and free() with alignment specified replacement _mm_malloc() and _mm_free().
- These routines take an alignment parameter (in bytes) as the second argument. These replacements provided by the Intel C++ Compiler also use the same size argument and return types as malloc() and free().
- The returned data will be 64 byte aligned.

```
buf = (char*) _mm_malloc(bufsizes[i], 64);
```

- Note that for C/C++ arrays that are dynamically allocated, **it is not enough**. It is also required to use a clause of the form `__assume_aligned(a, 64)` before the loop of interest.
- Without this step, the compiler will not detect the optimal alignment for accesses using such arrays. This is discussed more in a later section of this article.
- Different in GCC https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html

```
void myfunc2( double *p2, double *p3, double *p4, int n)
{
    for (int j=0; j<n; j+=8) {
        __assume_aligned(p2, 64);
        __assume_aligned(p3, 64);
        __assume_aligned(p4, 64);
        for (int i=0; i<8; i++) {
            p2[j+i] = p3[j+i] * p4[j+i];
        }
    }
}
```

Instrinsics example with SSE

We assume properly aligned vectors

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; ++i) C[i] = A[i] + B[i];  
}
```

Solution simple if $len = 4 * \text{something}$

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; i += 4) {  
        __m128i v1 = loadu_si128(&A[i]);  
        __m128i v2 = loadu_si128(&B[i]);  
        __m128i res = add4(v1, v2);  
        store_si128(&C[i], res);  
    } }
```

Pseudocode

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    for (int i = 0; i < len; i += 4) {  
        v1 = load 4 values from A  
        v2 = load 4 values from B  
        res = 4-packed add(v1, v2)  
        store res (which is 4 values) in C  
    } }
```

Full solution

```
void sum_arrays(int *A, int *B, int *C, int len) {  
    int end = (len % 4 == 0) ? len : len - 4;  
    int i = 0;  
    for (; i < end; i += 4) {  
        __m128i v1 = loadu_si128(&A[i]);  
        __m128i v2 = loadu_si128(&B[i]);  
        __m128i res = add4(v1, v2);  
        store_si128(&C[i], res); }  
    for (; i < len; ++i) C[i] = A[i] + B[i];  
}
```

BASIC requirements for Vectorization

Countable at runtime

- Number of loop iterations is known before loop executes (i does not change in the body)
- No conditional termination (break statements)

Have single control flow

- No switch statements
- 'if' statements are allowable when they can be implemented as masked assignments (e.g. if .. var = x else var = y)

Must be the innermost loop if nested

- The outermost loop is normally parallelized with openMP
- Compiler may reverse loop order as an optimization!

No function calls

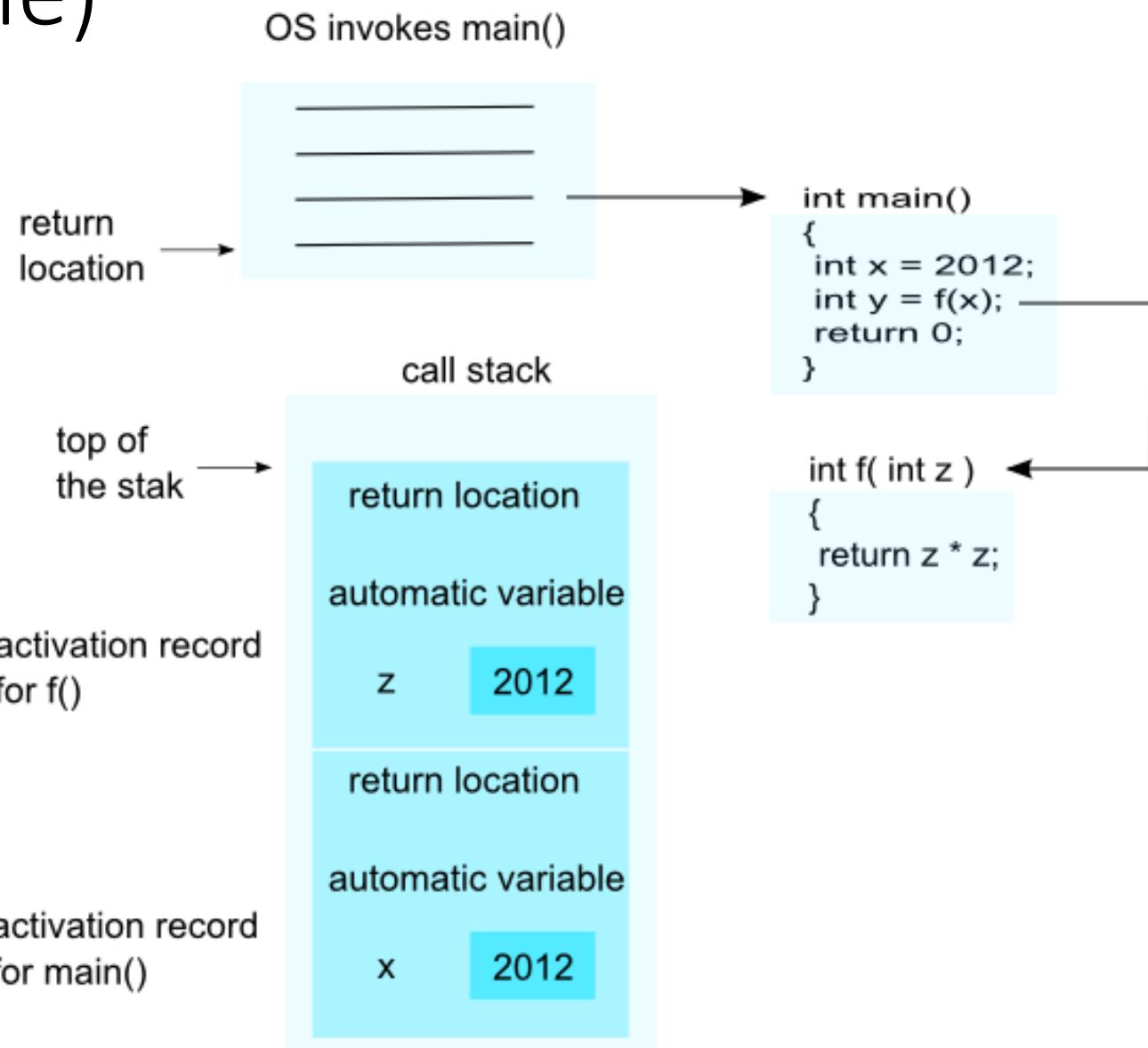
- Basic math is allowed: pow(), sqrt(), sin(), etc
- Some inline functions allowed

```
while (i < 100) {  
    a[i] = b[i] * c[i];  
    if (a[i] < 0.0) // data-dependent  
        break;  
    ++i;  
} // loop not vectorized
```

```
for (int i = 0; i < length; i++) {  
    float s = b[i] * b[i] - 4 * a[i] * c[i];  
    if (s >= 0) x[i] = sqrt(s);  
    else x[i] = 0.;  
} // loop vectorized (because of masking)
```

NOFUNC CALL (example)

- Function calls can represent a major bottleneck during program execution since they usually force register contents to be spilled into memory.
- Such register to memory spills are much more pronounced in presence of recursion.
- A function call is usually accompanied by the creation of its activation record at function entry.



See for example https://flylib.com/books/en/2.253.1/function_call_stack_and_activation_records.html

Vectorization Report

Shows which loops are or are not vectorized and why with -qopt-report-phase=vec
By default, reports are written to files suffixed with .optrpt, one per source file.

- Intel: -qopt-report =<n>
 - 0: None
 - 1: Lists vectorized loops
 - 2: 1 + Lists loops not vectorized, with explanation
 - 3: 2 + Outputs additional dependency information
 - 4: 3+ Lists loops not vectorized, without explanation
 - 5: All the available info
- Reports are essential for determining where the compiler finds a dependency
- Compiler is conservative, you need to go back and verify that there really is a dependency.
- Works also with -qopt-report-phase=openmp or par
- **Don't use –ipo when you ask the report!**

Details

```
icc -std=c99 -O3 -D NOFUNCCALL -qopt-report=2 -qopt-report-phase=vec Multiply.c Driver.c -o MatVector
```

```
LOOP BEGIN at Driver.c(145,2)
  remark #15542: loop was not vectorized: inner loop was already ve
  LOOP BEGIN at Driver.c(148,3)
    remark #15542: loop was not vectorized: inner loop was already
  LOOP BEGIN at Driver.c(150,4)
    remark #15300: LOOP WAS VECTORIZED
  LOOP END

  LOOP BEGIN at Driver.c(150,4)
  <Remainder loop for vectorization>
  LOOP END
  LOOP END
  LOOP END

143      //start timing the matrix multiply code
144      startTime = clock_it();
145      for (k = 0;k < REPEATNTIMES;k++) {
146      #ifdef NOFUNCCALL
147          int i, j;
148          for (i = 0; i < size1; i++) {
149              b[i] = 0;
150              for (j = 0;j < size2; j++) {
151                  b[i] += a[i][j] * x[j];
152              }
153          }
154      #else
155          matvec(size1,size2,a,b,x);
156      #endif
157          x[0] = x[0] + 0.000001;
158      }
159      endTime = clock_it();
160      execTime = endTime - startTime;
```

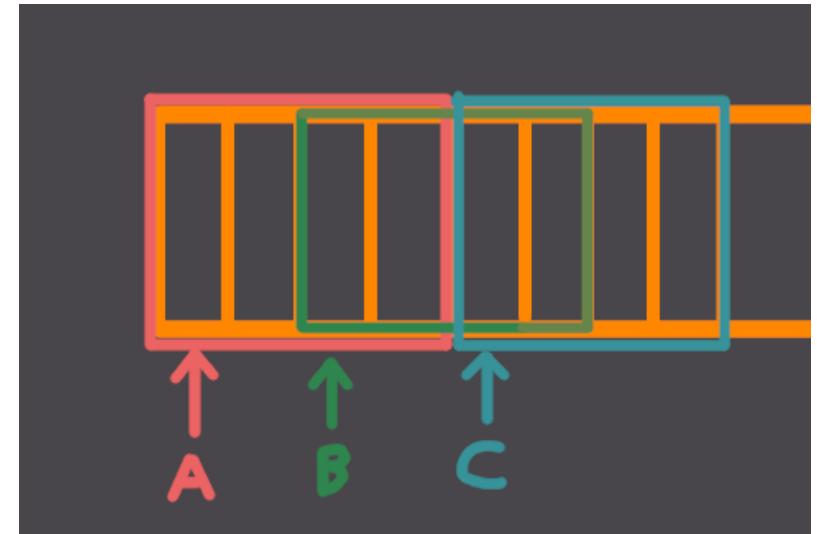
This is not true if you omit -D NOFUNCCALL

```
icc -std=c99 -O3 -qopt-report=2 -qopt-report-phase=vec
Multiply.c Driver.c -o MatVector

Execution time is 44.542 seconds
GigaFlops = 0.458044
```

Aliasing and correctness

$C[i] = A[i] + B[i]$, but $B[0-1]$ is $A[2-3]$ and,
more important, $C[0-1]$ is $B[2-3]$



- In the scalar version, we first write into $C[0]$ in the first iteration ($i==0$) and then we read from it (using $B[2]$) in the third iteration ($i==2$)
- In the vectorized version, the first iteration covers 0-4. We will read the old value of $B[2]$ (i.e. whatever was there before), before writing into it and that's obviously wrong.
- In these cases no way, the compiler cannot vectorialise. But if they do not overlap? We must say it "trust me, this pointer does not alias with any other pointer in the function"

Aliases

Two pointers are aliased if both point to the same memory location. Storing to memory using a pointer that might be aliased may prevent some optimizations.

Sometimes, the compiler can generate both a vectorized and a non-vectorized version of a loop and test for aliasing at runtime to select the appropriate code path.

If you know that pointers do not alias and **inform the compiler**, it can avoid the runtime check.

```
icc ... -DNOALIAS -xHost Multiply.c Driver.c  
-o MatVector
```

Execution time is 22.623 seconds
GigaFlops = 0.901823

Vs (#else branch)

Execution time is 45.749 seconds
GigaFlops = 0.445955

```
LOOP BEGIN at Multiply.c(55,3)  
  remark #15344: loop was not vectorized: vector dependence prevents vectorization.  
  First dependence is shown below. Use level 5 report for details  
    remark #15346: vector dependence: assumed FLOW dependence between b[i] (56:4) and  
    b[i] (56:4)  
    remark #25439: unrolled with remainder by 2  
LOOP END
```

```
37 #ifdef NOALIAS  
38 void matvec(int size1, int size2, FTYPE a[][size2], FTYPE b[restrict], FTYPE x[])  
39 #else  
40 void matvec(int size1, int size2, FTYPE a[][size2], FTYPE b[], FTYPE x[])  
41 #endif  
42 {  
43     int i, j;  
44  
45     for (i = 0; i < size1; i++) {  
46         b[i] = 0;  
47  
48 #ifdef ALIGNED  
49 // The pragma vector aligned below tells the compiler to assume that the data in  
50 // the loop is aligned on 16-byte boundary so the vectorizer can use  
51 // aligned instructions to generate faster code.  
52 #pragma vector aligned  
53 #endif  
54  
55     for (j = 0; j < size2; j++) {  
56         b[i] += a[i][j] * x[j];  
57     }  
58 }  
59 }
```

Restrict is c99, not available in C++.
In these cases is compiler-specific,
e.g. int * __restrict__ p

Details

With -DNOALIAS

```
icc -std=c99 -O3 -qopt-report=5 -qopt-report-phase=vec -DNOALIAS -xHost Multiply.c Driver.c -o MatVector
```

See next slide for peeled and remainder concepts.

You already saw data alignment.

```
LOOP BEGIN at Multiply.c(55,3)
<Peeled loop for vectorization>
  remark #15389: vectorization support: reference a[i][j] has unaligned access [ Multiply.c(56,12) ]
  remark #15389: vectorization support: reference x[j] has unaligned access [ Multiply.c(56,22) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15335: peel loop was not vectorized: vectorization possible but seems inefficient. Use vector a
  remark #15305: vectorization support: vector length 8
  remark #15309: vectorization support: normalized vectorization overhead 1.955
LOOP END

LOOP BEGIN at Multiply.c(55,3)
  remark #15389: vectorization support: reference a[i][j] has unaligned access [ Multiply.c(56,12) ]
  remark #15389: vectorization support: reference x[j] has unaligned access [ Multiply.c(56,22) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 2.733
  remark #15300: LOOP WAS VECTORIZED
  remark #15442: entire loop may be executed in remainder
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 8
  remark #15477: vector cost: 0.930
  remark #15478: estimated potential speedup: 6.460
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at Multiply.c(55,3)
<Remainder loop for vectorization>
  remark #15389: vectorization support: reference a[i][j] has unaligned access [ Multiply.c(56,12) ]
  remark #15389: vectorization support: reference x[j] has unaligned access [ Multiply.c(56,22) ]
  remark #15381: vectorization support: unaligned access used inside loop body
  remark #15305: vectorization support: vector length 8
  remark #15309: vectorization support: normalized vectorization overhead 1.955
  remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
LOOP END
```

```
LOOP BEGIN at Multiply.c(45,2)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between b[i] (46:3) and a[i][j] (56:4)
  remark #15346: vector dependence: assumed ANTI dependence between a[i][j] (56:4) and b[i] (46:3)
```

```
LOOP BEGIN at Multiply.c(55,3)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between b[i] (56:4) and a[i][j] (56:4)
  remark #15346: vector dependence: assumed ANTI dependence between a[i][j] (56:4) and b[i] (56:4)
  remark #15346: vector dependence: assumed ANTI dependence between b[i] (56:4) and b[i] (56:4)
  remark #15346: vector dependence: assumed FLOW dependence between b[i] (56:4) and b[i] (56:4)
  remark #15346: vector dependence: assumed FLOW dependence between b[i] (56:4) and b[i] (56:4)
  remark #15346: vector dependence: assumed ANTI dependence between b[i] (56:4) and b[i] (56:4)
  LOOP END
```

Without -DNOALIAS

Other Transformations – Loop peeling

- Remove the first/s and/or the last/s iteration of the loop into separate code outside the loop
- It allows enforcing a particular initial memory alignment on array references prior to loop vectorization

LOOP BEGIN at gas_dyn2.f90(2330,26)

<Peeled>

remark #15389: vectorization support: reference AMAC1U has unaligned access
remark #15381: vectorization support: unaligned access used inside loop body
remark #15301: PEEL LOOP WAS VECTORIZED

LOOP END

LOOP BEGIN at gas_dyn2.f90(2330,26)

remark #25084: Preprocess Loopnests: Moving Out Store
remark #15388: vectorization support: reference AMAC1U has aligned access
remark #15399: vectorization support: unroll factor set to 2
remark #15300: LOOP WAS VECTORIZED
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 8
remark #15477: vector loop cost: 0.620
remark #15478: estimated potential speedup: 15.890
remark #15479: lightweight vector operations: 5
remark #15488: --- end vector loop cost summary ---

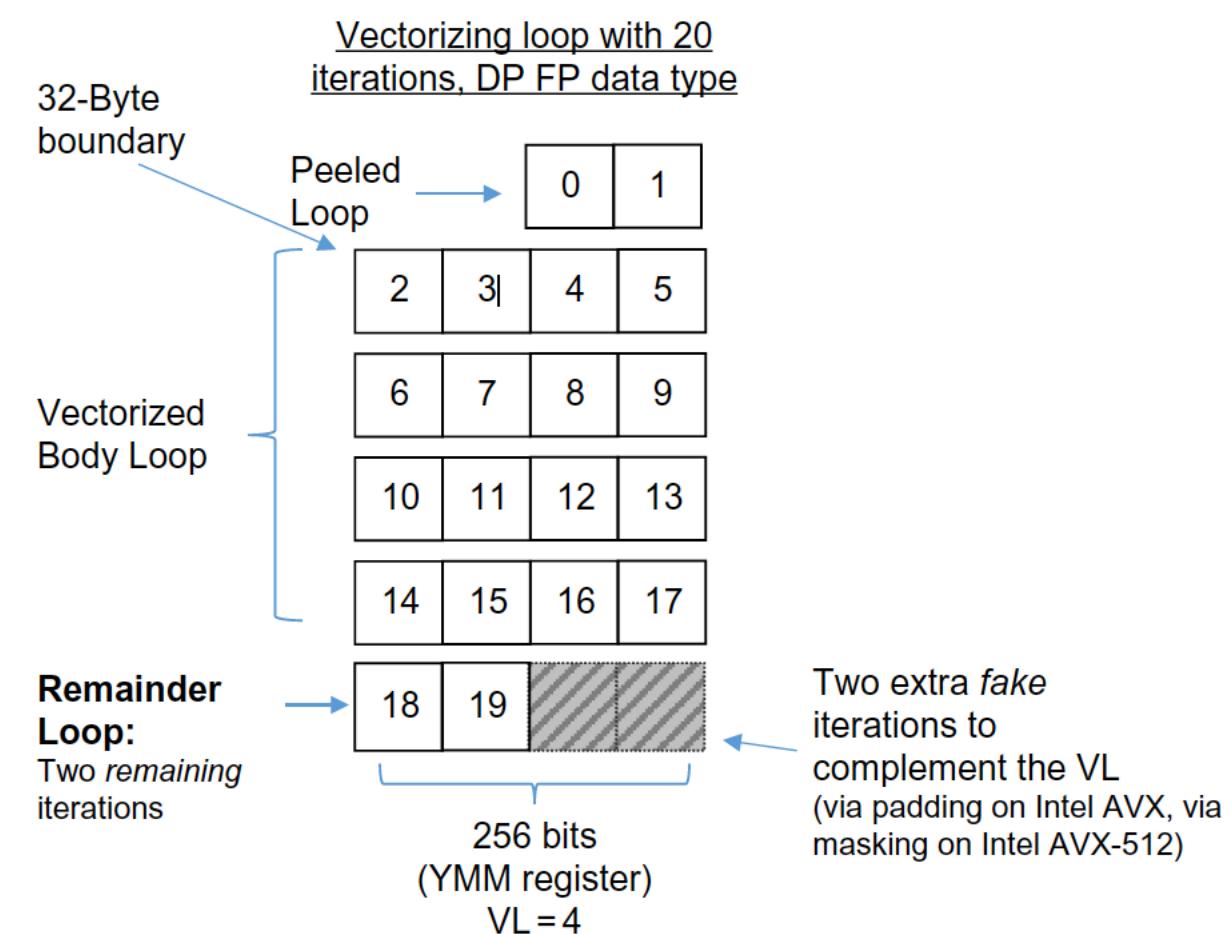
LOOP END

LOOP BEGIN at gas_dyn2.f90(2330,26)

<Remainder>

remark #15388: vectorization support: reference AMAC1U has aligned access
remark #15388: vectorization support: reference AMAC1U has aligned access
remark #15301: REMAINDER LOOP WAS VECTORIZED

LOOP END



Remember

Xeon Phi has been designed for providing high performance **when using all the cores**. It is however based on an Intel Atom architecture, therefore a classic Xeon processor like my E5645 provides the same or better performance for less \$ (about \$500 vs \$2,500).

Here the E5645 performance

- `icc -O0 -D NOFUNCCALL Multiply.c Driver.c -o MatVector`
Execution time is 49.071 seconds - GigaFlops = 0.415761
- `icc -O3 -D NOFUNCCALL Multiply.c Driver.c -o MatVector`
Execution time is 6.733 seconds - GigaFlops = 3.030179
- `icc -std=c99 -D NOALIAS -D ALIGNED -ipo Multiply.c Driver.c -o MatVector`
Execution time is **5.193** seconds - GigaFlops = **3.928563**
- `pgcc -fast -O3 -D NOALIAS -D ALIGNED Multiply.c Driver.c -o MatVector`
Execution time is 10.873 seconds - GigaFlops = 1.876454

Also linuxge.ge.infn.it is faster...

Another requirement for Vectorization: CORRECTNESS!

- Vectorization changes the order of computation compared to sequential case!
- Therefore, compiler must be able to prove that vectorization will produce correct result.
 - A first example was the pointer aliasing
- Need to consider independence of *unrolled* loop operations
- Compiler performs dependency analysis
- Rule of thumb: if you can execute the loop in the opposite order
(e.g. for $i=n; i>0; i--$) it should be ok

Auto vectorization: not all loops will vectorize

Data dependencies between iterations

- Proven Read-after-Write data (i.e., loop carried) dependencies
- Assumed data dependencies
 - Aggressive optimizations (e.g., IPO) might help

Vectorization won't be efficient

- Compiler estimates how better the vectorized version will be
- Affected by data alignment, data layout, etc.

Unsupported loop structure

- While-loop, for-loop with unknown number of iterations
- Complex loops, unsupported data types, etc.
- (Some) function calls within loop bodies
 - Not the case for SVML functions

RaW dependency

```
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

Inefficient vectorization

```
for (int i = 0; i < N; i++)
    a[c[i]] = b[d[i]];
```

Function call within loop body

```
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```

Loop-Carried Dependence vs. Loop Independent Dep.

They are determined by the relationships between statements in iterations of a loop

- LCD: when a statement in one iteration of a loop depends in some way on a statement in a **different iteration** of the same loop
- LID: if a statement in one iteration of a loop depends only on a statement **in the same iteration** of the loop

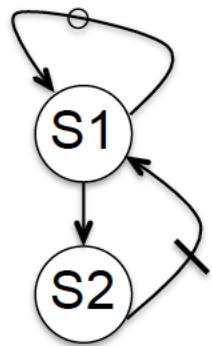
AND

- Two memory accesses are involved in a **data dependence** if they may refer to the same memory location **and** one of the references is a **write**

```
// Code block 1
for(i = 1 i < 4; i++)
    LCD
    S1: b[i] = 8;
    S2: a[i] = b[i-1] + 10;
```

```
// Code block 2
for(i = 0; i < 4; i++)
    LID
    S1: b[i] = 8;
    S2: a[i] = b[i] + 10;
```

LCD vs LID



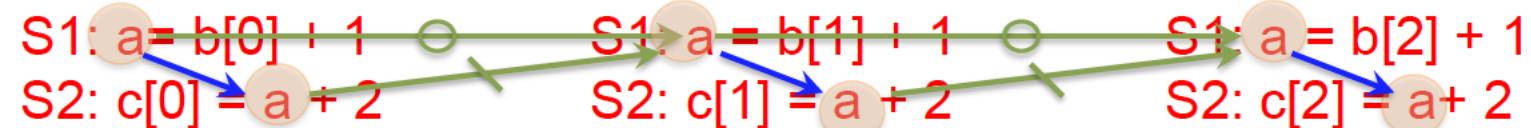
```

for (i=0; i<n; i++){
S1  a = b[i] + 1;
S2  c[i] = a + 2;
}
  
```

i=0

i=1

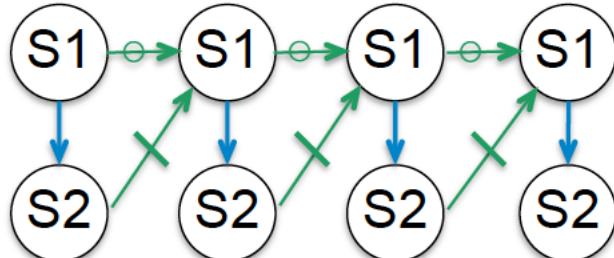
i=2



iteration:

0 1 2 3 ...

instances of S1:



instances of S2:

- Loop independent dependence
- Loop carried dependence
- → Write after Write
- + → Write after read
- Read after Write

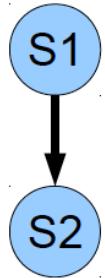
Data Dependence

Read After Write

- Also called “flow” dependency
- Variable written first, then read
- Not vectorizable

```
for( i=1; i<N; i++)  
    a[i] = a[i-1] + b[i];
```

S1 $a = b + c;$
S2 $d = 2 * a;$



Write after Read

- Also called “anti” dependency
- Variable read first, then written
- vectorizable

```
for( i=0; i<N-1; i++)  
    a[i] = a[i+1] + b[i];
```

S1 $c = a + b;$
S2 $a = 2 * a;$

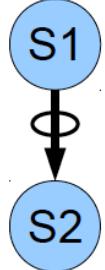


Write after Write

- a.k.a “output” dependency
- Variable written, then re-written
- Not vectorizable

```
for( i=0; i<N; i++)  
    a[i%2] = b[i] + c[i];
```

S1 $a = k;$
if ($a > 0$) {
 a = 2 * c;
}



Read after Read is **not** a real dependency, thus always vectorizable

Loop dependencies: RAW

Consider the loop:

$a = \{0, 1, 2, 3, 4\}$

$b = \{5, 6, 7, 8, 9\}$

```
for ( i=1; i<N; i++)
    a[i] = a[i-1] + b[i];
```

Applying each operation sequentially:

$$a[1] = a[0] + b[1] \rightarrow a[1] = 0 + 6 \rightarrow a[1] = 6$$

$$a[2] = a[1] + b[2] \rightarrow a[2] = 6 + 7 \rightarrow a[2] = 13$$

$$a[3] = a[2] + b[3] \rightarrow a[3] = 13 + 8 \rightarrow a[3] = 21$$

$$a[4] = a[3] + b[4] \rightarrow a[4] = 21 + 9 \rightarrow a[4] = 30$$

$a = \{0, 6, 13, 21, 30\}$

Loop dependencies: RAW

Now let's try vector operations:

$a = \{0, 1, 2, 3, 4\}$

$b = \{5, 6, 7, 8, 9\}$

```
for( i=1; i<N; i++)
    a[i] = a[i-1] + b[i];
```

Applying vector operations, $i=\{1,2,3,4\}$:

$a[i-1] = \{0, 1, 2, 3\}$ (load)

$b[i] = \{6, 7, 8, 9\}$ (load)

$\{0, 1, 2, 3\} + \{6, 7, 8, 9\} = \{6, 8, 10, 12\}$ (operate)

A should be { 0, 6, 13, 21 ... }

$a[i] = \{6, 8, 10, 12\}$ (store)

$a = \{0, 6, 8, 10, 12\} \neq \{0, 6, 13, 21, 30\}$ NOT VECTORIZABLE

Loop dependencies: WAR

Consider the loop:

a = {0,1,2,3,4}

b = {5,6,7,8,9}

```
for ( i=0; i<N; i++)
    a[i] = a[i+1] + b[i];
```

Applying each operation sequentially:

$$a[0] = a[1] + b[0] \rightarrow a[0] = 1 + 5 \rightarrow a[0] = 6$$

$$a[1] = a[2] + b[1] \rightarrow a[1] = 2 + 6 \rightarrow a[1] = 8$$

$$a[2] = a[3] + b[2] \rightarrow a[2] = 3 + 7 \rightarrow a[2] = 10$$

$$a[3] = a[4] + b[3] \rightarrow a[3] = 4 + 8 \rightarrow a[3] = 12$$

a = {6, 8, 10, 12 , 4}

Loop dependencies: WAR

Now let's try vector operations:

a = {0,1,2,3,4}

b = {5,6,7,8,9}

```
for( i=0; i<N; i++)
    a[i] = a[i+1] + b[i];
```

Applying vector operations, i={1,2,3,4}:

a[i+1] = {1,2,3,4} (load)

b[i] = {5,6,7,8} (load)

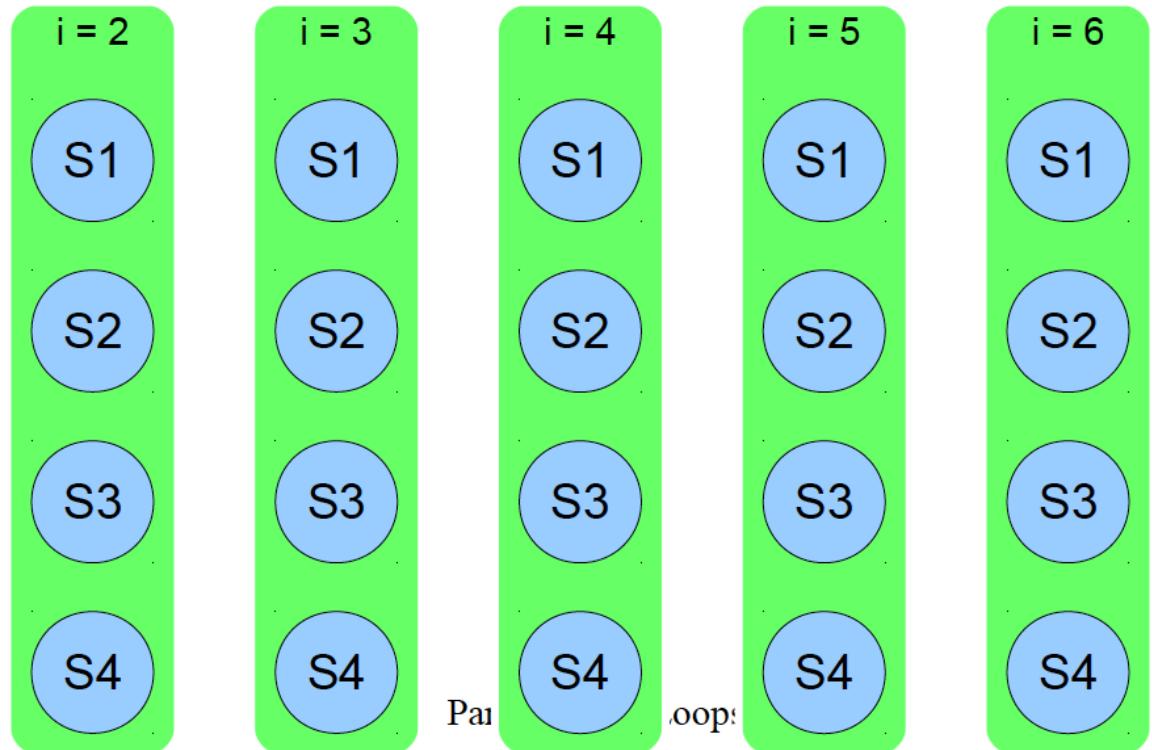
{1,2,3,4} + {5,6,7,8} = {6, 8, 10, 12} (operate)

a[i] = {6, 8, 10, 12} (store)

a = {0, 6, 8, 10, 12} = {0, 6, 8, 10, 12} VECTORIZABLE

Exercise: draw the dependencies

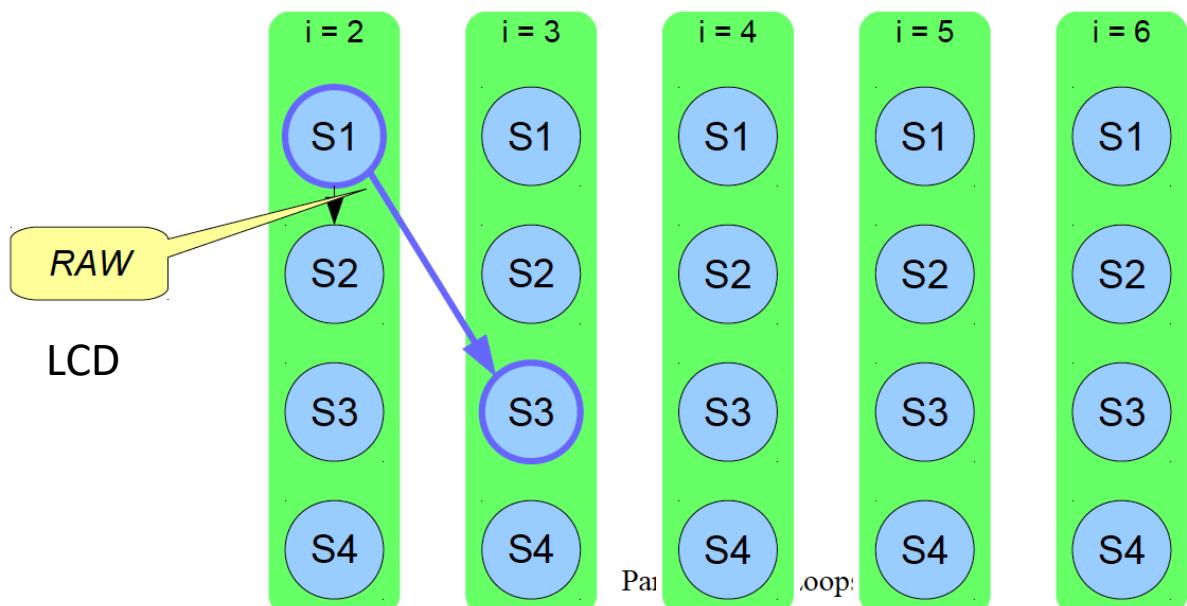
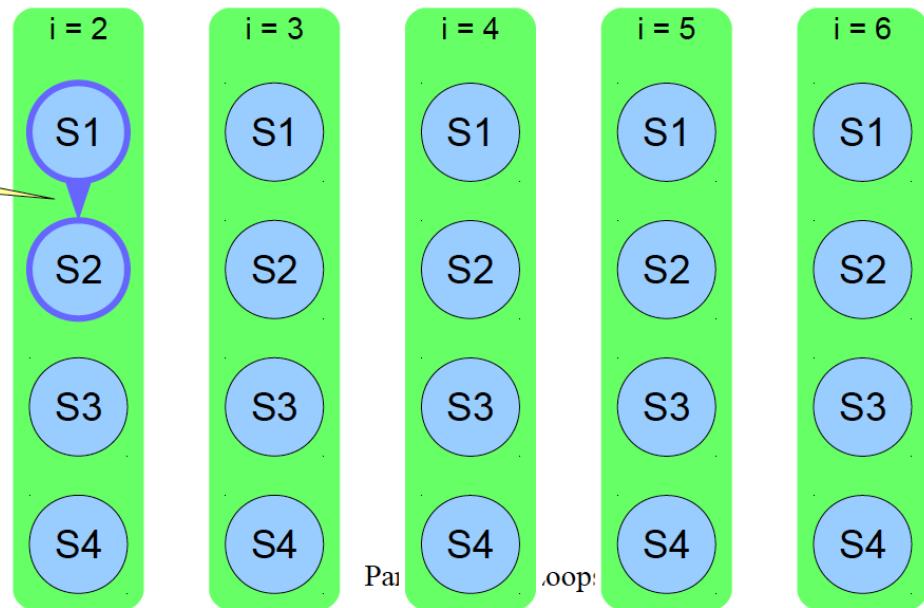
```
for (i=2; i<n; i++) {  
    S1 a[i] = 4 * c[i-1] - 2;  
    S2 b[i] = a[i] * 2;  
    S3 c[i] = a[i-1] + 3;  
    S4 d[i] = b[i] + c[i-2];  
}
```



Dependencies on a[i]

```
for (i=2; i<n; i++) {  
    S1 a[i] = 4 * c[i-1] - 2;  
    S2 b[i] = a[i] * 2;  
    S3 c[i] = a[i-1] + 3;  
    S4 d[i] = b[i] + c[i-2];  
}
```

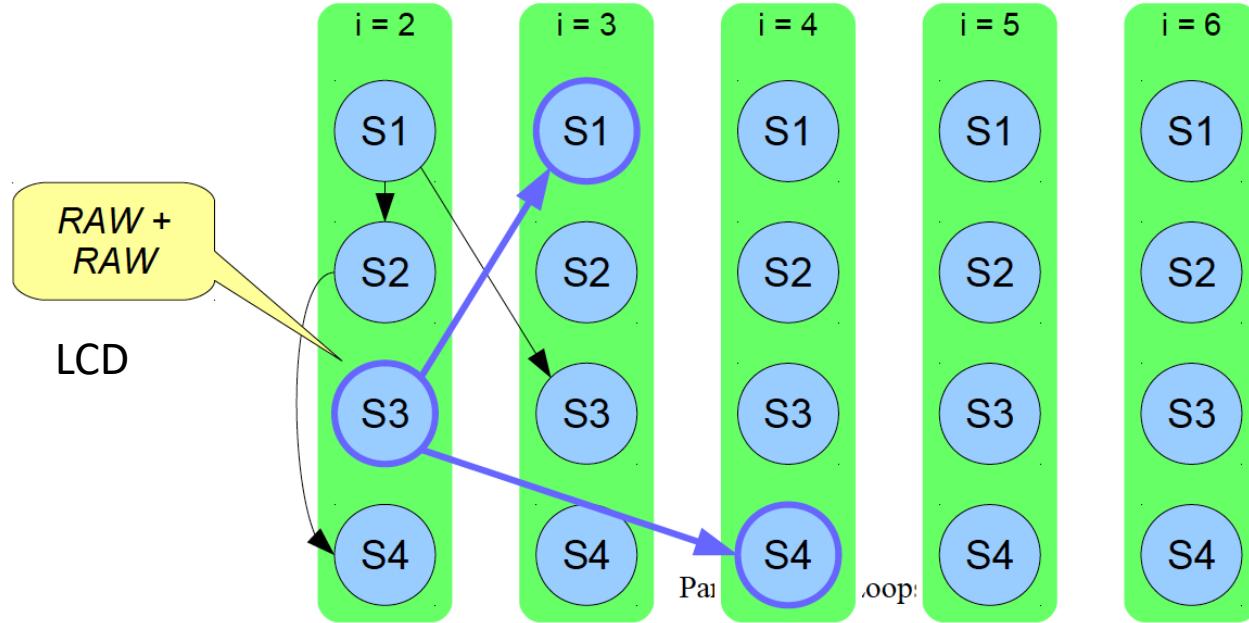
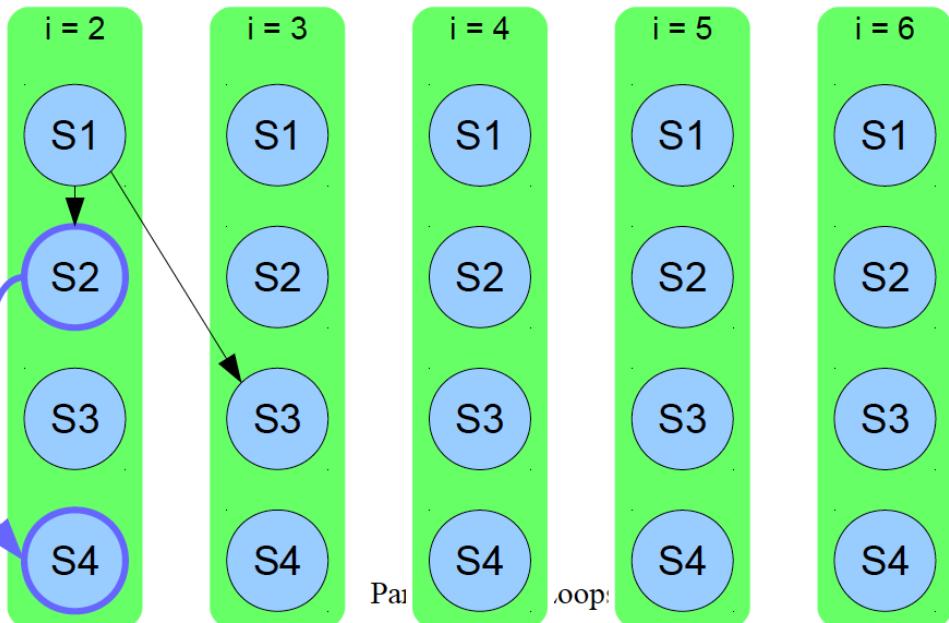
```
for (i=2; i<n; i++) {  
    S1 a[i] = 4 * c[i-1] - 2;  
    S2 b[i] = a[i] * 2;  
    S3 c[i] = a[i-1] + 3;  
    S4 d[i] = b[i] + c[i-2];  
}
```



Dependencies on b and c

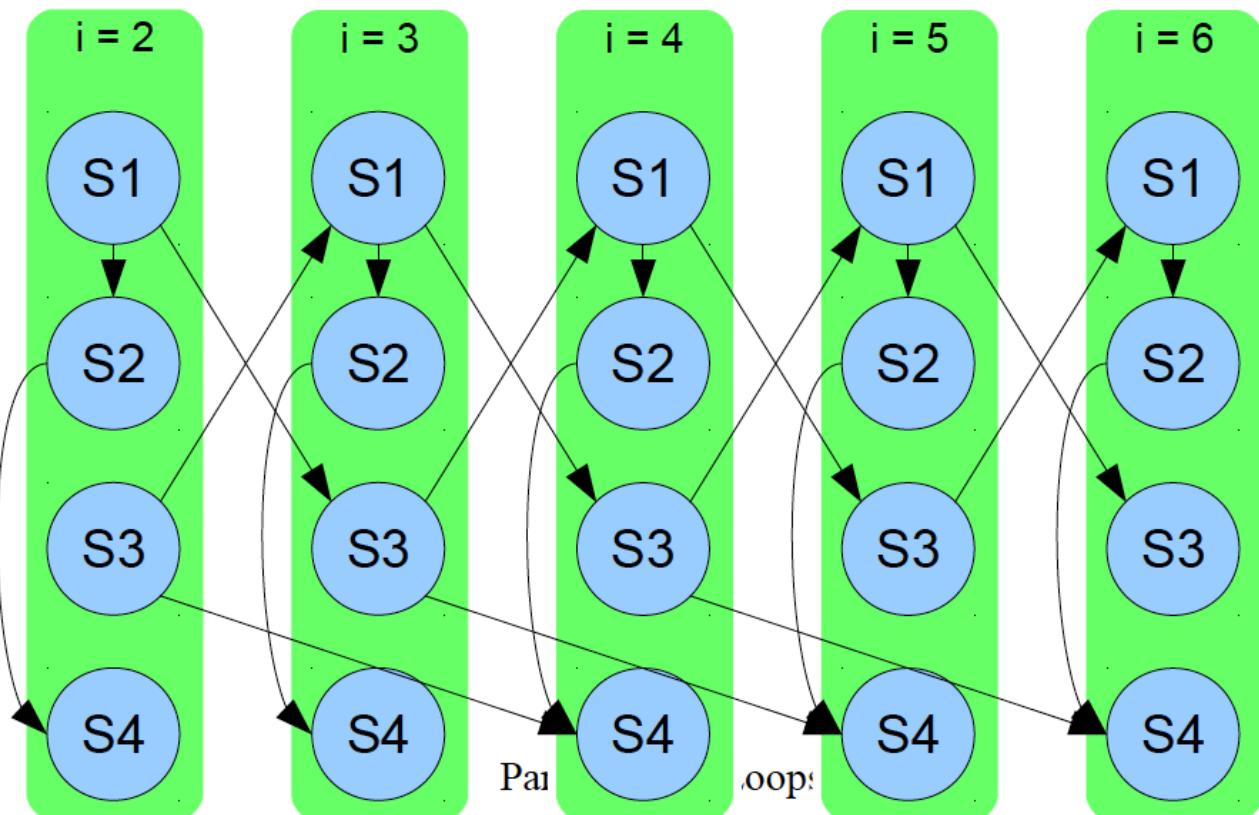
```
for (i=2; i<n; i++) {  
    S1 a[i] = 4 * c[i-1] - 2;  
    S2 b[i] = a[i] * 2;  
    S3 c[i] = a[i-1] + 3;  
    S4 d[i] = b[i] + c[i-2];  
}
```

```
for (i=2; i<n; i++) {  
    S1 a[i] = 4 * c[i-1] - 2;  
    S2 b[i] = a[i] * 2;  
    S3 c[i] = a[i-1] + 3;  
    S4 d[i] = b[i] + c[i-2];  
}
```



All the dependencies

```
for (i=2; i<n; i++) {  
    S1 a[i] = 4 * c[i-1] - 2;  
    S2 b[i] = a[i] * 2;  
    S3 c[i] = a[i-1] + 3;  
    S4 d[i] = b[i] + c[i-2];  
}
```



So?

Vectorization in most cases can be achieved by:

- Separating (distributing) the statements not in a cycle
- Removing dependences
- Freezing loops
- *Changing the algorithm*

In general, a statement inside a loop which is not in a cycle of the dependence graph (LCD) can be vectorized

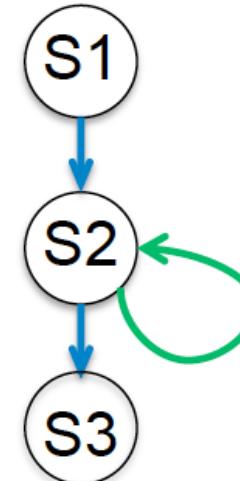
Distributing

Only S1 and S3 can be vectorized

```
for (i=2 i<n; i++){  
S1 b[i] = b[i] + c[i];  
S2 a[i] = a[i-1]*a[i-2]+b[i];  
S3 c[i] = a[i] + 1;  
}
```



```
b[1:n-1] = b[1:n-1] + c[1:n-1];  
for (i=1; i<n; i++){  
    a[i] = a[i-1]*a[i-2]+b[i];  
}  
c[1:n-1] = a[1:n-1] + 1;
```



- Loop independent dependence
- Loop carried dependence
- → Write after Write
- + → Write after read 😊
- Read after Write

It means vectorized

Removing dependences

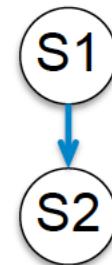
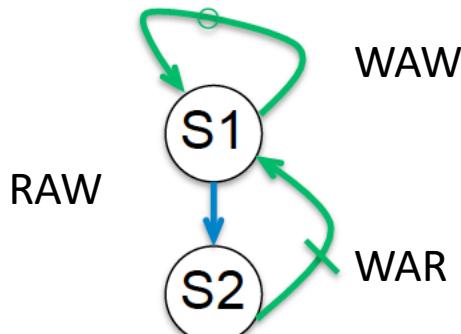
```
for (i=0; i<n; i++){  
S1  a = b[i] + 1;  
S2  c[i] = a + 2;  
}
```



```
for (i=0; i<n; i++){  
S1  a'[i] = b[i] + 1;  
S2  c[i] = a'[i] + 2;  
}  
a=a'[n-1]
```



```
S1  a'[0:n-1] = b[0:n-1] + 1;  
S2  c[0:n-1] = a'[0:n-1] + 2;  
a=a'[n-1]
```



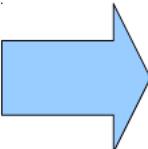
- Loop independent dependence
- Loop carried dependence
- Write after Write
- Write after read
- Read after Write



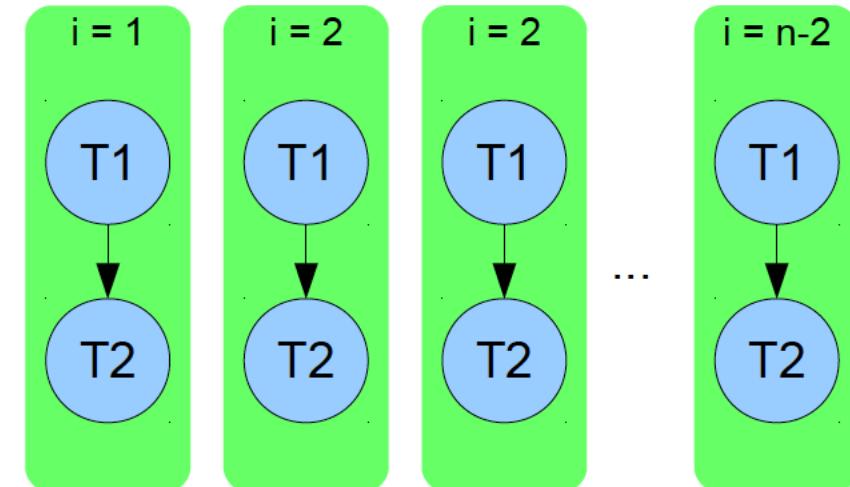
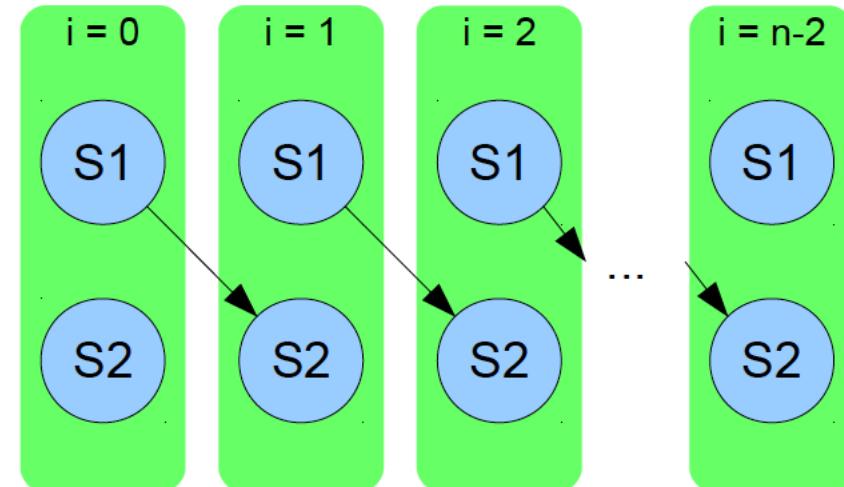
Removing dependencies: loop aligning

- Dependencies can sometimes be removed by **aligning** loop iterations

```
a[0] = 0;  
for (i=0; i<n-1; i++) {  
    S1 a[i+1] = b[i] * c[i];  
    S2 d[i] = a[i] + 2;  
}
```



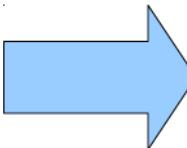
```
a[0] = 0;  
d[0] = a[0] + 2;  
for (i=1; i<n-1; i++) {  
    T1 a[i] = b[i-1] * c[i-1];  
    T2 d[i] = a[i] + 2;  
}  
a[n-1] = b[n-2] * c[n-2];
```



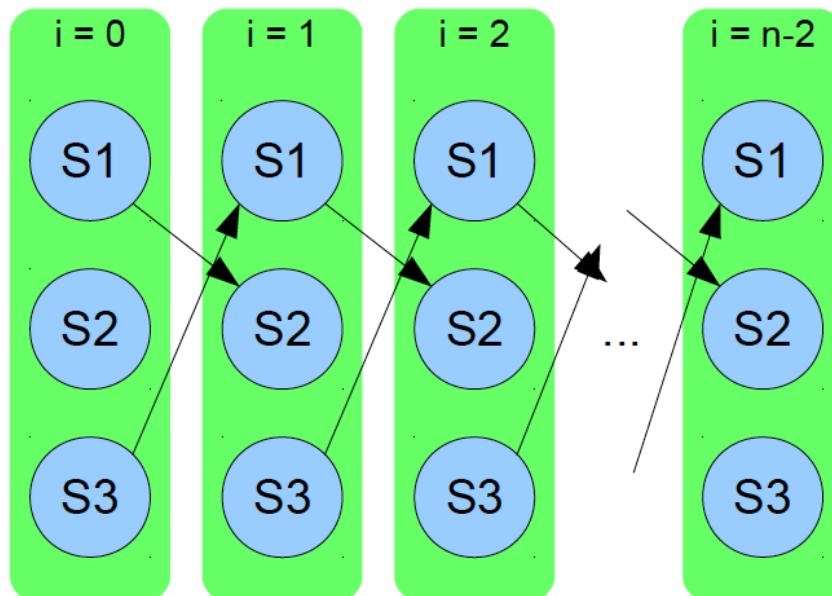
Removing dependencies: Reordering 1

- Some dependencies can be removed by **reordering**

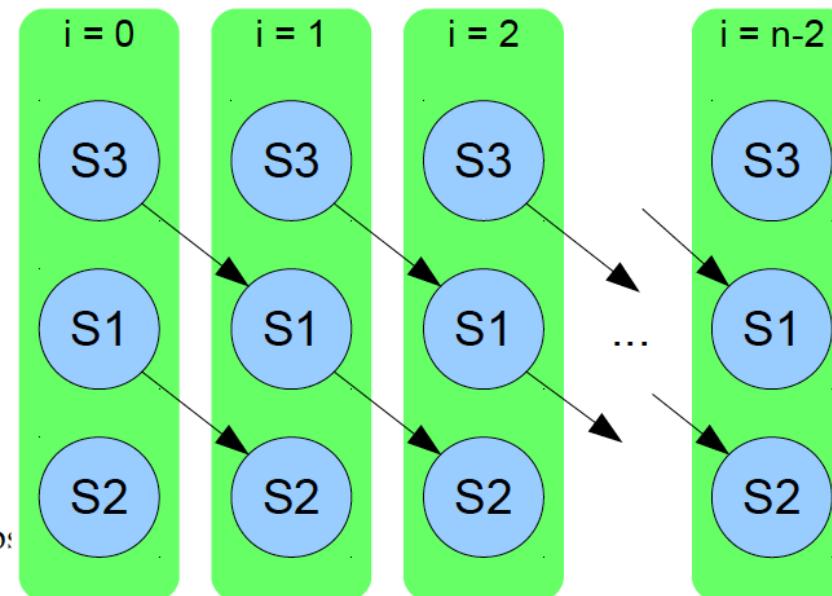
```
for (i=0; i<n-1; i++) {  
    S1 a[i+1] = c[i] + k;  
    S2 b[i]     = b[i] + a[i];  
    S3 c[i+1] = d[i] + w;  
}
```



```
for (i=0; i<n-1; i++) {  
    S3 c[i+1] = d[i] + w;  
    S1 a[i+1] = c[i] + k;  
    S2 b[i]     = b[i] + a[i];  
}
```



Original Loops



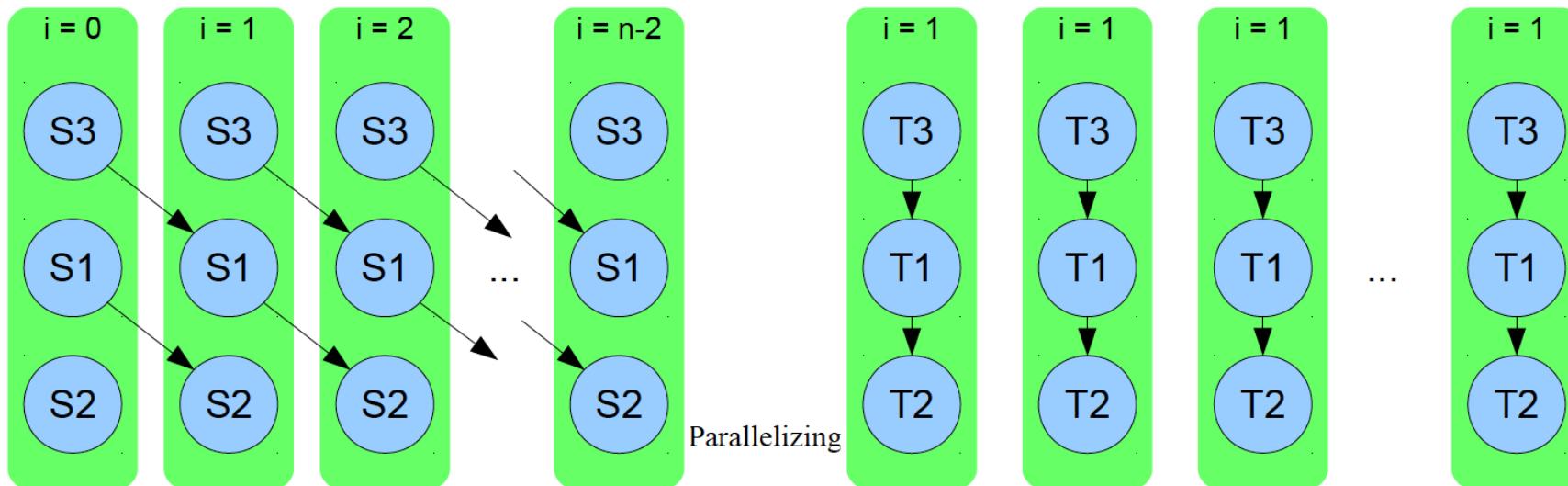
Reordered Loops

Removing dependencies: Reordering 2

- After reordering, we can align loop iterations

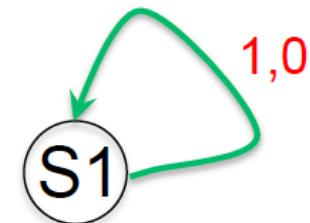
```
for (i=0; i<n-1; i++) {  
    S3 c[i+1] = d[i] + w;  
    S1 a[i+1] = c[i] + k;  
    S2 b[i]     = b[i] + a[i];  
}
```

```
b[0] = b[0] + a[0];  
a[1] = c[0] + k;  
b[1] = c[1] + a[1];  
for (i=1; i<n-2; i++) {  
    T3 c[i] = d[i-1] + w;  
    T1 a[i+1] = c[i] + k;  
    T2 b[i+1] = b[i+1] + a[i+1];  
}  
c[n-2] = d[n-3] + w;  
a[n-1] = c[n-2] + k;  
c[n-1] = d[n-2] + w;
```



Freezing Loops

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
        a[i][j]=a[i][j]+a[i-1][j];  
    }  
}
```



Ignoring (freezing) the outer loop:

```
for (j=1; j<n; j++) {  
    a[i][j]=a[i][j]+a[i-1][j];  
}
```



```
for (i=1; i<n; i++) {  
    a[i][1:n-1]=a[i][1:n-1]+a[i-1][1:n-1];  
}
```

Reduction

- A reduction variable (or simply "reduction") is an operation that happens in a specific variable, in every iteration of a loop.
- It is in the form: `new_value = old_value OP data`.
- That is, in every iteration, the new value of the variable is an operation involving the old value and some data.
 - a reduction is a special kind of a loop-carried dependency
 - But it can be parallelised for associative Ops: $((1 + 2) + 3) + 4 = (1+2) + (3+4)$, $(3+1)+(2+4)\dots$
 - The important thing here is that the OP is the *same* in *every* iteration.

```
int sum_of_array(int *a, int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i) sum = sum + a[i];  
}
```

#pragma simd foricc

```
void add_floats(float *a, float *b, float *c, float *d, float *e, int n){  
    int i;  
    #pragma simd  
    for (i=0; i<n; i++) a[i] = a[i] + b[i] + c[i] + d[i] + e[i];  
}
```

In the example above, the function add_floats()

- uses too many unknown pointers for the compiler's automatic runtime independence check optimization
- has a reduction

The programmer can enforce the vectorization of this loop by using the *simd* pragma to avoid the overhead of runtime check.

Non-Intel compiler can exploit #pragma omp simd

<https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/pragmas/intel-specific-pragma-reference/simd.html>

Further Hinting and Pragmas

```
void vec1(double s1, int M, int N, double *x) {  
    ...  
    for (i=ib; i<ie; i++) { x[i] = x[i+M] + s1;  
    }  
}
```

- In this case, the Intel compiler is able to recognize that certain **negative** values of M will result in a read-after-write dependency.
- But the compiler is not entirely prevented from vectorizing the loop; instead, it produces **multiversioned code** that can branch to either a vectorized or an unvectorized loop, depending on the value of M!
- In many ways this outcome is better than if we force the compiler to vectorize the loop through the compiler directives presented below. Why take the risk of generating incorrect results through the heavy-handed use of directives?

Further Hinting and Pragmas

- There are other instances where the compiler cannot so easily distinguish between safe and unsafe cases.
- In such instances, compiler hints via flags or pragmas can be the most suitable way to ensure safety.
- `#pragma ivdep` (Ignore Vector DEPENDencies)
It implies (or guarantees) that any apparent data dependencies are safe to ignore.
- NOTE: if the programmer is wrong, and the loop actually does have a "bad" dependency like read after write (as previous code with $M < 0$), the program will likely produce **incorrect** results

Further Hinting and Pragmas

#pragma omp simd early_exit

- icc specific
- Defines that the vector execution of the *for* loop is safe even though the loop may exit before the loop upper bound condition $j < ub$ becomes false
- Safety of such vector evaluation is programmer's responsibility.

```
void foo(int lb, int ub) {  
    float a = 0;  
    #pragma omp simd early_exit reduction(+:a)  
    for(j=lb; j<ub; j++) {  
        if (b[j] <= 0 )  
            break;  
        a += b[j];  
    }  
}
```

Recap on data access

- Data travel back and forth between memory levels in units of a cache line, about 64 bytes (8 doubles).
- Data properly aligned (i.e. allocated) move readily from registers to cache to memory in single operations.
- In the case of arguments to functions or subroutines, though, the compiler may not know for certain that all the variables given as arguments will be suitably aligned. You should specify it.
- It should be clear enough at this point of the course how to properly access both one- and multi-dimensional arrays.
- But what about structs?

Data layout

Use Structure of Arrays (SoA)
instead of Array of Structures (AoS).

```
struct Vector3s { //AoS
    double x;
    double y;
    double z;
};
```



```
struct Vectors3s { //SoA
    double* x;
    double* y;
    double* z;
};
```



Array of Structures vs Structure of Arrays

```
// Array of Structures (AoS)
struct coordinate {
    float x, y, z;
} crd[N];
...
for (int i = 0; i < N; i++)
    ... = ... f(crd[i].x, crd[i].y, crd[i].z);
```

Consecutive elements in memory →



```
// Structure of Arrays (SoA)
struct coordinate {
    float x[N], y[N], z[N];
} crd;
...
for (int i = 0; i < N; i++)
    ... = ... f(crd.x[i], crd.y[i], crd.z[i]);
```

Consecutive elements in memory →



<https://vorbrodt.blog/2019/01/31-aos-vs-soa-performance/>

Example

```
struct Person
{
    Person(const string& n, uint8_t a, uint32_t d)
        : name(n), age(a), dob(d) {}

    string name;
    uint8_t age;
    uint32_t dob;
};

using VP = vector<Person>;

void addPerson(VP& v, Person&& p) { v.push_back(move(p)); }

uint64_t averageNameLen(const VP& v)
{
    return accumulate(begin(v), end(v), (uint64_t)0,
        [] (auto sum, auto& p) { return sum + p.name.length(); }) / v.size();
}
```

```
struct Persons
{
    vector<string> names;
    vector<uint8_t> ages;
    vector<uint32_t> dobs;

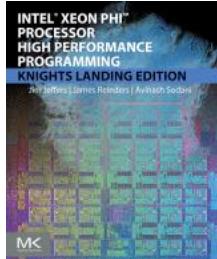
    void addPerson(const string& n, uint8_t a, uint32_t d)
    {
        names.push_back(n);
        ages.push_back(a);
        dobs.push_back(d);
    }

    uint64_t averageNameLen() const
    {
        return accumulate(begin(names), end(names), (uint64_t)0,
            [] (auto sum, auto& n) { return sum + n.length(); }) / names.size();
    }
}
```

Try icpc -std=c++17 array-structure.cpp -o array-structure (99 vs 62)
and icpc -std=c++17 -xHost array-structure.cpp -o array-structure (92 vs 28)

Useful links

- <http://www.sci.utah.edu/~mb/Teaching/Week5/9-Vectorization.pdf>
- <https://software.intel.com/en-us/advisor/documentation/get-started>
- <https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-automatic-vectorization>
- <https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-vectorization-and-loops>
- [https://hpc-forge.cineca.it/files/ScuolaCalcoloParallelo_WebDAV/public/anno-2017/26th Summer School on Parallel Computing/Bologna/SCP-KNL-vectorization.pdf](https://hpc-forge.cineca.it/files/ScuolaCalcoloParallelo_WebDAV/public/anno-2017/26th_Summer_School_on_Parallel_Computing/Bologna/SCP-KNL-vectorization.pdf)
- <https://www.sciencedirect.com/science/article/pii/B9780128091944000090>



Exercise 1

Compile the simple_dp.c and simple_sp.c codes (Aulaweb – Vect Examples) with the following combinations

- `icc -qopenmp -no-vec simple_XX.c`
- `icc -qopenmp simple_XX.c`
- `icc -qopenmp -xHost simple_XX.c`
- What are the timings?
- Do you notice anything about the value of sum with non-vectorized and vectorized codes ?
- And what happens if you add, after the main loop?

```
for(i=1;i<VECSIZE;i++) {  
    a[i]=a[i]+a[i-1];  
}
```

Solution

	Non vect	Vect	AVX-512	Sum non-vect	Sum vect
SP	4.641423	0.145412	0.036836	1119.203491210938	1100.000488281250
DP	4.643434	0.290392	0.073117	1100.000000058394	1100.000000000005

- 1) $T(SP) = T(DP)/2$
- 2) Vectorized code might give slightly different results because some non-vectorized instructions act on different representation (e.g. 80-bit floating point internally instead of 64-bit for double precision)
- 3) LOOP BEGIN at simple2_sp.c(27,3)
 - remark #15344: loop was not vectorized: vector dependence prevents vectorization
 - remark #15346: vector dependence: assumed FLOW dependence between $a[i]$ (28:5) and $a[i-1]$ (28:5)LOOP END

Floating point sum

- You must move the decimal point of the number that has the smaller exponent so as to align it with the larger one.

- Let us consider, for example, $9,999 \times 10^1 + 1,610 \times 10^{-1}$

- Step 1: alignment

$$0,01610 \times 10^1 + 9,999 \times 10^1$$

- Step 2: sum

$$10,0151 \times 10^1$$

- Step 3: conversion to the scientific normalised form

$$1,00151 \times 10^2$$

- BUT what happens if I have only 4 digits?

In step 2 i will use $0,016 \times 10^1$, tso the result will be $1,002 \times 10^2$

Problems

La limitatezza della precisione dell'aritmetica dei calcolatori porta ad avere dei problemi con le proprietà delle operazioni aritmetiche.

Ad esempio, la somma in virgola mobile non è associativa, cioè, in generale, non è vero che $x+(y+z) = (x+y)+z$

I problemi nascono quando si vogliono sommare due numeri molto grandi di segno opposto, con altro un numero molto piccolo

$$x = 1.5_{10} \cdot 10^{38}$$

$$y = -1.5_{10} \cdot 10^{38} \quad \text{con } x, y, z \text{ espressi in singola precisione}$$

$$z = 1.0_{10}$$

$$\begin{aligned} x + (y+z) &= -1.5 \cdot 10^{38} + (1.5 \cdot 10^{38} + 1.0) \\ &= -1.5 \cdot 10^{38} + 1.5 \cdot 10^{38} = 0.0_{10} \end{aligned}$$

Dovrei spostare l'1 di 38 posizioni a destra

$$\begin{aligned} (x+y)+z &= (-1.5 \cdot 10^{38} + 1.5 \cdot 10^{38}) + 1.0 \\ &= 0.0 + 1.0 = 1.0_{10} \end{aligned}$$

Arithmetic accuracy

- L'hardware utilizza bit aggiuntivi per i calcoli, onde poter arrotondare in modo più accurato
- Lo standard prevede due bit aggiuntivi a destra della mantissa, la guardia e l'arrotondamento
 - Provare cosa succede sommando $2,56$ con $2,34 \cdot 10^2$ con o senza due cifre aggiuntive
- Vedrete questi aspetti in un corso successivo

Exercise 2 and 3

- `icc -xHost -qopenmp alias.c`
Addition took 0.072852 seconds
- `icc -xHost -qopenmp alias_func.c alias_main.c`
Addition took 0.276665 seconds
- Try to solve the issues with `-guide`
- Analyze and try to speedup `forward.c` by removing the dependencies

Solutions

- Exercise 2: it is sufficient to compile with -fargument-noalias
But nothing changes...

Try with –no-vec to see the expected non vectorized execution time and -qopt-report=5 -qopt-report-phase=vec to see that the compiler produced two alternative versions of the loop

- One for the case update_final(a,b,VECSIZE,ITERATIONS);
- The second for update_final(b,b,VECSIZE,ITERATIONS);

```
LOOP BEGIN at alias_func.c(9,5)
<Multiversioned v2>
  remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END
```

- Exercise 3: split the loop but pay attention to the final sum!

To summarize

1. use a compiler switch for auto-vectorization (and hope it vectorizes)
2. interact with the compiler to get hints
-qopt-report=5 -qopt-report-phase=vec
and give it your compiler hints (and hope it vectorizes)
3. code explicitly to vectorize the loop
4. disregard vectorization and use explicit parallelism

C99 *restrict* keyword IVDEP (ignore assumed vector dependencies)

```
void v_add (float *restrict c,
            float *restrict a,
            float *restrict b)
{
    for (int i=0; i<= MAX; i++)
        c[i]=a[i]+b[i];
}
```

```
void v_add (float *c,
            float *a,
            float *b)
{
    #pragma ivdep
    for (int i=0; i<= MAX; i++)
        c[i]=a[i]+b[i];
}
```

```
void v_add (float *c,
            float *a,
            float *b)
{
    #pragma omp simd
    for (int i=0; i<= MAX; i++)
        c[i]=a[i]+b[i];
}
```

Profiling and Tuning

- Actually, the first step for improving the performance of a program is the profiling, to understand where are the hotspots
- Many Tools
 - Program code
 - GNU gprof
 - Nvidia Nsight <https://developer.nvidia.com/nsight-systems>
 - Intel VTUNE <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>
- Vtune tutorials
<https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/introduction/tutorials-and-samples.html>
- From the profiling to support for the optimization of the code
 - Intel Advisor
<https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html>

Intel® VTune™ Profiler

Locate performance bottlenecks fast. Advanced sampling and profiling techniques quickly analyze your code, isolate issues, and deliver insights for optimizing performance on modern processors.



Single Thread

Optimize single-threaded performance.



Multithread

Effectively use all available cores.



System

See a system-level view of application performance.



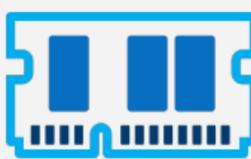
Media & OpenCL™ Applications

Deliver high-performance image and video processing pipelines.



HPC & Cloud

Access specialized, in-depth analyses for HPC and cloud computing.



Memory & Storage Management

Diagnose memory, storage, and data plane bottlenecks.



Analyze & Filter Data

Mine data for answers.



Environment

Fits your environment and workflow.

Intel® Advisor

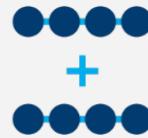
Design and optimize high-performing code for modern computer architectures. Effectively use more cores, vectorization, memory, and heterogeneous processing.



Roofline Analysis

Optimize your application for memory and compute.

[Read More →](#)



Vectorization Optimization

Enable more vector parallelism and improve its efficiency.

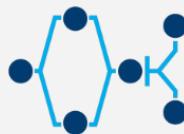
[Read More →](#)



Thread Prototyping

Model, tune, and test multiple threading designs.

[Read More →](#)



Build Heterogeneous Algorithms

Create and analyze data flow and dependency computation graphs.

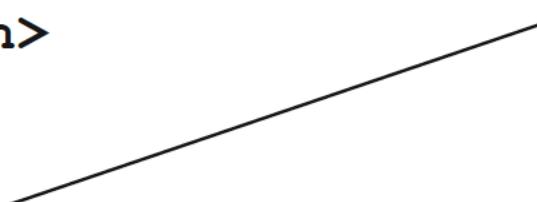
Timing with program code

For benchmarking, an accurate wallclock timer (end-to-end stop watch) is required:

- `clock_gettime()`, POSIX compliant timing function
- `MPI_Wtime()` and `omp_get_wtime()`, standardized programming-model-specific timing routines for MPI and OpenMP

```
#include <stdlib.h>
#include <time.h>

double getTimeStamp()
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9;
}
```



Usage:

```
double S, E;
S = getTimestamp();
/* measured code region */
E = getTimestamp();
return E-S;
```

Sampling-based runtime profile with perf

Instrumentation based with gprof

Compile with **-pg** switch:

```
icc -pg -O3 myfile1.c
```

Execute the application. During execution a file
gmon.out is generated.

Analyze the results with:

```
gprof ./a.out | less
```

The output contains three parts: A flat profile, the call graph, and an alphabetical index of routines.

The flat profile is what you are usually interested in.

Example

Each sample counts as 0.01 seconds.						
% time	seconds	cumulative seconds	self calls	self s/call	total s/call	name
66.86	26.14	26.14	502	0.05	0.05	ForceLJ::compute(Atom&, Neighbor&, Comm&, int)
30.77	38.17	12.03	26	0.46	0.46	Neighbor::build(Atom&)
1.43	38.73	0.56	1	0.56	38.46	Integrate::run(Atom&, Force*, Neighbor&, Comm&, Thermo&, Timer&)
0.36	38.87	0.14	2850	0.00	0.00	Atom::pack_comm(int, int*, double*, int*)
0.15	38.93	0.06	2850	0.00	0.00	Atom::unpack_comm(int, int, double*)
0.13	38.98	0.05	26	0.00	0.00	Atom::pbc()
0.10	39.02	0.04				__intel_ssse3_rep_memcpy
0.08	39.05	0.03	25	0.00	0.00	Atom::sort(Neighbor&)
0.08	39.08	0.03	1	0.03	0.03	create_atoms(Atom&, int, int, int, double)
0.05	39.10	0.02	26	0.00	0.00	Comm::borders(Atom&)
0.00	39.10	0.00	1221559	0.00	0.00	Atom::pack_border(int, double*, int*)
0.00	39.10	0.00	1221559	0.00	0.00	Atom::unpack_border(int, double*)
0.00	39.10	0.00	131072	0.00	0.00	Atom::addatom(double, double, double, double, double, double)
0.00	39.10	0.00	1025	0.00	0.00	Timer::stamp(int)
0.00	39.10	0.00	502	0.00	0.00	Thermo::compute(int, Atom&, Neighbor&, Force*, Timer&, Comm&)
0.00	39.10	0.00	500	0.00	0.00	Timer::stamp()
0.00	39.10	0.00	475	0.00	0.00	Comm::communicate(Atom&)
0.00	39.10	0.00	26	0.00	0.00	Comm::exchange(Atom&)
0.00	39.10	0.00	25	0.00	0.00	Timer::stamp_extra_stop(int)
0.00	39.10	0.00	25	0.00	0.00	Timer::stamp_extra_start()
0.00	39.10	0.00	25	0.00	0.00	Neighbor::binatoms(Atom&, int)
0.00	39.10	0.00	7	0.00	0.00	Timer::barrier_stop(int)
0.00	39.10	0.00	1	0.00	0.00	create_box(Atom&, int, int, int, double)
0.00	39.10	0.00	1	0.00	0.00	create_velocity(double, Atom&, Thermo&)

Intel Advisor

- <https://software.intel.com/en-us/advisor-tutorial-roofline-roofline-use-case>

- `icc -g -O2 -std=c++11 roofline.cpp`

- For the analysis here
add `-qopenmp`

and select

```
#define GROUP_4
#define G$_SOA
```

```
*****  
//#define GROUP_1  
-#ifdef GROUP_1  
//#define G1_AOS_SCALAR  
#define G1_SOA_SCALAR  
//#define G1_SOA_VECTOR  
#endif *****  
//#define GROUP_2  
-#ifdef GROUP_2  
#define G2_AOS_SCALAR  
#define G2_SOA_SCALAR  
#define G2_SOA_VECTOR  
#endif *****  
//#define GROUP_3  
-#ifdef GROUP_3  
#define G3_AOS_SCALAR  
//#define G3_SOA_SCALAR  
//#define G3_AOS_VECTOR  
//#define G3_SOA_VECTOR_PMAS  
#endif *****  
#define GROUP_4  
-#ifdef GROUP_4  
#define G4_SOA  
//#define G4_AOSOA  
#endif *****
```

```
#define MAXVALUE 10000000  
#define ARRAY_SIZE_1 1328  
#define REPEAT_1 10000000  
#define REPEAT_2 2000  
#define REPEAT_2 30000000  
#define UNROLL_COUNT_2
```

Group 1: $X = Y_a + Y_b$

Group 2: $X = Y_a + Y_b + Y_b$

Group 3: $X = Y_a + Y_a + Y_b + Y_b + Y_b$

Group 4: $X = Y_a + Y_a + Y_b + Y_b$

Analysis

```
name@machine:~$ advixe-cl --collect survey --project-dir ./outputfolder -- yourprogram  
name@machine:~$ advixe-cl --collect tripcounts -flops-and-masks --project-dir ./outputfolder -- yourprogram
```

Command line

GUI

You can recompile your code many times and repeat the analyses keeping the name of the executable



In details... roofline

Load at least these scripts

```
. /opt/intel/parallel_studio_xe_2019/bin/psxevars.sh intel64  
. /opt/intel/parallel_studio_xe_2019/advisor_2019/advixe-vars.sh
```

advixe-cl -collect survey -- executable

advixe-cl -report survey -format=xml -- executable (or csv, text)

advixe-cl -collect tripcounts -flops -- executable

advixe-cl -collect roofline --project-dir=. -- executable

advixe-cl -report roofline -project-dir=. -report-output=./filename.html

advixe-cl --help collect

advixe-cl --help report

<https://software.intel.com/content/www/us/en/develop/articles/tutorial-intel-advisor-vectorization-sample-readme.html>

In depth analysis

advixe-cl -collect survey -- executable

advixe-cl -collect tripcounts -flop -- executable

advixe-cl -collect dependencies -- executable

advixe-cl -collect map – executable

Then analyse on your local PC

scp e000/ Vectorization_Advisor.advixeproj ...



Welcome e000 X

Elapsed time: 94.61s



Vectorized

Not Vectorized



FILTER: All Modules ▾ All Sources ▾

Loops And Functions ▾

All Threads ▾



Smart Mode



INTEL ADVISOR 2017

Summary Survey & Roofline Refinement Reports

Higher instruction set architecture (ISA) available

Consider recompiling your application using a higher ISA.

SURVEY

Performance (GFLOPS)

 Use Single-Threaded Roofs

35.97

L1 Bandwidth (single-threaded): 92.94 GB/sec?

L2 Bandwidth (single-threaded): 54.93 GB/sec?

DRAM Bandwidth (single-threaded): 8.56 GB/sec?

0.19

0.045

Self Elapsed Time: 94.115 s Total Time: 94.115 s

DP Vector Add Peak (single-threaded): 17.61 GFLOPS?

Scalar Add Peak (single-threaded): 2.05 GFLOPS?

main roofline.cpp:310
 Performance: 1.91 GFLOPS
 L1 Arithmetic Intensity: 0.12 FLOP/Byte
 Self Elapsed Time: 94.115 s
 Total Time: 94.115 s

0.25

Arithmetic Intensity (FLOP/Byte)

[Source](#) [Top Down](#) [Code Analytics](#) [Assembly](#) [💡 Recommendations](#) [❓ Why No Vectorization?](#)

File: roofline.cpp:310 main

Line	Source	Total Time	%	Loop/Function Time	%	Traits
304	for (int r = 0; r < REPEAT_2; r++)					
305	{					
306	for (int j = 0; j < 2; j++)					
307	{					
308	#pragma nounroll					
309	#pragma omp simd simdlen(VECTOR_LENGTH)					
310	for (int i = 0; i < ARRAY_SIZE_2 / 2; i++)	24,000s	94,115s			
311	{					
312	AoSoA_X[(j * (ARRAY_SIZE_2 / 2)) + i] = AoSoA_Y[j].a[i] + AoSoA_Y[j].a[i]	35,480s				
313	+ AoSoA_Y[j].b[i] + AoSoA_Y[j].b[i];	34,635s				
314	}					
315	}					
Selected (Total Time):			24,000s			

Program metrics

Elapsed Time	94,61s		
Vector Instruction Set	SSE, SSE2	Number of CPU Threads	1
Total GFLOP Count	180,00	Total GFLOPS	1,90
Total Arithmetic Intensity <small>?</small>			0,12

Loop metrics

Metrics	Total	
Total CPU time	94,60s	<div style="width: 100%;"><div style="width: 100%;">100,0%</div></div>
Time in 1 vectorized loop	94,12s	<div style="width: 99,5%; background-color: #0070C0;"></div>
Time in scalar code	0,48s	<div style="width: 2%; background-color: #0070C0;"></div>

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency <small>?</small>	2,67x	<div style="width: 67%; background-color: #FFA500;">67%</div>	<div style="width: 33%; background-color: #D3D3D3;">33%</div>
---	-------	---	---



Welcome e000 X

Vectorization Workflow

Threading Workflow

OFF Batch mode

Run Roofline

▶ Collect

1. Survey Target

▶ Collect

1.1 Find Trip Counts and FLOPS

▶ Collect

 Trip Counts FLOPS

Mark Loops for Deeper Analysis

Select loops in the Survey Report for Dependencies and/or Memory Access Patterns analysis.

1 loop is marked

2.1 Check Dependencies

▶ Collect

2.2 Check Memory Access Patterns

▶ Collect

↻ Re-finalize Survey

Elapsed time: 94,29s Vectorized Not Vectorized FILTER: All Modules ▾ All Sources ▾

Summary Survey & Roofline Refinement Reports

INTEL ADVISOR 2017

Site Location Loop-Carried Dependencies Strides Distribution Access Pattern Max. Site Footprint Site Name Recommendations

[loop in main at roofline.cpp:... No information available 100% / 0% / 0% All unit strides 240B loop_site_4

```

308     #pragma nounroll
309     #pragma omp simd simdlen(VECTOR_LENGTH)
310     for (int i = 0; i < ARRAY_SIZE_2 / 2; i++)
311     {
312         AoSoA_X[(j * (ARRAY_SIZE_2 / 2)) + i] = AoSoA_Y[j].a[i] + AoSoA_Y[j].a[i]

```

Memory Access Patterns Report Dependencies Report Recommendations

ID		Stride	Type	Source	Nested Function	Variable references	Access Footprint	Modules	Site Name	Access Type
P1		1	Unit stride	roofline.cpp:312		AoSoA_Y	240B	libc.so.6; roofline	loop_site_4	Read

```

310     for (int i = 0; i < ARRAY_SIZE_2 / 2; i++)
311     {
312         AoSoA_X[(j * (ARRAY_SIZE_2 / 2)) + i] = AoSoA_Y[j].a[i] + AoSoA_Y[j].a[i]
313                     + AoSoA_Y[j].b[i] + AoSoA_Y[j].b[i];
314     }

```

+P2		1	Unit stride	roofline.cpp:312		AoSoA_X	240B	libc.so.6; roofline	loop_site_4	Write
-----	--	---	-------------	------------------	--	---------	------	---------------------	-------------	-------

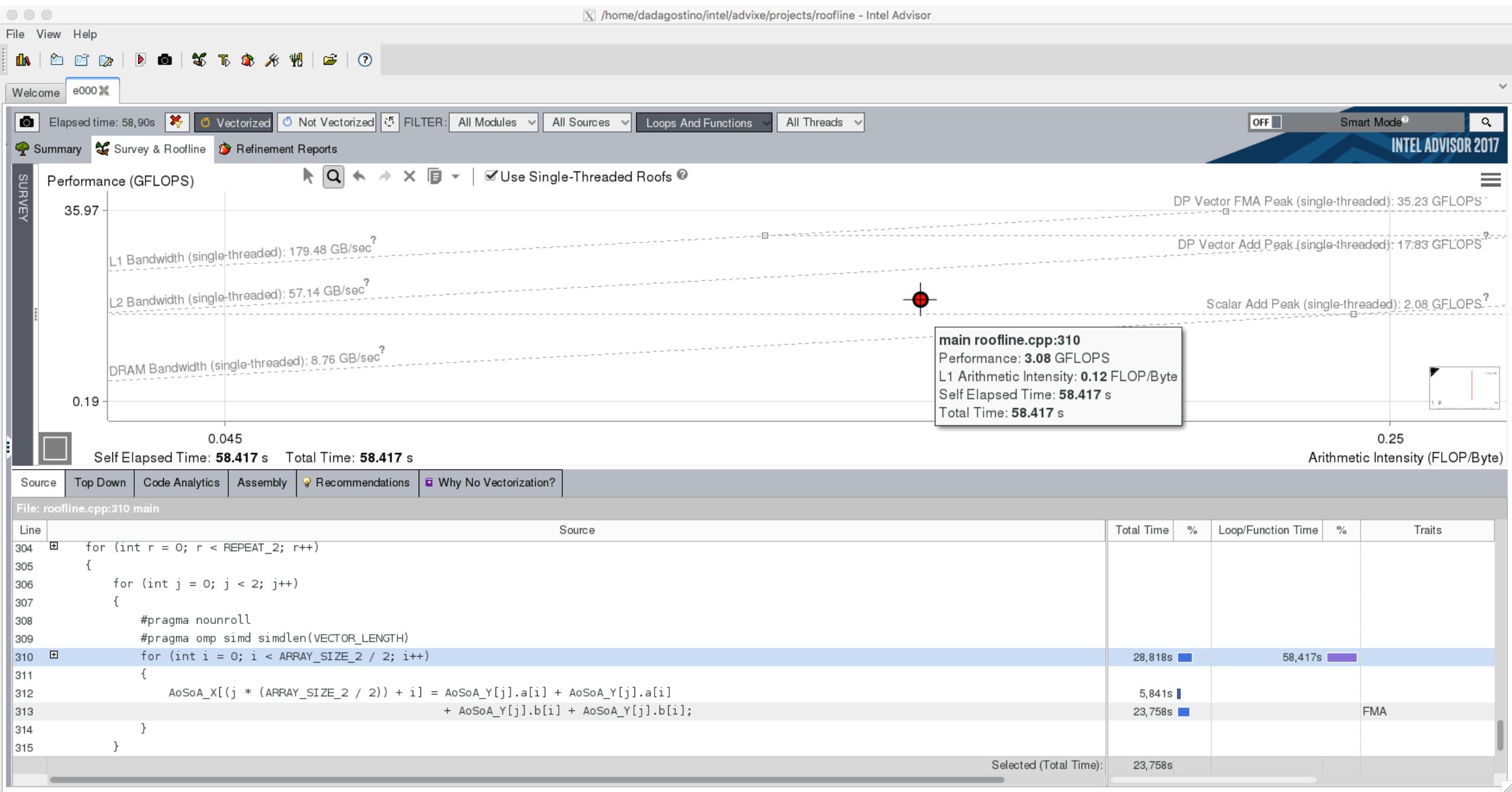
+P3		1	Unit stride	roofline.cpp:313		AoSoA_Y	240B	libc.so.6; roofline	loop_site_4	Read
-----	--	---	-------------	------------------	--	---------	------	---------------------	-------------	------

+P4			Parallel site information	roofline.cpp:310				roofline	loop_site_4	
-----	--	--	---------------------------	------------------	--	--	--	----------	-------------	--

Very time consuming!



```
icc -g -O2 -std=c++11 -qopenmp -xHost -qopt-report=5 -qopt-report-phase=vec roofline.cpp -o roofline
```



For the others cases follow the video tutorial

<https://software.intel.com/en-us/videos/roofline-analysis-in-intel-advisor-2017>

<https://www.youtube.com/watch?v=h2QEM1HpFgg>

Add efficiency to MatVector

The screenshot shows the Intel Advisor interface for the project "MatVector". The main window displays a summary of performance metrics and a detailed analysis of loops and functions.

Summary: Elapsed time: 51,39s. Vectorized: 2 loops, Not Vectorized: 1 loop.

Higher instruction set architecture (ISA) available: Consider recompiling your application using a higher ISA.

Function Call Sites and Loops:

Loop	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	FLOPS And AVX-512 Mask Usage	Trip Counts	Instructions
						Vecto... Gain ... VL (V... Self GFLOPS Self AI Mask Utilization	Average Call Count Total		
[loop in matvec at Multiply.c:55]	2 Assumed dependence	47,638s	47,638s	Scalar	vector dependence pre...	0,424 0,083	50 101000000		
[loop in matvec at Multiply.c:45]	1 Opportunity for...	3,720s	51,358s	Scalar	outer loop was not auto...	0,054 0,042	101 1000000		
matvec		0,020s	51,378s	Function					1000000
_start		0,000s	51,378s	Function					1
[loop in main at Driver.c:145]		0,000s	51,378s	Scalar	loop with function call n...	0,031	1000000 1		
main		0,000s	51,378s	Function		0,021			1 Shared

Recommendations: Why No Vectorization?

Vector dependence prevents vectorization:

Cause: The compiler detected or assumed a vector dependence in the loop.

C++ Example:

```
int foo(float *A, int n) {
    int inx = 0;
    float max = A[0];
    int i;
    for (i=0;i < n;i++) {
        if (max < A[i]) {
            max = A[i];
            inx = i*i;
        }
    }
    return inx;
}
```

Fortran Example:

```
integer function foo(a, n)
    implicit none
    integer, intent(in) :: n
    real intent(inout) :: a(n)
```

The Process for Vectorizing Your Application

- Measure Baseline Release Build Performance
(essentially compile the code without optimization and –g)
- Determine Hotspots
(with a profiler or manually instrumenting the code)
- Get Advice
(using compiler's features, Intel Advisor...)
- Implement Vectorization Recommendations
- Repeat!

<https://software.intel.com/en-us/articles/vectorization-toolkit/>

OpenMP

A collage of several tutorials

What is OpenMP?

- OpenMP = **Open Multi-Parallelism**
- It is an API to explicitly direct *multi-threaded shared-memory parallelism*.
- Comprised of three primary API components
 - **Compiler directives**
 - These define which parts of the code are run in parallel
 - **Run-time library routines**
 - Support routines such as identifying the thread ID
 - **Environment variables**
 - Used to adjust the runtime behaviour of the parallel application



The OpenMP* Common Core

Yun (Helen) He

LBL/NERSC

Ruud van der Pas

Oracle Linux Engineering

Tim Mattson

Intel

Alice Koniges

Maui HPC Center

The first version of the “Common Core” slides was created by Tim Mattson, Intel Corp. Many others (e.g. Barbara Chapman, Mark Bull, Larry Meadows, ...) have contributed to these slides.

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.



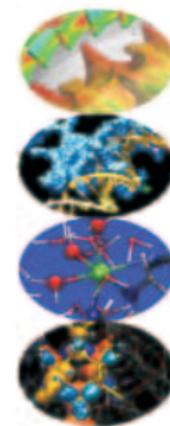
Introduction to OpenMP

Claudia Truini - c.truini@cineca.it

Vittorio Ruggiero - v.ruggiero@cineca.it

Mariella Ippolito - m.ippolito@cineca.it

SuperComputing Applications and Innovation Department





UNIVERSITÀ
DEGLI STUDI
FIRENZE



Parallel Computing

Prof. Marco Bertini

Outline

- **Introduction**
- Why Common Core?
- OpenMP Basics
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

OpenMP is ideally suited for modern architectures

Portable, and portable, and portable

Memory and threading model map naturally

Tight integration with accelerator support

Lightweight

Mature, yet continues to evolve

Widespread support and still on the rise

OpenMP Overview



OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

C\$OMP PARALLEL COPYIN(/blk/)

C\$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

What OpenMP Really Is

- OpenMP is an industry standard API for C, C++, and Fortran to support shared memory, vectorization, and heterogeneous device programming
- Specifications are defined by the OpenMP Architecture Review Board (ARB)
 - 35+ members, including industry, government labs, and academia
 - Available in major commercial and open source compilers

The OpenMP ARB Mission

To standardize directive-based multi-language high-level parallelism that is performant, productive, and portable

Support Continues to Increase!

OpenMP
TM

 AMD Greg Rodgers	 Argonne National Laboratory Kalyan Kumaran	 ARM Graham Hunter	 ASC/Lawrence Livermore National Laboratory Bronis R. de Supinski	 Barcelona Supercomputing Center Xavier Martorell	 NEC Shin-ichi Okano	 NVIDIA Jeff Larkin	 Oak Ridge National Laboratory Oscar Hernandez	 Red Hat Torvald Riegel	 RWTH Aachen University Dieter an Mey
 Bristol University Simon McIntosh-Smith	 Brookhaven National Laboratory Vivek Kale	 cOMPunity Yonghong Yan	 CRAY Deepak Eachempati	 Edinburgh Parallel Computing Centre Mark Bull	 Sandia National Laboratory Stephen Oliver	 Stony Brook University Dr. Barbara Chapman	 SUSE Michael Matz	 Texas Advanced Computing Center Kent Milfield	 Texas Instruments Gaurav Mitra
 Fujitsu Naoki Sueyasu	 IBM Kelvin Li	 INRIA Olivier Aumage	 Intel Xinmin Tian	 Lawrence Berkeley National Laboratory Helen He	 The University of Manchester Antoniu Pop	 University of Delaware Sunita Chandrasekaran			
 Leibniz Supercomputing Centre Volker Weining	 Los Alamos National Laboratory Jamal Mohd-Yusof	 Maui High Performance Computing Center Alice Koniges	 Micron Randy Meyer	 NASA Henry Jin					

OpenMP is widely supported by the industry, as well as the research and academic community

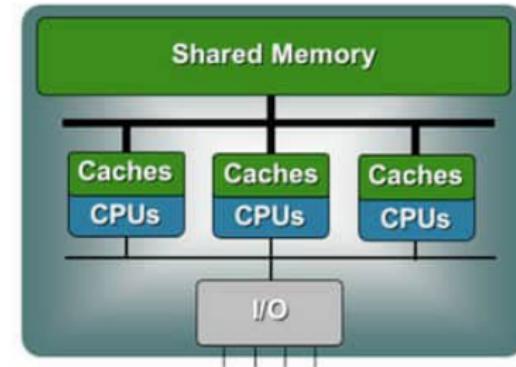
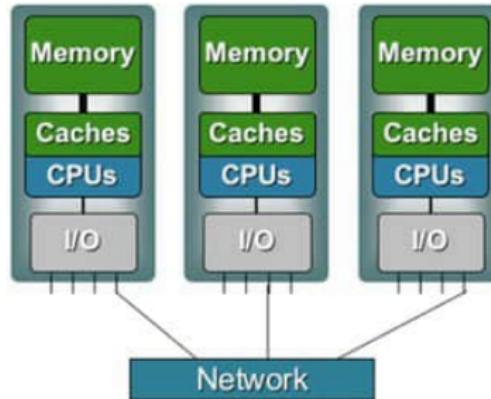


<http://www.openmp.org>

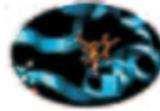


Shared memory architectures

- All processors may access the whole main memory

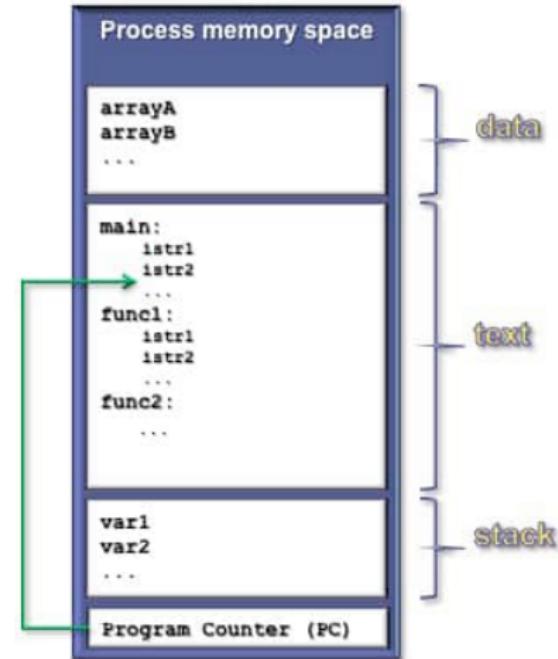


- Non-Uniform Memory Access
 - Memory access time is non-uniform
- Uniform Memory Access
 - Memory access time is uniform



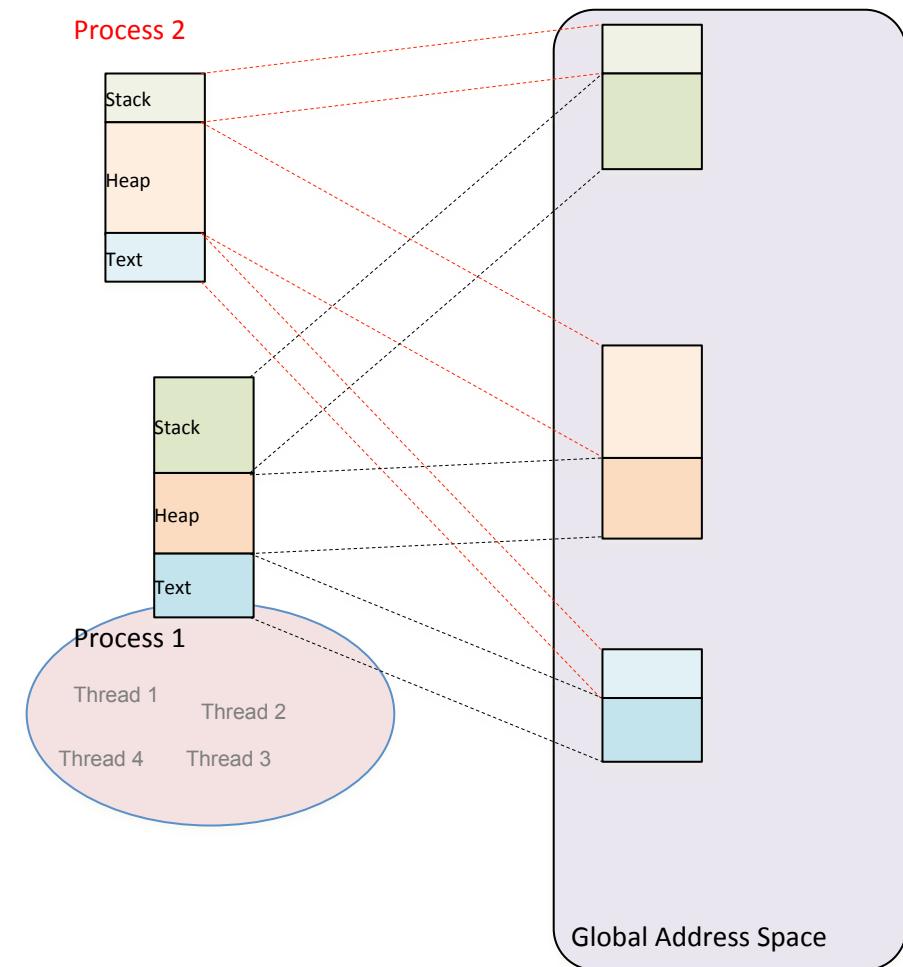
Process and thread

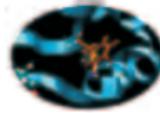
- A process is an instance of a computer program
- Some information included in a process are:
 - Text
 - Machine code
 - Data
 - Global variables
 - Stack
 - Local variables
 - Program counter (PC)
 - A pointer to the instruction to be executed



Operating System Memory Model

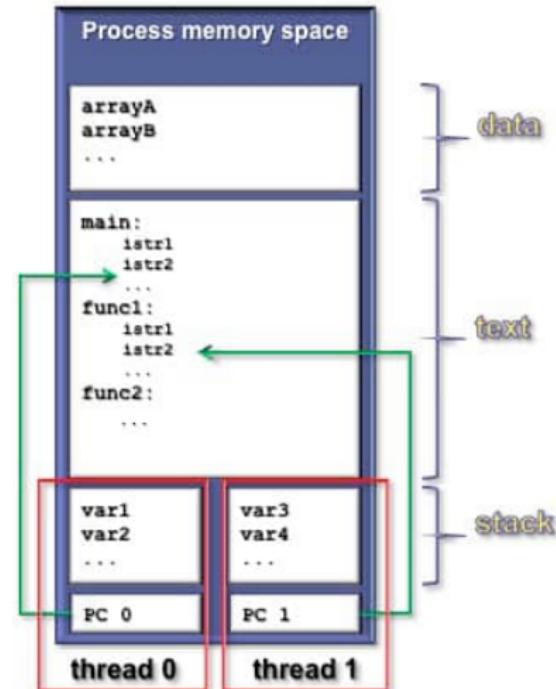
- A process, owns a lot of state information including the memory, file handles, etc.
- Operating systems provide a separate address space for each process
- One process cannot see the memory of another process
- Memory within a process is managed separately between
 - text and data segment - static read only areas for code and data known at compile time
 - the heap - an area of memory managed by the operating system for dynamic data allocation
 - the stack - a piece of memory managed by the process itself
- Multiple threads launched from the same process share the same address space
- Threads, launched by the process, share the state information, **including memory**, of the launching process and so are considered *light weight tasks*





Multi-threading

- The process contains several concurrent execution flows (threads)
 - Each thread has its own program counter (PC)
 - Each thread has its own private stack (variables local to the thread)
 - The instructions executed by a thread can access:
 - the process global memory (data)
 - the thread local stack



Why use Threading?

- Exploit *more* parallelism to increase scaling and performance
 - SMPD parallelism
 - E.g. Distributed memory (e.g. MPI) tasks on one spatial dimension, threads (e.g. OpenMP) on another.
 - Or, Reduce number of distributed memory tasks thus reducing communications overhead.
 - Functional parallelism
 - Threads executing different tasks, perhaps on the same data, perhaps not.
 - Improve load balance via constructs that enable *work stealing*.
 - Reduce memory overheads from many separate processes by using lightweight threads which share memory

Alternatives 1 - Pthreads

- An alternative model for combining threads with message passing is to use the POSIX Threads interface and library Pthreads
- Most OpenMP implementations are built on top of Pthreads !!
 - ... in fact most threading libraries of any type are built on Pthreads
- Pthreads provides the ability to dynamically create threads which are launched to run a specific task
- Pthreads provides finer grained control than OpenMP
 - Uses **mutex** and **condition** variables for synchronisation
- The disadvantages of Pthreads compared to OpenMP are
 - There is no Fortran interface in the standard
 - Although IBM did produce a Fortran interface for their XLF compiler
 - You have to manage (re-create) any worksharing yourself
 - It is a low level interface
 - As MPI is often referred to as a low level interface this might not be a problem
 - You will typically take many more lines of coding using Pthreads than OpenMP
 - OpenMP is normally a more *productive* way of coding for numerical codes

Alternatives 2 - TBB

- Intel Thread Building Blocks (TBB) is a C++ template library that adds parallel programming for C++ programmers.
 - Not applicable to Fortran or C codes
- “Extends” C++ by adding a set of parallel keywords
 - These “extensions” are not actually real changes to the languages
 - As a template library, any standard-conforming C++ compiler can use TBB
- Algorithms can be parallelised in a number of ways
 - `parallel_for`, `parallel_reduce`, `parallel_while` etc.
- TBB adds containers, locks, a task scheduler etc.
- TBB was built out of positive user experiences from OpenMP, and the desire to provide an object-oriented, template based approach to parallelism in C++

TBB vs OpenMP

- Note – Intel not only provides TBB but it also has many members, directors and officers of the OpenMP forum
- Here are some suggestions or tips provided on the TBB website
 - “Everyone should use OpenMP as much as they can. It is easy to use, it is standard, it is supported by all major compilers, and it exploits parallelism well.”
 - “OpenMP [*is*] the standard way to do parallelism in C and Fortran.”
 - “Use OpenMP if the parallelism is primarily for bounded loops over built-in types, or if it is flat do-loop centric parallelism. ... It can be very challenging to match OpenMP performance with TBB for such problems. It is seldom worth the effort to bother – just use OpenMP.”
 - “Should I expect TBB to outperform OpenMP and MPI?
No, TBB may offer a competitive alternative but in general TBB exists to help where OpenMP cannot, and to be far easier to program than MPI.”
 - ”OpenMP and MPI continue to be good choices in High Performance Computing applications; TBB has been designed to be more conducive to application parallelization on client platforms such as laptops and desktops, going beyond data parallelism ...”
- TBB might be a good choice for C++ programmers if their code does not fit the standard data-parallel model
- Note also that TBB and OpenMP can coexist

So Why OpenMP ?

- OpenMP is ubiquitous
 - Available with almost all compilers
 - gcc provides OpenMP support
 - Therefore anyone can get hold of an OpenMP enabled C/Fortran compiler for free
 - GCC 4.7 provides full support for OpenMP 3.1 standard
 - Vendor tuned implementations available
- Easy interface available for incremental parallelism
- Is the best fit for data-parallel codes
- Is being updated to incorporate methods for modern programming methods and technologies
 - Task parallel features were added in OpenMP 3.0
 - Current roadmap suggests that accelerator directives will be added in OpenMP 4.0

Outline

- Introduction
- **Why Common Core?**
- OpenMP Basics
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

Choice of Programming Models for Modern HPC Systems

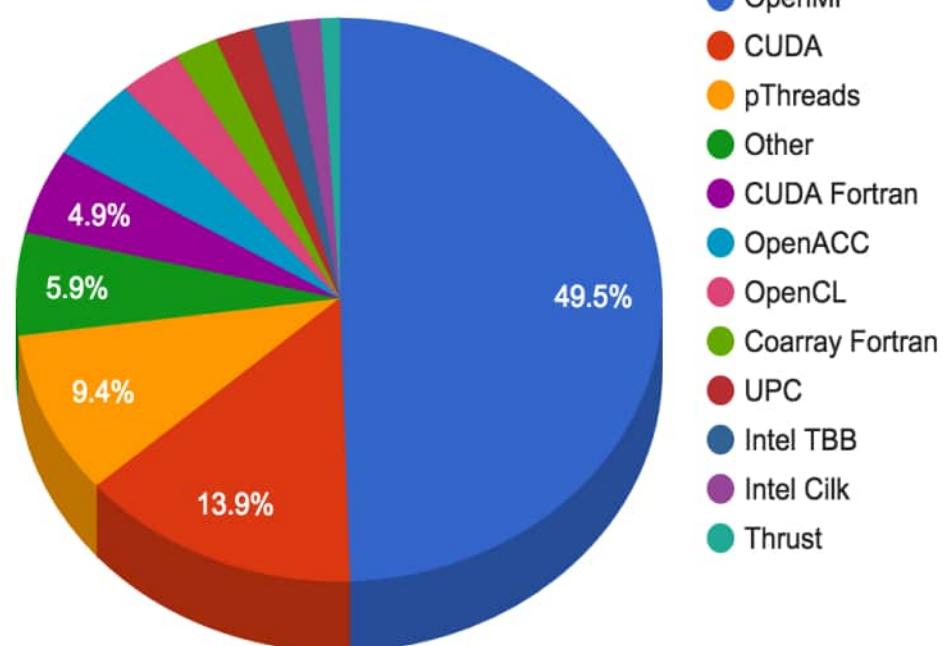


- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for **shared memory** or intra-node (intra means within)
 - Trend: multi-socket nodes with rapidly increasing core counts
 - Trend: accelerators
- Hybrid Programming (**MPI+X**) is when we use a solution with different programming models for inter vs. intra-node parallelism

What is X in “MPI+X”?
OpenMP is about 50% in 2015 at NERSC

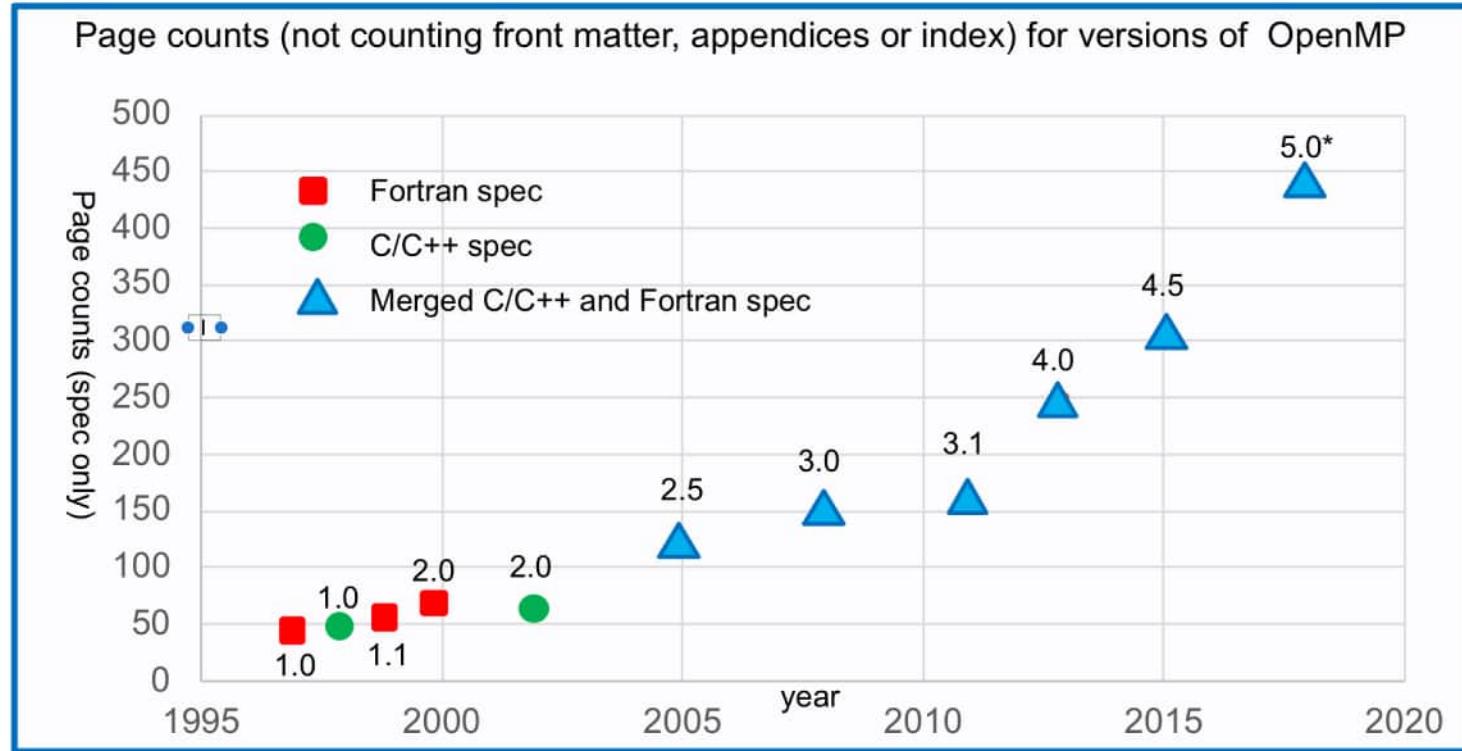
Update in 2017: OpenMP increases to 75%

The National Energy Research Scientific Computing Center (NERSC) is the primary scientific computing facility for the Office of Science in the U.S. Department of Energy.



Why the Common Core?

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



* Does not include the tools interface added with OpenMP 5.0 which pushes the page count to 618

The complexity of the full spec is overwhelming, so we focus on the 21 items most OpenMP programmers restrict themselves to, the so called “OpenMP Common Core”

The OpenMP Common Core: Most OpenMP Programs Only Use These 21 Items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(None)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

Common Core Reference Guide

<https://www.openmp.org/specifications/>

OpenMP API Common Core

OpenMP.org openmp.org

Common Core

Page 1

Directives and Constructs

parallel construct

parallel [2.6] [2.5]
Founds parallel threads and starts parallel execution.

single [2.4.2] [2.7.8]
Specifies that the associated structured block is executed by only one of the threads in the team. Has implied barrier unless turned off with nowait.

Worksharing constructs

single [2.4.2] [2.7.8]
Specifies that the associated structured block is executed by only one of the threads in the team. Has implied barrier unless turned off with nowait.

for [2.3.2] [2.7.8]
Specifies the iterations of associated loops will be executed in parallel by threads in the team. Has implied barrier unless turned off with nowait.

combined [2.13.1] [2.11.1]
Specifies a parallel construct containing one worksharing-loop construct with one or more associated loops.

Environment Variable

OMP_NUM_THREADS [4.2]
Sets default number of threads to request for parallel regions

Tasking constructs

task [2.10.1] [2.11.1]
Defines an executable block.

omp_parallel_task [clause] [, clause] ...
structured-block

omp_parallel [clause] [, clause] ...
structured-block

omp_parallel_for [clause] [, clause] ...
structured-block

omp_parallel_for [clause] [, clause] ...
structured-block

taskwait [2.11.3] [2.11.4]
Specifies wait on the completion of child tasks of the current task.

Synchronization constructs

critical [2.11.1] [2.11.2]
Restricts execution of the associated structured block to a single thread at a time.

omp_critical [clause] [, clause] ...
structured-block

omp_parallel_critical [clause] [, clause] ...
structured-block

Memory consistency

reduction [op] [2.19.4] [2.19.3]
Reduces values of one or more list items.

reduction [op] [2.19.4] [2.19.3]
Reduces list items to private to each thread or explicit task and assigns them the value the original variable has at the time the construct is encountered.

Clauses

Data Sharing Clauses [2.19.4] [2.19.3]
These clauses apply only to variables whose names are visible in the construct on which the clause appears.

shared[ist]
The items in the list are shared between threads or explicit tasks executing the construct.

private[ist]
Creates a new variable for each item in the list that is private to each thread or explicit task. The private variable is not given an initial value.

firstprivate[ist]
Declares list items to be private to each thread or explicit task and assigns them the value the original variable has at the time the construct is encountered.

Runtime Library Routines

omp_set_num_threads [3.2.1] [3.2.1]
Sets default number of threads to request for parallel regions.

omp_get_thread_num [3.2.4] [3.2.4]
Returns the thread number of the calling thread within the current team.

omp_get_wtime [3.2.1] [3.2.1]
Returns elapsed wall clock time in seconds. Not guaranteed to be globally consistent across all the threads.

© 2018 OpenMP ARB

OMP1118C-CC

OpenMP On Your Laptop (1)

- Mac OS X/Sierra
 - Compiler installed in native XCode doesn't "do" OpenMP
 - Install "homebrew"
 - <https://brew.sh/>
 - Install GNU Compiler Collection
 - \$ brew update
 - \$ brew install gcc (**this will take a long time**)
 - Use the compilers! (**note the command names**)
 - \$ gcc-9 -fopenmp program.c
 - \$ g++-9 -fopenmp program.cpp
 - \$ gfortran-9 -fopenmp program.f90
- Windows
 - Install free version of Visual Studio
 - <https://www.visualstudio.com/downloads/>

OpenMP On Your Laptop (2)



- Linux
 - Install GNU Compiler Collection (C, C++, Fortran)
 - Ubuntu-like: C, C++ already there by default
 - # apt install gfortran
 - CentOS-like
 - # yum install gcc-c++ gcc-gfortran
 - Fedora-like
 - # dnf install gcc-c++ gcc-gfortran
 - Use compilers!
 - \$ gcc -fopenmp program.c
 - \$ g++ -fopenmp program.cpp
 - \$ gfortran -fopenmp program.f90

Outline

- Introduction
- Why Common Core?
- **OpenMP Basics**
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

How Does OpenMP Work?



- Teams of OpenMP threads are created to perform the computation in a code
 - **Work is divided** among the threads, which run on the different cores
 - The threads collaborate **by sharing variables**
 - Threads **synchronize** to order accesses and prevent data corruption
 - **Structured programming** is encouraged to reduce likelihood of bugs
- OpenMP components:
 - Compiler Directives and Clauses
 - Runtime Libraries
 - Environment Variables

OpenMP Basic Definitions: Basic Solution Stack

User layer

End User

Prog.

Directives,
Compiler

OpenMP library

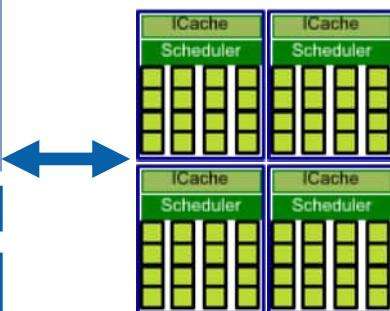
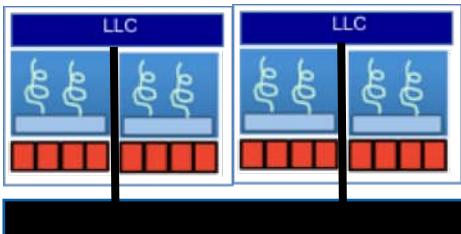
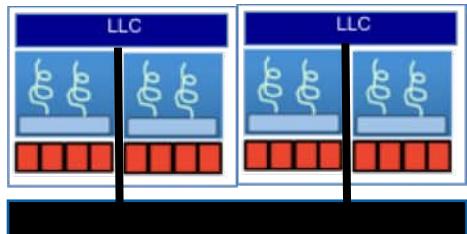
Environment
variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW



Shared address space (NUMA)



IWOMP 2019 Tutorial – The Common Core

SIMD units



GPU cores



OpenMP Basic Definitions: Basic Solution Stack



User layer

End User

Prog.

Directives,
Compiler

OpenMP library

Environment
variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW



Shared address space (SMP)

For the OpenMP Common Core, we focus on Symmetric Multiprocessor Case
i.e. lots of threads with “equal cost access” to memory

OpenMP Basic Syntax

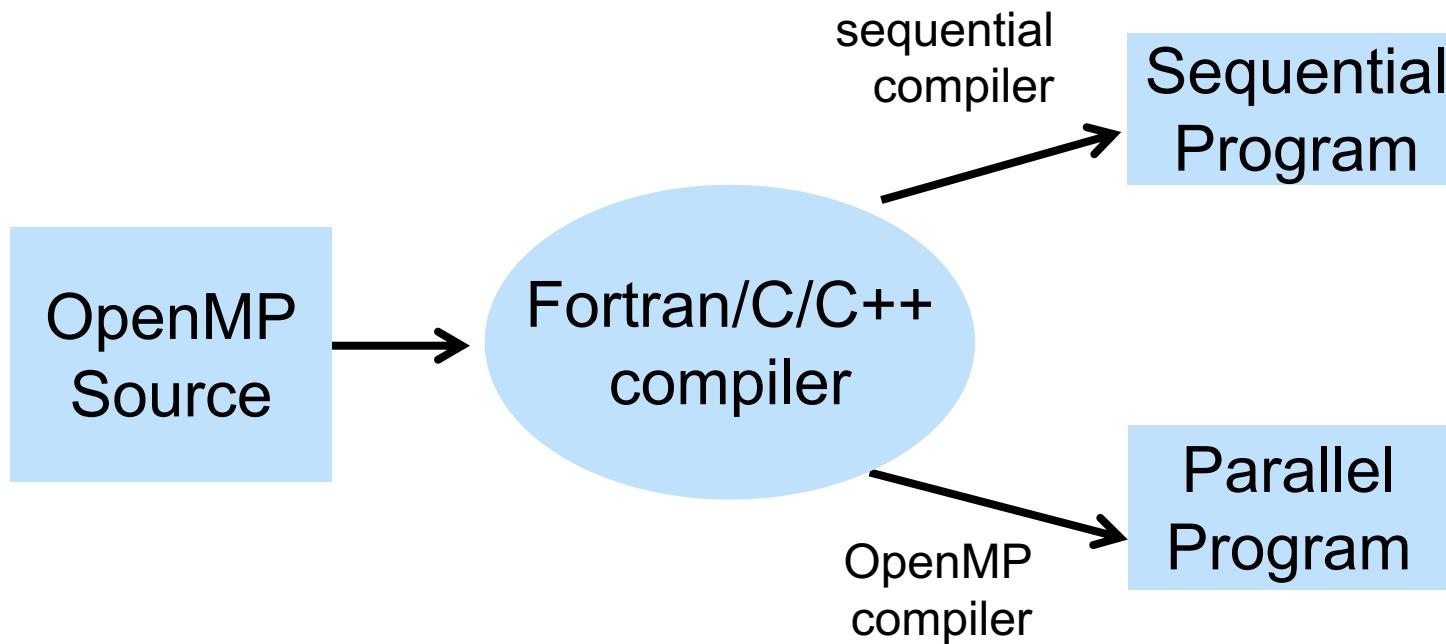


- Most of the constructs in OpenMP are compiler directives.

C and C++	Fortran
Compiler directives	
#pragma omp construct [clause [clause] ...]	!\$OMP construct [clause [clause] ...]
Example	
#pragma omp parallel private(x) { }	!\$OMP PARALLEL !\$OMP END PARALLEL
Function prototypes and types:	
#include <omp.h>	use OMP_LIB

- Most OpenMP constructs apply to a “structured block”.
 - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
 - It’s OK to have an exit() within the structured block.

OpenMP Usage



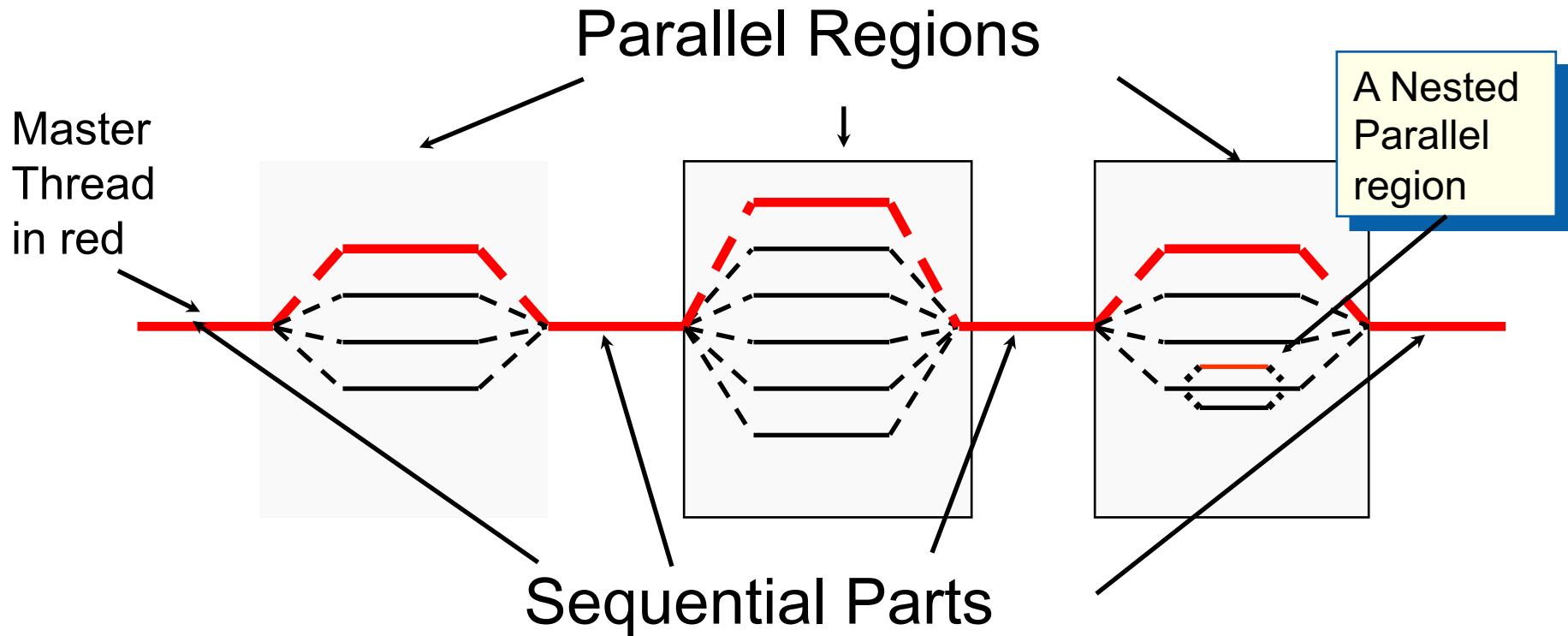
- Maintains single source code for serial and parallel programs.
- Needs special flag to enable OpenMP, such as:
 - Intel compiler: -fopenmp
 - GNU compiler: -fopenmp
 - PGI compiler: -mp
 - Cray compiler: none

OpenMP Programming Model

OpenMP™

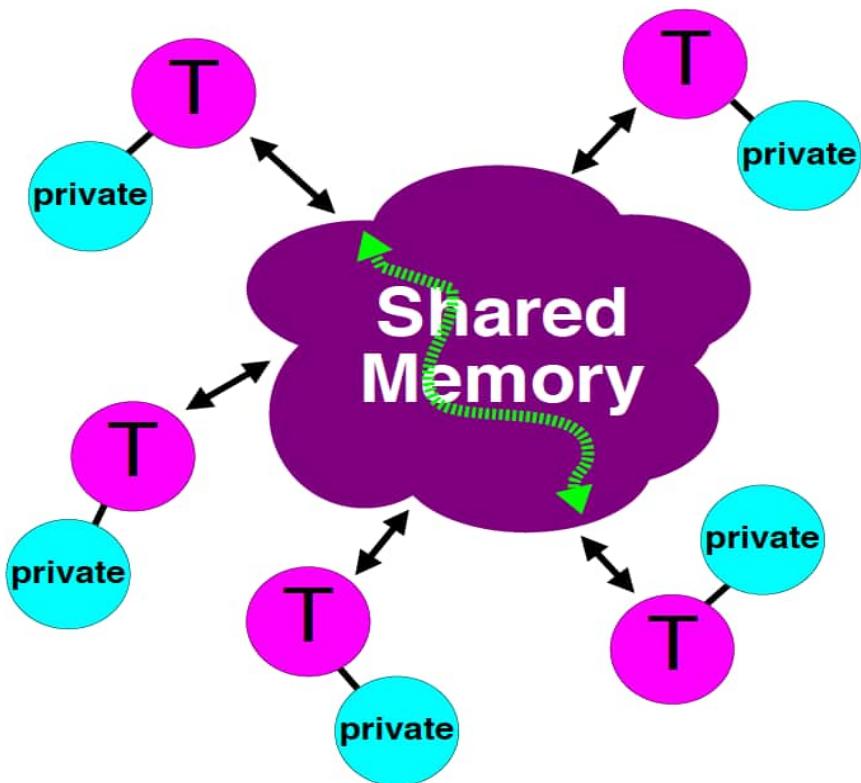
Fork-Join Parallelism:

- ◆ Master thread spawns a team of threads as needed.
- ◆ Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



OpenMP Memory Model

OpenMP™



- ✓ All threads have access to the same, globally shared, memory
- ✓ Data can be shared or private
- ✓ Shared data is accessible by all threads
- ✓ Private data can only be accessed by the thread that owns it
- ✓ Data transfer is transparent to the programmer
- ✓ Synchronization takes place, but it is mostly implicit

Serial vs. OpenMP



Serial

```
void main ()  
{  
    double x[256];  
    for (int i=0; i<256; i++)  
    {  
        some_work(x[i]);  
    }  
}
```

OpenMP

```
#include "omp.h"  
void main ()  
{  
    double x(256);  
#pragma omp parallel for  
    for (int i=0; i<256; i++)  
    {  
        some_work(x(i));  
    }  
}
```

**OpenMP is not just about parallelizing loops!
It offers a lot more**

Advantages of OpenMP



- Simple programming model
 - Data decomposition and communication handled by compiler directives
- Single source code for serial and parallel codes
 - No major redesign of the serial code
- Portable implementation
- Easy to get started: incremental parallelization
 - Start from most critical or time consuming part of the code

A Multi-Threaded “Hello World” Program

- Write a multithreaded program where each thread prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int main()
{
#pragma omp parallel
{
    printf(" hello ");
    printf(" world \n");
}
}
```

OpenMP include file

Parallel region with
default number of threads

End of the Parallel region

Sample Output:

hello hello world

world

hello hello world

world

The statements are
interleaved based on how the
operating schedules the
threads

A Simple OpenMP Program

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main () {
    int tid, nthreads;
#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread %d\n", tid);
#pragma omp barrier
        if ( tid == 0 ) {
            nthreads = omp_get_num_threads();
            printf("Total threads= %d\n",nthreads);
        }
    }
}
```

Sample Compile and Run:

```
% gcc -fopenmp test.c
% export OMP_NUM_THREADS=4
% ./a.out
```

Program main

```
use omp_lib      (or: include "omp_lib.h")
integer :: id, nthreads
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) "Hello World from thread", id
!$OMP BARRIER
    if ( id == 0 ) then
        nthreads = omp_get_num_threads()
        write (*,*) "Total threads=",nthreads
    end if
!$OMP END PARALLEL
End program
```

The statements are interleaved based on how the operating schedules the threads

Sample Output: (no specific order)

Hello World from thread	0
Hello World from thread	2
Hello World from thread	3
Hello World from thread	1
Total threads=	4

Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- **Creating Threads**
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

Thread Creation: the parallel Construct

FORTRAN:

```
!$OMP PARALLEL PRIVATE(id)
    id = omp_get_thread_num()
    write (*,*) "I am thread", id
 !$OMP END PARALLEL
```

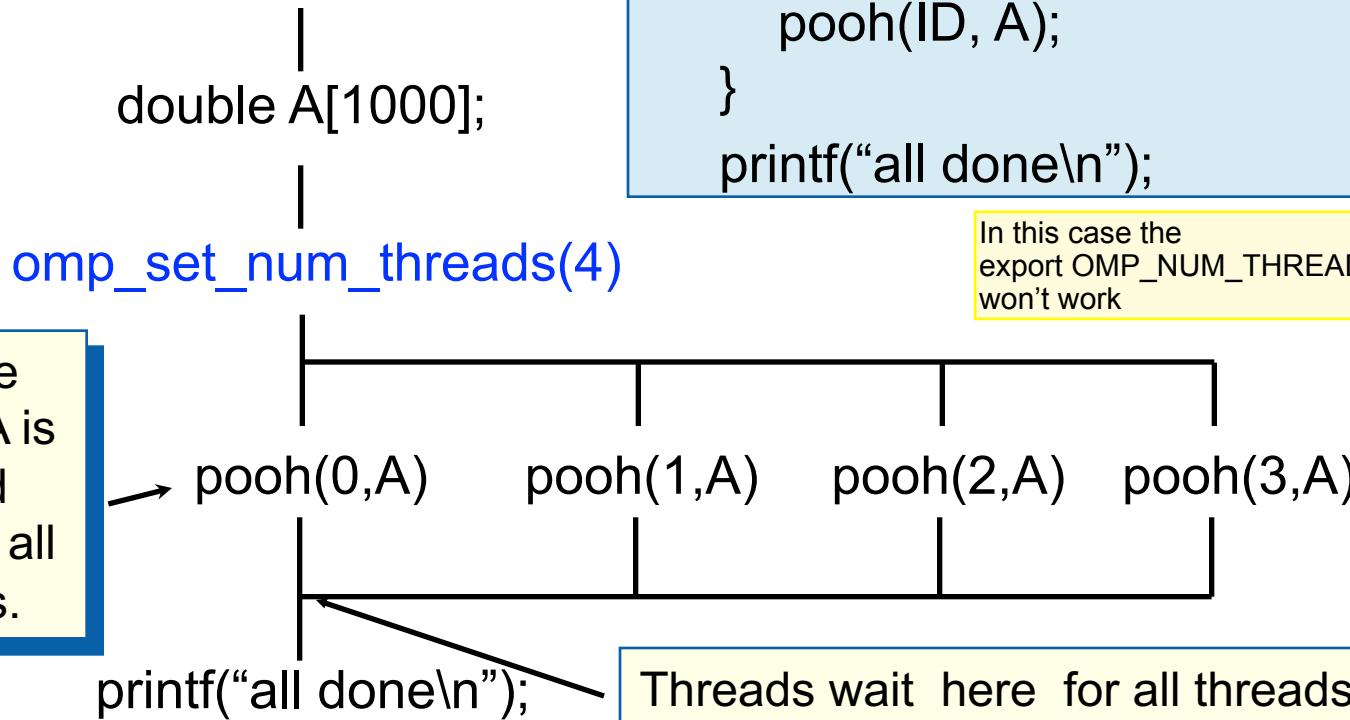
C/C++:

```
#pragma omp parallel private(thid)
{
    thid = omp_get_thread_num();
    printf("I am thread %d\n",
    thid);
}
```

- A team of **threads** for parallel execution can only be created with the **parallel** construct.
- Each thread executes codes within the OpenMP parallel region.
- Threads wait at the end of parallel construct until all threads are finished with the parallel region before any proceed past the end of the parallel region.

Parallel Regions Example

- Each thread executes the same code redundantly.



Single Worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
        {exchange_boundaries();}
    do_many_other_things();
}
```

Various Methods to Set #threads



1) Use num_threads clause

```
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

3) Set runtime environment

```
export OMP_NUM_THREADS=4
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

2) Call omp_set_num_threads API

```
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

4) Do none of the three above.

Code will use an implementation dependent default number of threads defined by the compiler.

- **Precedence: 1) > 2) > 3) > 4)**
- **You may get fewer threads than you requested, check with `omp_get_num_threads()`**

Thread creation: How Many Threads Did You Actually Get?

- You create a team threads in OpenMP with the parallel construct.
- You can request a number of threads with `omp_set_num_threads()`
- But is the number of threads requested the number you actually get?
 - NO! An implementation can silently decide to give you a team with fewer threads.
 - Once a team of threads is established ... the system will not reduce the size of the team.

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID      = omp_get_thread_num();
    int nthrds = omp_get_num_threads();
    pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

- Each thread calls `pooh(ID,A)` for $ID = 0$ to $nthrds - 1$

Performance Tips



- Experiment to find the best number of threads on your system
- Put as much code as possible inside parallel regions
 - Amdahl's law: **If 1/s of the program is sequential, then you cannot ever get a speedup better than s**
 - So if 1% of a program is serial, speedup is limited to 100, no matter how many processors it is computed on
- Have large parallel regions
 - Minimize overheads: starting and stopping threads, executing barriers, moving data into cache
 - Directives can be “orphaned”; procedure calls inside regions are fine
- Run-time routines are your friend
 - Usually very efficient and allow maximum control over thread behavior
- Barriers are expensive
 - With large numbers of threads, they can be slow
 - Depends in part on HW and on implementation quality
 - Some threads might have to wait a long time if load not balanced

Orphaned Directives

Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit.

Directives such as CRITICAL, BARRIER, SECTIONS, SINGLE, MASTER, and DO (FOR in C) can occur by themselves in a program unit, dynamically "binding" to the enclosing parallel region at run time.

Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring. Orphaning can also improve performance by enabling a single parallel region to bind with multiple DO directives located within called subroutines.

```
...
!$OMP PARALLEL
CALL PHASE1
CALL PHASE2
!$OMP END PARALLEL
...

SUBROUTINE PHASE1
!$OMP DO PRIVATE(i) SHARED(n)
DO i = 1, n
    CALL SOME_WORK(i)
END DO
!$OMP END DO
END

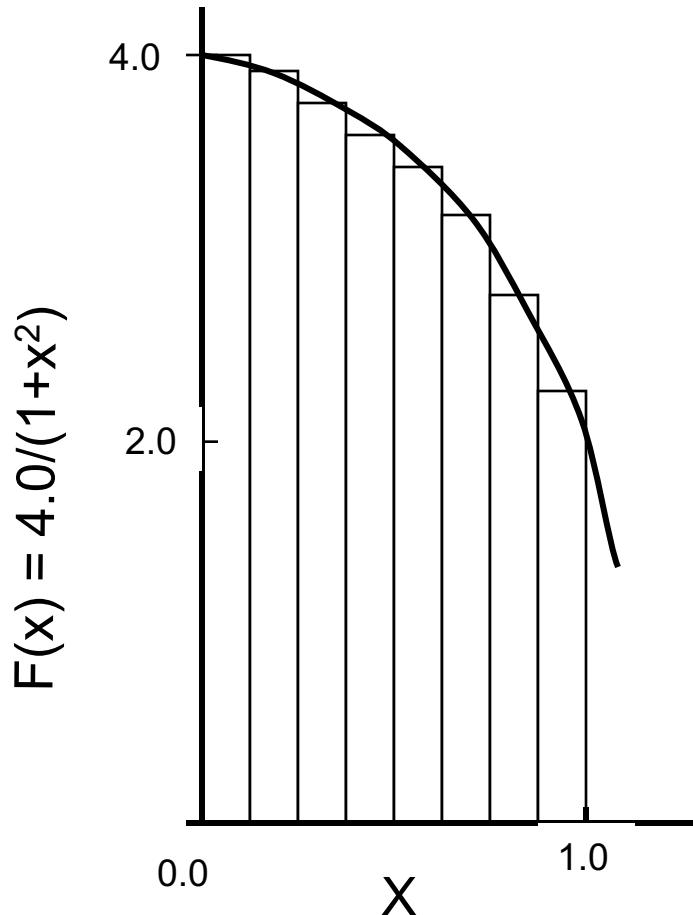
SUBROUTINE PHASE2
!$OMP DO PRIVATE(j) SHARED(n)
DO j = 1, n
    CALL MORE_WORK(j)
END DO
!$OMP END DO
END
```

		Run-time routines
<code>void omp_set_num_threads(int num_threads)</code>	Dynamically set the number of threads to use for this region.	
<code>int omp_get_num_threads(void)</code>	Determine what the current number of threads is that is allowed to execute a region.	
<code>int omp_get_max_threads(void)</code>	Obtains the maximum number of threads ever allowed with this OpenMP* implementation.	
<code>int omp_get_thread_num(void)</code>	Determines the unique thread number of the thread currently executing this section of code.	
<code>int omp_get_num_procs(void)</code>	Determines the number of processors on the current machine.	
<code>int omp_in_parallel(void)</code>	Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel, otherwise it returns zero.	
<code>void omp_set_dynamic(int dynamic_threads)</code>	Enable or disable dynamic adjustment of the number of threads used to execute a parallel region. If <code>dynamic_threads</code> is non-zero, dynamic threads are enabled. If <code>dynamic_threads</code> is zero, dynamic threads are disabled.	
<code>int omp_get_dynamic(void)</code>	Returns non-zero if dynamic thread adjustment is enabled and returns zero otherwise.	
<code>void omp_set_nested(int nested)</code>	Enable or disable nested parallelism. If parameter is non-zero, enable. Default is disabled.	
<code>int omp_get_nested(void)</code>	Always returns zero in the current version of compiler.	
<code>void omp_init_lock(omp_lock_t *lock)</code>	Initialize a unique lock and set lock to point to it.	
<code>void omp_destroy_lock(omp_lock_t *lock)</code>	Disassociate lock from any locks.	
<code>void omp_set_lock(omp_lock_t *lock)</code>	Force the executing thread to wait until the lock associated with lock is available. The thread is granted ownership of the lock when it becomes available.	
<code>void omp_unset_lock(omp_lock_t *lock)</code>	Release executing thread from ownership of lock associated with lock. lock must be initialized via <code>omp_init_lock()</code> , and behavior undefined if executing thread does not own the lock associated with lock.	
<code>int omp_test_lock(omp_lock_t *lock);</code>	Attempt to set lock associated with lock. If successful, return non-zero. lock must be initialized via <code>omp_init_lock()</code> .	
<code>void omp_init_nest_lock(omp_nest_lock_t *lock)</code>	Initialize a unique nested lock and set lock to point to it.	
<code>void omp_destroy_nest_lock(omp_nest_lock_t *lock)</code>	Disassociate the nested lock lock from any locks.	
<code>void omp_set_nest_lock(omp_nest_lock_t *lock)</code>	Force the executing thread to wait until the lock associated with lock is available. The thread is granted ownership of the lock when it becomes available	
<code>void omp_unset_nest_lock(omp_nest_lock_t *lock)</code>	Release executing thread from ownership of lock associated with lock if count is zero. lock must be initialized via <code>omp_init_nest_lock()</code> . Behavior is undefined if executing thread does not own the lock associated with lock.	
<code>int omp_test_nest_lock(omp_nest_lock_t *lock)</code>	Attempt to set lock associated with lock. If successful, return nesting count, otherwise return zero. lock must be initialized via <code>omp_init_lock()</code> .	

An Interesting Pi Program

Numerical integration

Mathematically, we know that:



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i .

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i;    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Serial PI Program

```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{
    int i;    double x, pi, sum = 0.0, tdata;

    step = 1.0/(double) num_steps;
    double tdata = omp_get_wtime();
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    tdata = omp_get_wtime() - tdata;
    printf(" pi = %f in %f secs\n",pi, tdata);
}
```

The library routine `get_omp_wtime()` is used to find the elapsed “wall time” for blocks of code

Exercise: the Parallel Pi Program

- Create a parallel version of the pi program using a parallel construct:

```
#pragma omp parallel
```

- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines

```
- int omp_get_num_threads();
```

Number of threads in the team

```
- int omp_get_thread_num();
```

Thread ID or rank

```
- double omp_get_wtime();
```

Time in Seconds since a
fixed point in the past

```
- omp_set_num_threads();
```

Request a number of
threads in the team



Hints: the Parallel Pi Program

- Use a parallel construct:

```
#pragma omp parallel
```

- The challenge is to:

- divide loop iterations between threads (use the thread ID and the number of threads).
 - Create an accumulator for each thread to hold partial sums that you can later combine to generate the global sum.

SPMD: Single Program Multiple Data

- Run the same program on P processing elements where P can be arbitrarily large.
- Use P and the rank ... an ID ranging from 0 to (P-1) ... to select between a set of tasks and to manage any shared data structures.

This design pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern ... it is probably the most commonly used pattern in the history of parallel programming.

Wrong Code: Has Data Race

```
...
int main()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

#pragma omp parallel private(x, sum)
{
#pragma omp for
    for (i=0; i<=num_steps; i++)
    {
        x=(i+0.5)*step;
        sum=sum+ 4.0/(1.0+x*x);
    }
    pi = pi + step * sum; ←
}
    printf("pi=%f\n",pi);
    return 0;
}
```

Multiple threads
may update pi at
the same time.
Race condition!

Results*: Use an Array for Local Sum

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define NUM_THREADS 2
void main ()
{
    int i, nthreads;  double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #omp set num threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp get thread num();
        nthrds = omp get num threads();
        if(id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0;i< num_steps; i+=nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

The false sharing not always holds true. Therefore apply this technique only if the issue occurs.

threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

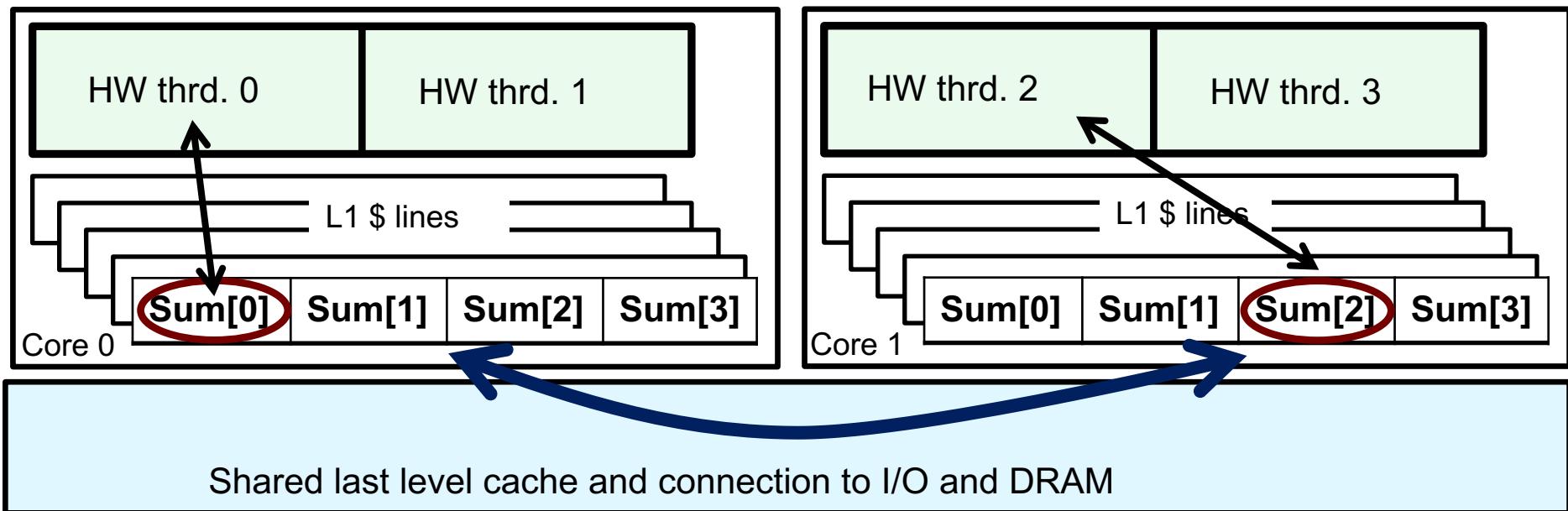
*SPMD: Single Program Multiple Data

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Why Such Poor Scaling? Has False Sharing



- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads
... This is called “**false sharing**”.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines
... Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Eliminate False Sharing via Padding

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define PAD 8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i, id,nthrds;
    double x;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthreads = nthrds;
    for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
        x = (i+0.5)*step;
        sum[id][0] += 4.0/(1.0+x*x);
    }
}
for(i=0, pi=0.0;i<nthreads;i++) pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line



Results*: Pi Program via Padding



- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define PAD 8          //assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id,nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0;i< num_steps; i+=nthreads) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1 st SPMD	1 st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- Creating Threads
- **Synchronization**
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

Synchronization

- High level synchronization included in the common core
 - Critical
 - Barrier
- The full OpenMP specification has MANY more, such as
 - Atomic

Synchronization is used to impose order constraints and to protect access to shared data

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

Threads wait
their turn – only
one at a time
calls consume()

```
float res; critical locks a code segment
```

```
#pragma omp parallel
```

```
{   float B;  int i, id, nthrds;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    for(i=id;i<niters;i+=nthrds){
```

```
        B = big_job(i);
```

```
#pragma omp critical
```

```
    res += consume (B);
```

```
    }
```

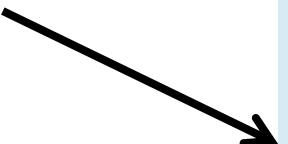
```
}
```

Synchronization: barrier

- Barrier: a point in a program all threads must reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code ... it is an executable statement.
- Barrier makes sure all the shared variables are (explicitly) synchronized.

```
double Arr[8], Brr[8];      int numthrds;  
omp_set_num_threads(8)  
#pragma omp parallel  
{  int id, nthrds;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    if (id==0) numthrds = nthrds;  
    Arr[id] = big_ugly_calc(id, nthrds);  
#pragma omp barrier  
    Brr[id] = really_big_and_ugly(id, nthrds, Arr);  
}
```

Threads
wait until all
threads hit
the barrier.
Then they
can go on.



atomic

- `# pragma omp atomic [read | write | update | capture]`
- Atomic can protect loads
 - `# pragma omp atomic read
v = x;`
- Atomic can protect stores
 - `# pragma omp atomic write
x = expr;`
- Atomic can protect updates to a storage location (this is the default behavior ... i.e. when you don't provide a clause)
 - `# pragma omp atomic update
x++; or ++x; or x--; or -x;
or x binop= expr; or x = x binop expr;
where binop ∈ {+, -, *, /, &, ^, |, <<, >>}`



atomic

- Atomic can protect the assignment of a value (its capture) **and** an associated update operation:
 - `# pragma omp atomic capture`
statement or structured block
- Where the statement is one of the following forms:
 - `v=x++; v=++x; v=x--; v= --x; v=x binop expr;`
- Where the structured block is one of the following forms:
 - `{v=x; x binop = expr;}` `{x binop = expr; v=x;}`
 - `{v=x; x = x binop expr;}` `{x = x binop expr; v=x;}`
 - `{v=x; x++;}` `{v=x; ++x}`
 - `{++x; v=x}` `{x++; v=x;}`
 - `{v=x; x--;}` `{v=x; --x;}`
 - `{--x; v=x;}` `{x--; v=x;}`

Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- Creating Threads
- Synchronization
- **Parallel Loops**
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

The Worksharing-Loop Constructs

- The worksharing-loop construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
```

```
{  
#pragma omp for  
    for (I=0;I<N;I++){  
        NEAT_STUFF(I);  
    }  
}
```

Loop construct name:

- C/C++: for
- Fortran: do

The loop control index I is made “private” to each thread by default.

Threads wait here until all threads are finished with the parallel loop before any proceed past the end of the loop

Worksharing-Loop Constructs

A Motivating Example



Sequential code

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP parallel
region

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

OpenMP parallel
region and a
worksharing for
construct

```
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

Worksharing-Loop Constructs: the schedule Clause



- The schedule clause affects how loop iterations are mapped onto threads
 - `schedule(static [,chunk])`
 - Deal-out blocks of iterations of size “chunk” to each thread.
 - `schedule(dynamic[,chunk])`
 - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

Schedule Clause	When To Use	
STATIC	Pre-determined and predictable by the programmer	Least work at runtime : scheduling done at compile-time
DYNAMIC	Unpredictable, highly variable work per iteration	Most work at runtime : complex scheduling logic used at run-time

Loop schedules

- Default: static scheduling of iterations.

Very efficient. Good if all iterations take the same amount of time.
schedule (static)

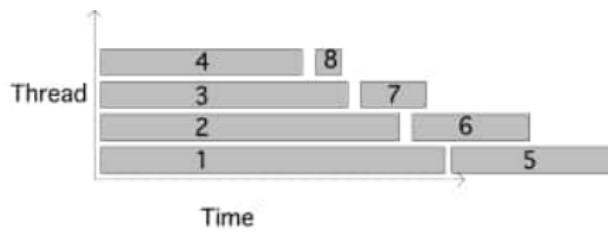
- Other possibility: dynamic.

Runtime overhead; better if iterations do not take the same amount of time.

schedule (dynamic)

Four threads, 8 tasks of decreasing size

dynamic schedule is better:



Chunk size

With N iterations and t threads:

- Static: each thread gets N/t iterations.

explicit chunk size: `schedule(static, 123)`

- Dynamic: each thread gets 1 iteration at a time

explicit chunk size: `schedule(dynamic, 45)`

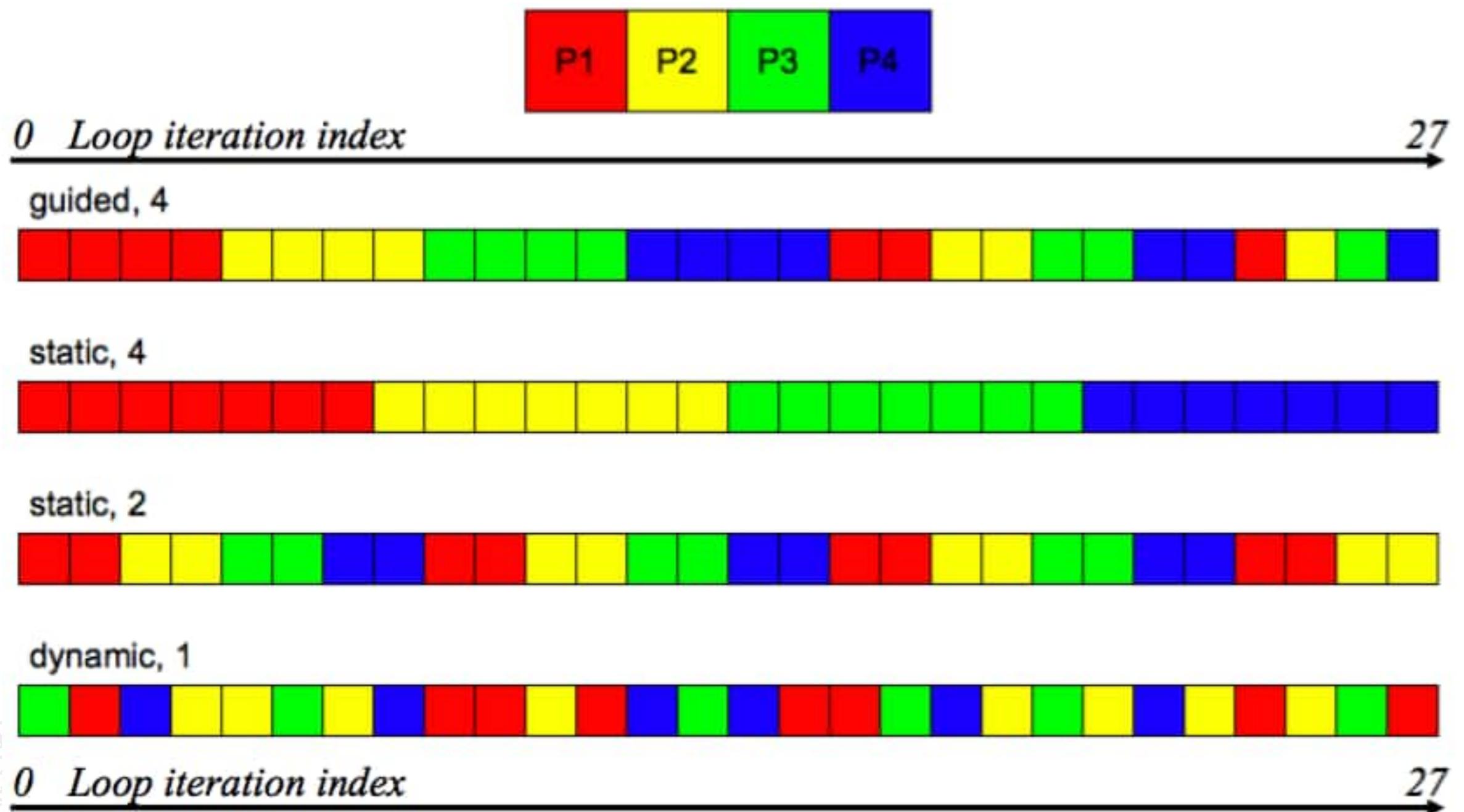
- Help from OpenMP:

guided schedule uses decreasing chunk size (with optional minimum chunk):

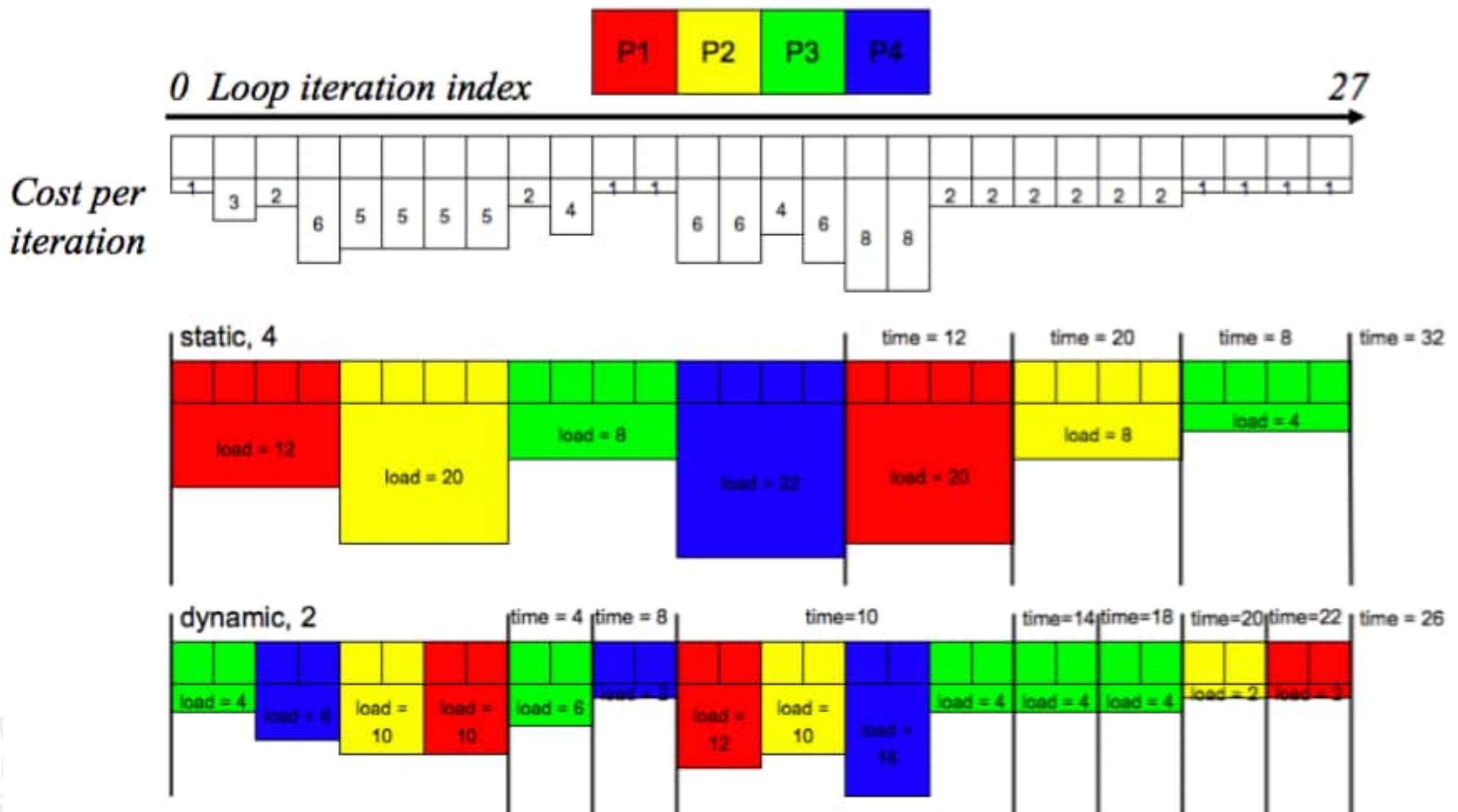
`schedule(guided, 6)`



for loop scheduling



for loop scheduling



Combined Parallel/Worksharing Construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0;i< MAX; i++) {  
        res[i] = huge();  
    }  
}
```

```
double res[MAX]; int i;  
#pragma omp parallel for  
for (i=0;i< MAX; i++) {  
    res[i] = huge();  
}
```

These are equivalent

The Reduction Clause

Serial

```
double ave=0.0, A[max]; int i;  
for (i=0; i< max; i++) {  
    ave += A[i];  
}  
ave = ave/max;
```

Parallel

```
double ave=0.0, A[max]; int i;  
#pragma omp parallel for reduction (+:ave)  
for (i=0; i< max; i++) {  
    ave += A[i];  
}  
ave = ave/max;
```

- Common accumulation pattern, with **loop dependencies in serial code**.
- Syntax: Reduction (operator : list)
- Reduces list of variables into one, using operator
- Reduced variables must be shared variables
- Allowed Operators in Common Core:
 - arithmetic: + - *
 - math: max min
- A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for +, 1 for *)
- Each thread does its local accumulation first, then a global reduction is done at the end

Reduction Operands/Initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

Exercise: Pi with Loops and a Reduction

- Go back to the serial Pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

```
#pragma omp parallel  
#pragma omp for  
#pragma omp parallel for  
#pragma omp for reduction(op:list)  
#pragma omp critical  
int omp_get_num_threads();  
int omp_get_thread_num();  
double omp_get_wtime();
```

Remember: OpenMP makes the loop control index in a loop workshare construct private for you ... you don't need to do this yourself

Pi with a Loop and a Reduction

```
#include <omp.h>
```

```
static long num_steps = 100000;      double step;
```

```
void main ()
```

```
{  int i;  double x, pi, sum = 0.0;  
  step = 1.0/(double) num_steps;  
  #pragma omp parallel
```

```
{
```

```
  double x;
```

```
  #pragma omp for reduction(+sum)
```

```
    for (i=0;i< num_steps; i++) {  
      x = (i+0.5)*step;  
      sum = sum + 4.0/(1.0+x*x);  
    }
```

```
}
```

```
  pi = step * sum;
```

```
}
```

Create a team of threads ...
without a parallel construct, you'll
never have more than one thread

Create a scalar local to each thread to hold
value of x at the center of each interval

Break up loop iterations
and assign them to
threads ... setting up a
reduction into sum.
Note ... the loop index is
local to a thread by default.

Results*: Pi with a Loop and a Reduction



- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Pi with a loop

```
#include <omp.h>
static long num_steps = 100000;
void main ()
{
    int i;        double x, pi, sum =
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;
}
```

threads	1 st SPMD	1 st SPMD padded	SPMD critical	PI Loop and reduction
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Autoparallelization

icc -fopenmp -parallel pi.c

If -qopt_report=5

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char** argv){
    static long num_steps = 1000000000;
    double step;
    double start, stop;

    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;
    start = omp_get_wtime();
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
        pi = step * sum;
    }
    stop = omp_get_wtime();
    printf("Pi is: %.20f\nIn %f seconds\n",pi, stop - start)
}
```

```
LOOP BEGIN at pi.c(13,5)
    remark #17109: LOOP WAS AUTO-PARALLELIZED
    remark #17101: parallel loop shared={} private={} firstprivate={ x i } lastprivate={} firstla...
    remark #15305: vectorization support: vector length 2
    remark #15399: vectorization support: unroll factor set to 4 ... reduction {sum}
    remark #15309: vectorization support: normalized vectorization overhead 0.108
    remark #15300: LOOP WAS VECTORIZED
    remark #15475: --- begin vector cost summary ---
    remark #15476: scalar cost: 46
    remark #15477: vector cost: 25.500
    remark #15478: estimated potential speedup: 1.800
    remark #15486: divides: 1
    remark #15487: type converts: 1
    remark #15488: --- end vector cost summary ---
    remark #25015: Estimate of max trip count of loop=125000000
LOOP END

LOOP BEGIN at pi.c(13,5)
    remark #15305: vectorization support: vector length 2
    remark #15399: vectorization support: unroll factor set to 4
    remark #15309: vectorization support: normalized vectorization overhead 0.108
    remark #15300: LOOP WAS VECTORIZED
    remark #15475: --- begin vector cost summary ---
    remark #15476: scalar cost: 46
    remark #15477: vector cost: 25.500
    remark #15478: estimated potential speedup: 1.800
    remark #15486: divides: 1
    remark #15487: type converts: 1
    remark #15488: --- end vector cost summary ---
    remark #25015: Estimate of max trip count of loop=125000000
LOOP END

LOOP BEGIN at pi.c(13,5)
    <Remainder loop for vectorization>
    remark #15335: remainder loop was not vectorized: vectorization possible but seems inefficient.
    remark #15305: vectorization support: vector length 2
    remark #15309: vectorization support: normalized vectorization overhead 0.627
    remark #25015: Estimate of max trip count of loop=100000000
LOOP END
```

The nowait Clause

- Barriers are really expensive. You need to understand when they are implied and how to skip them when its safe to do so.

```
double A[big], B[big], C[big];  
  
#pragma omp parallel  
{  
    int id=omp_get_thread_num();  
    A[id] = big_calc1(id);  
  
    #pragma omp barrier  
    #pragma omp for  
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);}  
  
    #pragma omp for nowait  
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }  
    A[id] = big_calc4(id);  
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        .....
    }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length $N \times M$ and then parallelize that.
- Useful if N is $O(\text{no. of threads})$ so parallelizing the outer loop makes balancing the load difficult.

Performance Tips



- Is there enough work to amortize overheads?
 - May not be worthwhile for very small loops (if clause can control this)
 - Might be overcome by choosing different loop, rewriting loop nest or collapsing loop nest
- Best choice of schedule might change with system, problem size
 - Experimentation may be needed
- Minimize synchronization
 - Use nowait where possible
- Locality
 - Most large systems are NUMA
 - Be prepared to modify your loop nests
 - Change loop order to get better cache behavior
- If performance is bad, look for false sharing
 - We talk about this in part 2 of the tutorial
 - Occurs frequently, performance degradation can be catastrophic

Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- Creating Threads
- Synchronization
- Parallel Loops
- **Data Environment**
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- Recap, Q&A

OpenMP Data Environment

- Most variables are shared by default
 - Fortran: common blocks, SAVE variables, module variables
 - C/C++: file scope variables, static variables
 - Both: dynamically allocated variables (ALLOCATE, malloc, new)
- Some variables are private by default
 - Certain loop indices
 - Stack variables in subprograms (Fortran) or functions (C) called from parallel regions
 - Automatic (local) variables within a statement block

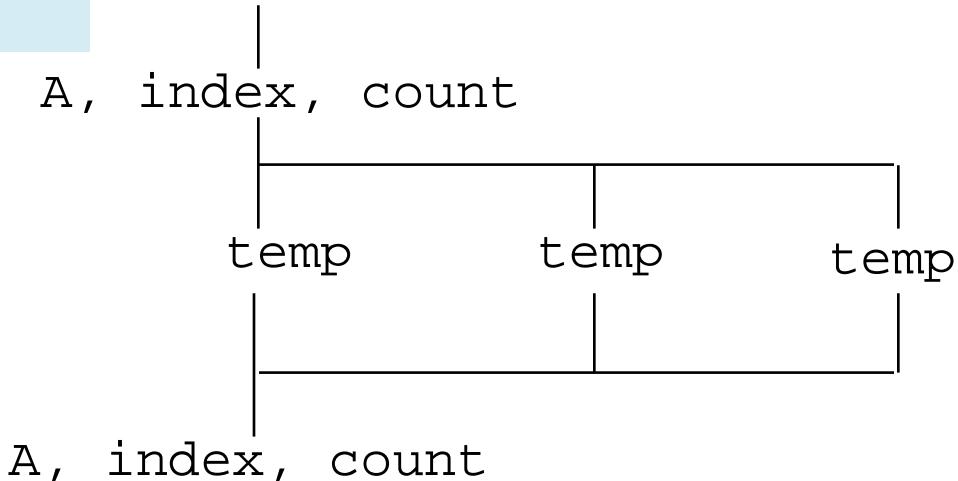
A Data Sharing Example

```
double A[10];  
  
int main() {  
    int index[10];  
  
#pragma omp parallel  
    work(index);  
  
    printf("%d\n", index[0]);  
}
```

A, index and count are shared by all threads.

temp is local to each thread

```
extern double A[10];  
  
void work(int *index) {  
    double temp[10];  
    static int count;  
  
    ...  
}
```



Shared memory problems

Race condition: simultaneous update of shared data:

process 1: $I = I + 2$

process 2: $I = I + 3$

Results can be indeterminate:

scenario 1.	scenario 2.	scenario 3.
$I = 0$		
read $I = 0$ compute $I = 2$ write $I = 2$	read $I = 0$ compute $I = 3$ write $I = 3$	read $I = 0$ compute $I = 2$ write $I = 2$
$I = 3$		
$I = 2$		
$I = 5$		

Data sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses: (note: list is a comma-separated list of variables)
 - `shared(list)`
 - `private(list)`
 - `firstprivate(list)`
- These can be used on parallel and for constructs ... other than shared which can only be used on a parallel construct
- Force the programmer to explicitly define storage attributes
 - `default (none)`

These clauses apply to the OpenMP construct NOT to the entire region.

`default()` can be used on parallel constructs

Data Sharing: private Clause

- `private(var)` creates a new local copy of var for each thread.
 - The value of the private copies is uninitialized
 - The storage of the private copy will be on the each thread's stack memory, and is unassociated with the original variable
 - The value of the original variable is unchanged after the region

```
void wrong() {  
    int tmp = 0;  
#pragma omp parallel for private(tmp)  
    for (int j = 0; j < 1000; ++j)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

When you need to reference the variable `tmp` that exists prior to the construct, we call it the **original variable**.

`tmp` was not initialized

`tmp` is 0 here

Data Sharing: Private Clause When is the Original Variable Valid?

- The original variable's value is unspecified if it is referenced outside of the construct
 - Implementations may reference the original variable or a copy a dangerous programming practice!
 - For example, consider what would happen if the compiler inlined work()?

```
int tmp;  
void danger() {  
    tmp = 0;  
#pragma omp parallel private(tmp)  
    work();  
    printf("%d\n", tmp);  
}
```

tmp has unspecified value

```
extern int tmp;  
void work() {  
    tmp = 5;  
}
```

unspecified which
copy of tmp

The Firstprivate Clause

- Initializes the variables in the list with the value of the shared variable when they **first enter** the construct
- C++ objects are copy-constructed

```
incr = 0;  
#pragma omp parallel for firstprivate(incr)  
for (i = 0; i <= MAX; i++) {  
    if ((i%2)==0) incr++;  
    A[i] = incr;  
}
```

Each thread gets its own copy of
incr with an initial value of 0

Data Sharing: A Data Environment Test

- Consider this example of PRIVATE and FIRSTPRIVATE

variables: A = 1, B = 1, C = 1

```
#pragma omp parallel private(B) firstprivate(C)
```

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Data Sharing: A Data Environment Test

- Consider this example of PRIVATE and FIRSTPRIVATE

variables: A = 1, B = 1, C = 1

```
#pragma omp parallel private(B) firstprivate(C)
```

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- “A” is shared by all threads; equals 1
- “B” and “C” are private to each thread.
 - B’s initial value is undefined
 - C’s initial value equals 1

Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

private variables in/out

- Use `firstprivate` to declare private variables that are initialized with the main thread's value of the variables
- Use `lastprivate` to declare private variables whose values are copied back out to main thread's variables by the thread that executes the last iteration of a parallel for loop, or the thread that executes the last parallel section
- These are special cases of `private`, and are useful to bring values in and out from the parallel section of code.

Data Sharing: default Clause

- **default(None)**: Forces you to define the storage attributes for variables that appear inside the static extent of the construct ... if you fail the compiler will complain. **Good programming practice!**
- You can put the default clause on **parallel** and **parallel + workshare** constructs.

```
#include <omp.h>
int main()
{
    int i, j=5;    double x=1.0, y=42.0;
    #pragma omp parallel for default(None) reduction(*:x)
    for (i=0;i<N;i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n", (float)x);
```

The static extent is the code in the compilation unit that contains the construct.

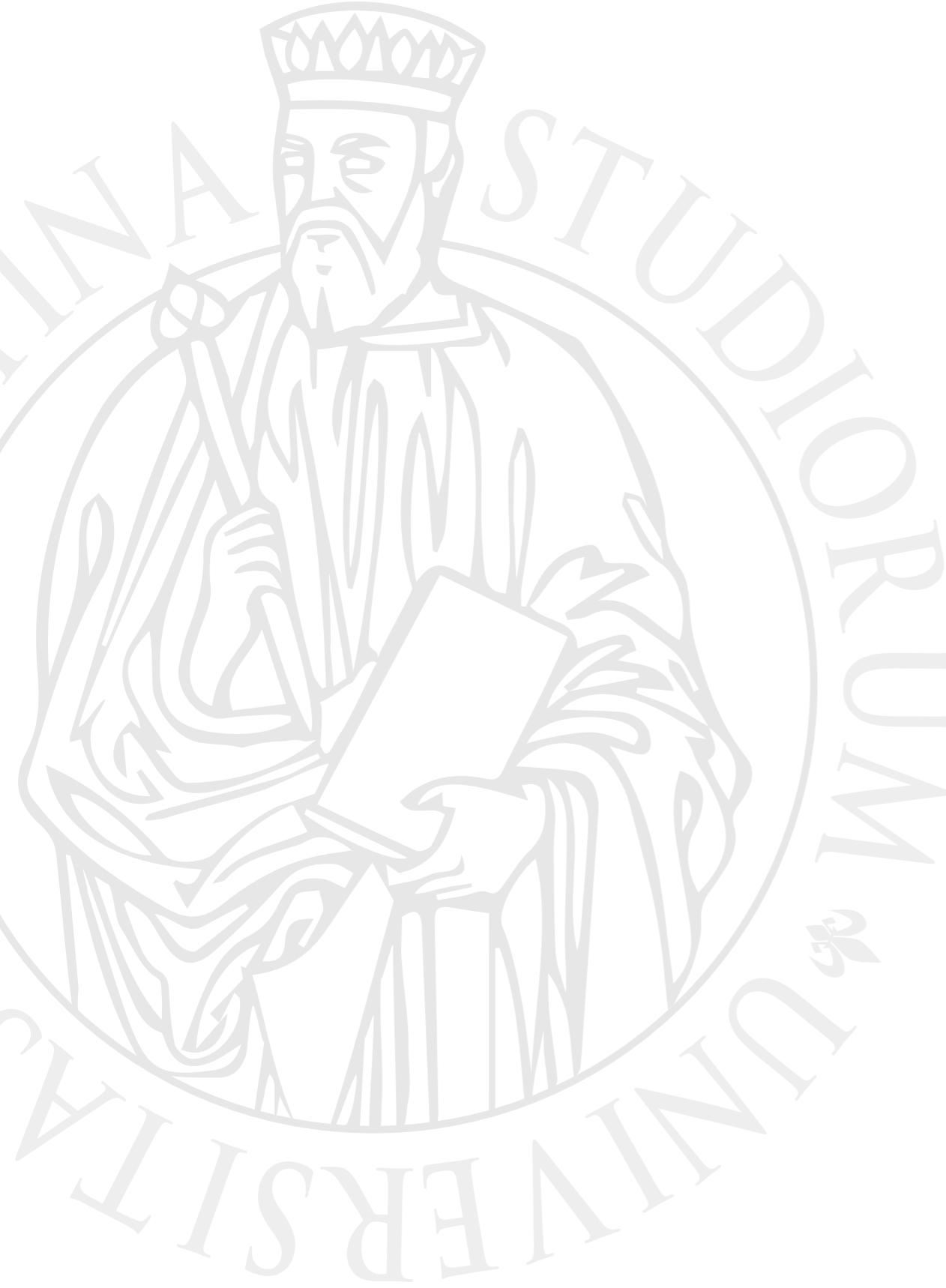
The compiler would complain about j and y, which is important since you don't want j to be shared

The full OpenMP specification has other versions of the default clause, but they are not used very often so we skip them in the common core

Performance and Correctness Tips



- There is one version of shared data
 - Keeping data shared reduces overall memory consumption
- Private data is stored locally, so use of private variables can increase efficiency
 - Avoids false sharing
 - May make it easier to parallelize loops
 - But private data is no longer available after parallel regions ends
- It is an error if multiple threads update the same variable at the same time (a data race)
- It is a good idea to use “default none” while testing code
- Putting code into a subroutine / function can make it easier to write code with many private variables
 - Local / automatic data in a procedure is private by default



OpenMP memory model and synchronization

OpenMP memory model

- In the shared memory model of OpenMP all threads share an address space ... but what they actually see at a given point in time may be complicated: a variable residing in shared memory may be in the cache of several CPUs/cores.
- A memory model is defined in terms of:
 - Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
 - Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

OpenMP memory model

- In the shared memory model of OpenMP all threads share an address space ... but what they actually see at a given point in time may be complicated: a variable residing in shared memory may be in the cache of several CPUs/cores.

At a given point in time, the “private view” seen by a thread may be different from the view in shared memory.

In fact, there are several re-orderings from the original source code:

- Compiler re-orders program order to the code order
- Machine re-orders code order to the memory commit order

Consistency

- Sequential Consistency:
 - In a multi-processor, ops (R, W, S) are sequentially consistent if:
 - They remain in program order for each processor.
 - They are seen to be in the same overall order by each of the other processors.
 - Program order = code order = commit order
 - Relaxed consistency:
 - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP consistency

- OpenMP has a relaxed consistency:
 - S ops must be in sequential order across threads.
 - Can not reorder S ops with R or W ops on the same addresses on the same thread
 - The S operation provided by OpenMP is flush



OpenMP consistency

Relaxed consistency means that memory updates made by one CPU may not be immediately visible to another CPU

- Data can be in registers
- Data can be in cache
(cache coherence protocol is slow or non-existent)

Therefore, the updated value of a shared variable that was set by a thread may not be available to another

The **flush** construct flushes shared variables from local storage (registers, cache) to shared memory

- The S operation provided by OpenMP is **flush**

OpenMP consistency

Relaxed consistency means that memory updates made by one CPU may not be immediately visible to another CPU

- Data can be in registers
- Data can be in cache
(cache coherence protocol is slow or non-existent)

Therefore, the updated value of a shared variable that was set by a thread may not be available to another

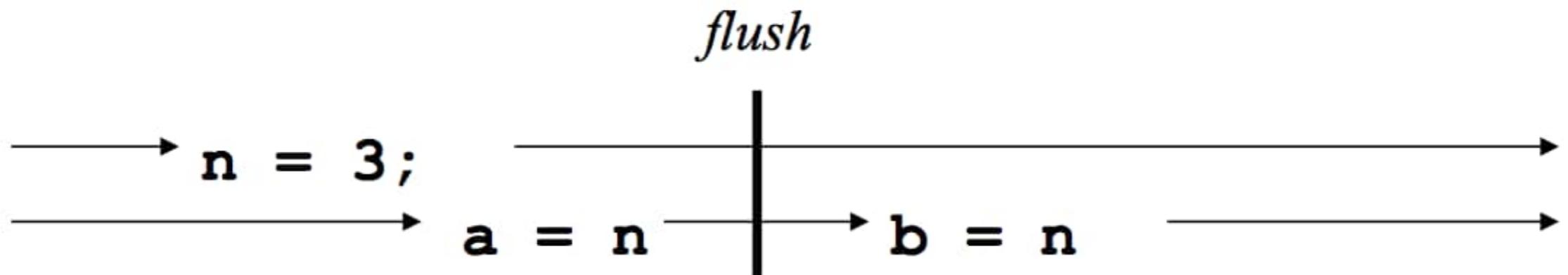
The **flush** construct flushes shared variables from local storage (registers, cache) to shared memory

```
double A;  
A = compute();  
#pragma omp flush(A); // flush to memory to make sure other  
// threads can pick up the right value
```

Implicit flush

- An OpenMP flush is automatically performed at:
 - Entry and exit of parallel and critical and atomic (only variable being atomically updated)
 - unless nowait is specified
- Exit of for
- Exit of sections
- Exit of single
- Barriers
- When setting/unsetting/testing locks after acquisition

- the flush operation **does not actually synchronize** different threads. It just ensures that a thread's values are made consistent with main memory.



- $b = 3$, but there is no guarantee that a will be 3



flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.
- The flush set is:
 - “all thread visible variables” for a `flush` construct without an argument list.
 - a list of variables when the `flush(list)` construct is used.
- The action of `flush` is to guarantee that:
 - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
 - All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
 - Flushes with overlapping flush sets can not be reordered.
- Flush forces data to be updated in memory so other threads see the most recent value.



- A flush construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items.
 - If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers
- A flush construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program is flushed.



Incorrect example:

a = b = 0

thread 1

- - b = 1**
 - flush(b)*
 - flush(a)*
 - if (a == 0) then**
 - critical section*
 - end if**

thread 2

- a = 1**
- flush(a)*
- flush(b)*
- if (b == 0) then**
 - critical section*
- end if**

Correct example:

a = b = 0

thread 1

- - b = 1**
 - flush(a,b)*
 - if (a == 0) then**
 - critical section*
 - end if**

thread 2

- a = 1**
- flush(a,b)*
- if (b == 0) then**
 - critical section*
- end if**

Exercise: Mandelbrot Set Area

- The supplied program (`mandel.c`) computes the area of a Mandelbrot set.
- The program has been parallelized with OpenMP, but we were lazy and didn't do it right.
- Find and fix the errors (hint ... the problem is with the data environment).
- Once you have a working version, try to optimize the program.
 - Try different schedules on the parallel loop.
 - Try different mechanisms to support mutual exclusion ... do the efficiencies change?

The Mandelbrot Area Program



```
#include <omp.h>
#define NPOINTS 1000
#define MXITR 1000
struct d_complex{
    double r;    double i;
};
void testpoint(struct d_complex);
struct d_complex c;
int numoutside = 0;

int main(){
    int i, j;
    double area, error, eps = 1.0e-5;
#pragma omp parallel for default(shared) private(c, j) \
firstprivate(eps)
    for (i=0; i<NPOINTS; i++) {
        for (j=0; j<NPOINTS; j++) {
            c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
            c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
            testpoint(c);
        }
    }
    area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
    numoutside)/(double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
}
```

```
void testpoint(struct d_complex c){
    struct d_complex z;
    int iter;
    double temp;

    z=c;
    for (iter=0; iter<MXITR; iter++){
        temp = (z.r*z.r)-(z.i*z.i)+c.r;
        z.i = z.r*z.i*2+c.i;
        z.r = temp;
        if ((z.r*z.r+z.i*z.i)>4.0) {
            #pragma omp critical
                numoutside++;
            break;
        }
    }
}
```

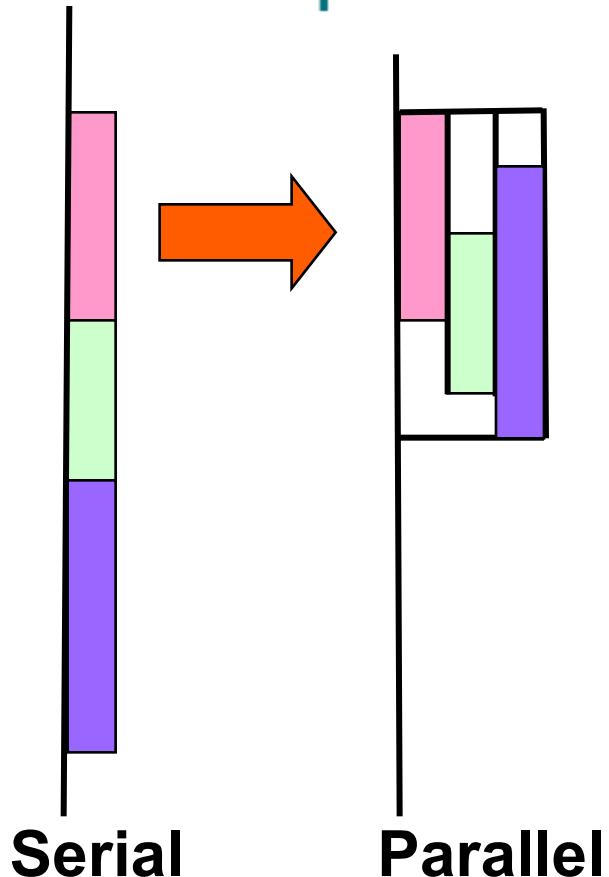
- `eps` was not initialized
- Protect updates of `numoutside`
- Which value of `c` does `testpoint()` see? Global or private?

Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- **Irregular Parallelism and Tasks**
- OpenMP and Performance
- Recap, Q&A

What are OpenMP Tasks?

- Task construct: a structured block of code + a data environment
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- The task is executed immediately, or deferred for later execution.
- Tasks can be nested: i.e. a task may itself generate tasks.



A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

list traversal

- When we first created OpenMP, we focused on common use cases in HPC ... Fortran arrays processed over “regular” loops.
- Recursion and “pointer chasing” were so far removed from our Fortran focus that we didn’t even consider more general structures.
- Hence, even a simple list traversal is exceedingly difficult with the original versions of OpenMP.

```
p=head;  
while (p) {  
    process(p);  
    p = p->next;  
}
```

Linked lists without tasks

- See the file `Linked_omp25.c`

```
while (p != NULL) {
```

```
    p = p->next;
```

```
    count++;
```

```
}
```

```
p = head;
```

```
for(i=0; i<count; i++) {
```

```
    parr[i] = p;
```

```
    p = p->next;
```

```
}
```

```
#pragma omp parallel
```

```
{
```

```
#pragma omp for schedule(static,1)
```

```
for(i=0; i<count; i++)
```

```
    processwork(parr[i]);
```

```
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

Conclusion

- We were able to parallelize the linked list traversal ... but it was ugly and required multiple passes over the data.
- To move beyond its roots in the array based world of scientific computing, we needed to support more general data structures and loops beyond basic for loops.
- To do this, we added tasks in OpenMP 3.0

Task Directive

```
#pragma omp task [clauses]
```

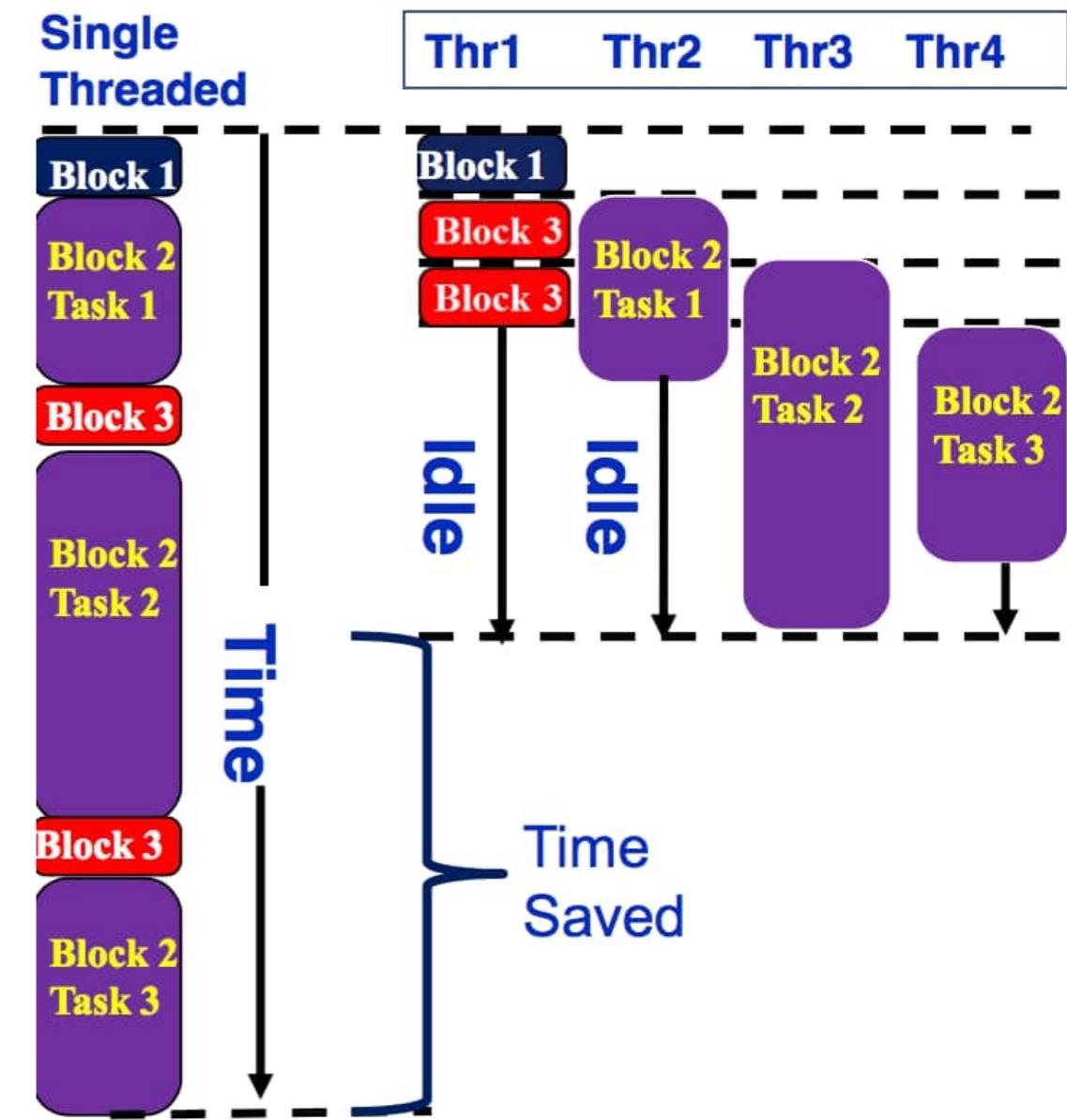
structured-block

```
#pragma omp parallel ← Create some threads
{
    #pragma omp single ← One Thread
    {
        #pragma omp task ← packages tasks
        fred();
        #pragma omp task ← Tasks executed by
        daisy();                                some thread in some
        #pragma omp task ← order
        billy();
    }
}
```

All tasks complete before this barrier is released

task motivations

```
#pragma omp parallel
{
#pragma omp single
{ //block 1
    node * p = head;
    while (p) { // block 2
#pragma omp task
        process(p);
        p = p->next; //block 3
    } // end while
} // end single block
} // end parallel block
```



for and task

- ```
#pragma omp parallel
{
 #pragma omp for private(p)
 for (int i =0; i <numlists ; i++) {
 p = listheads[i] ;
 while(p) {
 #pragma omp task
 process(p)
 p = next(p) ;
 } // end while
 } // end for
 } // end parallel
```

- Example – parallel pointer chasing on multiple lists using tasks (nested parallelism)

# Exercise: Simple Tasks



- Write a program using tasks that will “randomly” generate one of two strings:
  - I think race cars are fun
  - I think car races are fun
- Hint: use tasks to print the indeterminate part of the output (i.e. the “race” or “car” parts).
- This is called a “Race Condition”. It occurs when the result of a program depends on how the OS schedules the threads.
- NOTE: A “data race” is when threads “race to update a shared variable”. They produce race conditions. Programs containing data races are undefined (in OpenMP but also ANSI standards C++'11 and beyond).

```
#pragma omp parallel
#pragma omp task
#pragma omp single
```

# Tasks Example: Racey Cars



```
#include <stdio.h>
#include <omp.h>
int main()
{ printf("I think");
 #pragma omp parallel
 {
 #pragma omp single
 {
 #pragma omp task
 printf(" car");
 #pragma omp task
 printf(" race");
 }
 }
 printf("s");
 printf(" are fun!\n");
}
```

The output can sometimes be:  
I think **car races** are fun!

and sometimes be:  
I think **race cars** are fun!

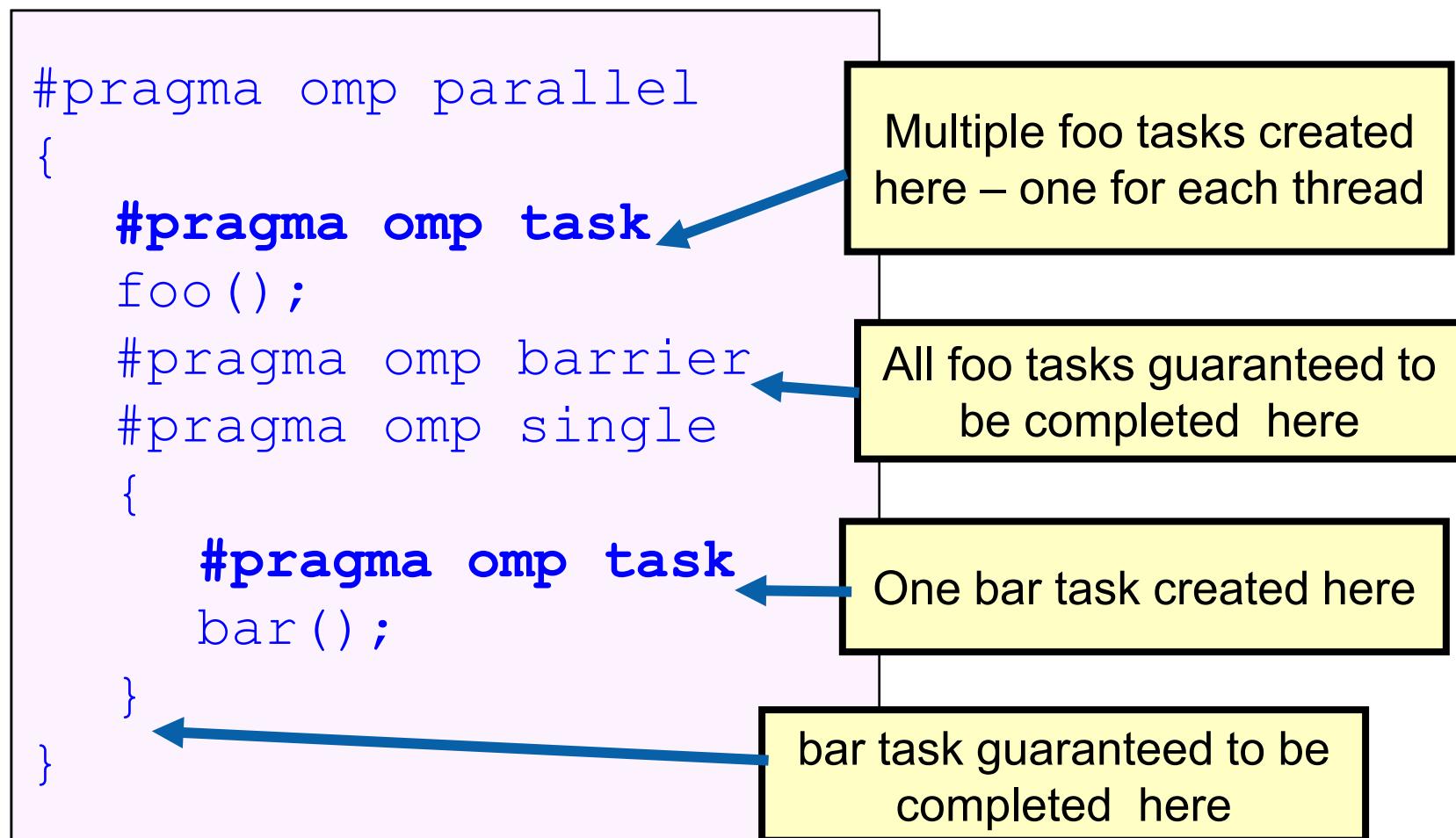
# When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:

#pragma omp barrier

- or task barriers

#pragma omp taskwait



# Example

```
#pragma omp parallel
{
 #pragma omp single
 {
 #pragma omp task
 fred();
 #pragma omp task
 daisy();
 #pragma taskwait
 #pragma omp task
 billy();
 }
}
```

fred() and daisy()  
must complete before  
billy() starts



# Parallel Linked List Traversal

Serial code:

```
p = listhead ;
while (p) {
 process (p) ;
 p=next (p) ;
}
```

```
#pragma omp parallel
{
 #pragma omp single
 {
 p = listhead ;
 while (p) {
 #pragma omp task firstprivate(p)
 {
 process (p) ;
 }
 p=next (p) ;
 }
 }
}
```

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP worksharing-loop directive

Only one thread packages tasks

Makes a copy of p when the task is packaged

# OpenMP Tasks: Data Scoping Defaults

- The behavior you want for tasks is usually **firstprivate**, because the task may not be executed until later (and variables may have gone out of scope)
  - **Variables that are private when the task construct is encountered are firstprivate by default**
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
 ...
#pragma omp task
 {
 int C;
 compute(A, B, C);
 }
}
```

A is shared  
B is firstprivate  
C is private

# Example: Fibonacci numbers

```
int fib (int n)
{
 int x,y;
 if (n < 2) return n;

 x = fib(n-1);
 y = fib (n-2);
 return (x+y);
}

Int main()
{
 int NW = 5000;
 fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient  $O(n^2)$  recursive implementation!

# Data Scoping with tasks: Fibonacci example.

This is an instance of the divide and conquer design pattern

```
int fib (int n)
{
 int x,y;
 if (n < 2) return n;
#pragma omp task
 x = fib(n-1);
#pragma omp task
 y = fib(n-2);
#pragma omp taskwait
 return x+y
}
```

n is private in both tasks

x is a private variable  
y is a private variable

What's wrong here?

A task's private variables are undefined outside the task

# Data Scoping with tasks: Fibonacci example.

```
int fib (int n)
{
 int x,y;
 if (n < 2) return n;
#pragma omp task shared (x)
 x = fib(n-1);
#pragma omp task shared(y)
 y = fib(n-2);
#pragma omp taskwait
 return x+y;
}
```

n is private in both tasks

x & y are shared  
**Good solution**  
we need both values to  
compute the sum

# Parallel Fibonacci

```
int fib (int n)
{
 int x,y;
 if (n < 2) return n;

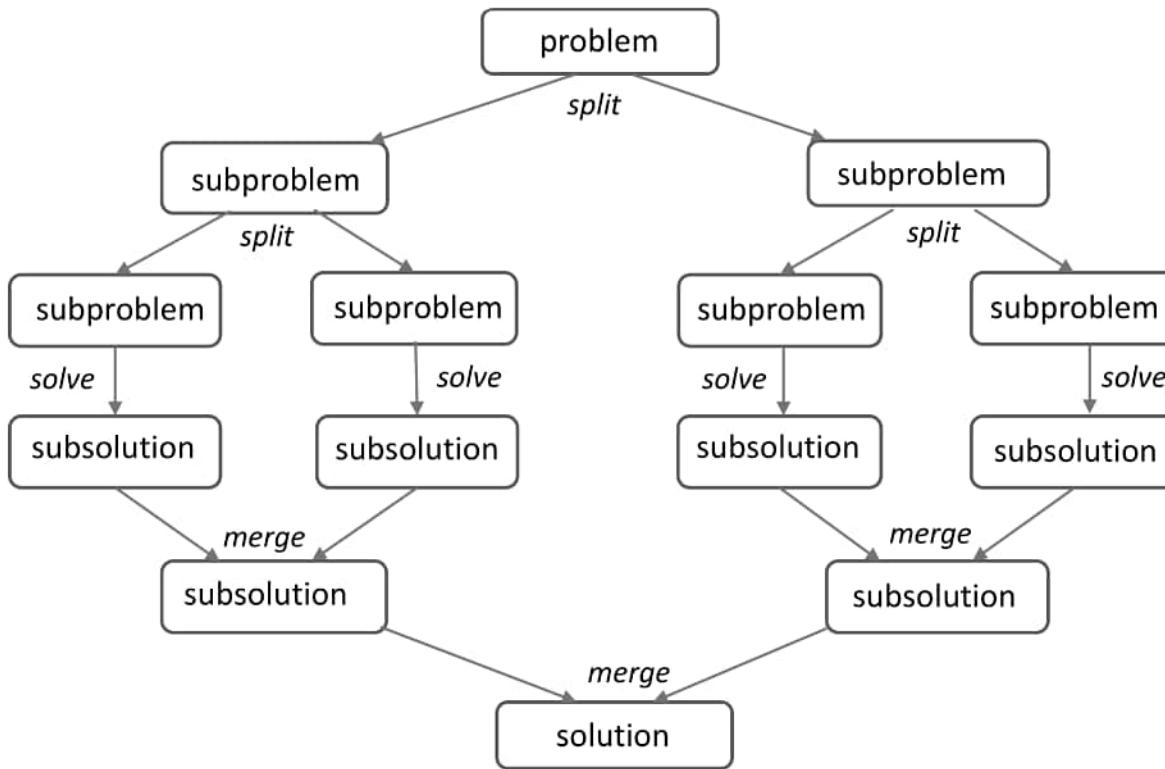
#pragma omp task shared(x)
 x = fib(n-1);
#pragma omp task shared(y)
 y = fib (n-2);
#pragma omp taskwait
 return (x+y);
}

Int main()
{ int NW = 5000;
#pragma omp parallel
{
 #pragma omp single
 fib(NW);
}
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- **x, y** are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

# Divide and Conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



## 3 Options:

- Do work as you split into sub-problems
- Do work only at the leaves
- Do work as you recombine

# Exercise: Pi with Tasks

- Consider the program `Pi_recur.c`. This program uses a recursive algorithm to integrate the function in the `pi` program.
  - Parallelize this program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

# Pi with Tasks



```
#include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK 10000000
double pi_comp(int Nstart,int Nfinish,double step)
{
 int i,iblk;
 double x, sum = 0.0,sum1, sum2;
 if (Nfinish-Nstart < MIN_BLK){
 for (i=Nstart;i< Nfinish; i++){
 x = (i+0.5)*step;
 sum = sum + 4.0/(1.0+x*x);
 }
 }
 else{
 iblk = Nfinish-Nstart;
 #pragma omp task shared(sum1)
 sum1 = pi_comp(Nstart, Nfinish-iblk/2,step);
 #pragma omp task shared(sum2)
 sum2 = pi_comp(Nfinish-iblk/2, Nfinish, step);
 #pragma omp taskwait
 sum = sum1 + sum2;
 }
 return sum;
}
```

```
int main ()
{
 int i;
 double step, pi, sum;
 step = 1.0/(double) num_steps;
 #pragma omp parallel
 {
 #pragma omp single
 sum =
 pi_comp(0,num_steps,step);
 }
 pi = step * sum;
}
```

Recursive divide-and-conquer algorithm

# Pi OpenMP Results



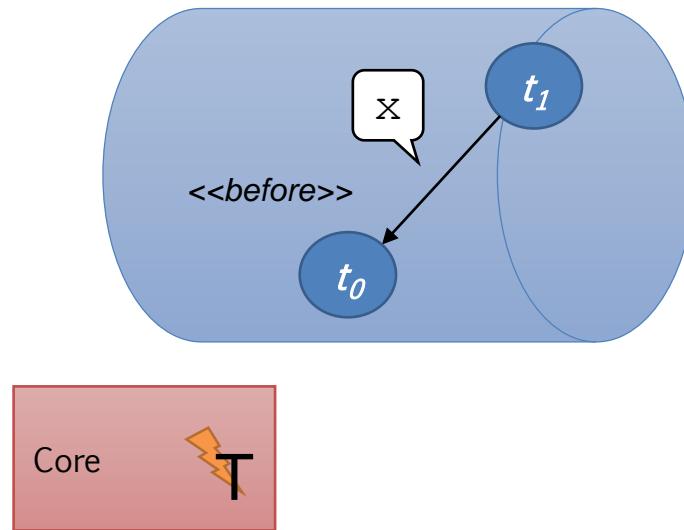
| threads | 1 <sup>st</sup> SPMD | 1 <sup>st</sup> SPMD padded | SPMD critical | PI Loop and reduction | Pi tasks |
|---------|----------------------|-----------------------------|---------------|-----------------------|----------|
| 1       | 1.86                 | 1.86                        | 1.87          | 1.91                  | 1.67     |
| 2       | 1.03                 | 1.01                        | 1.00          | 1.02                  | 1.00     |
| 3       | 1.08                 | 0.69                        | 0.68          | 0.80                  | 0.76     |
| 4       | 0.97                 | 0.53                        | 0.53          | 0.68                  | 0.52     |

- 1<sup>st</sup> SPMD uses array for sum ,has false sharing, hurt speedup.
- 1<sup>st</sup> SPMD uses padded array for sum, avoided false sharing, good speedup.
- SPMD critical uses scalar for sum, no false sharing, good speedup.
- Pi Loop and reduction uses scalar for partial sum, then reduction, relative good speedup, with some loop overhead.
- Pi tasks performs surprisingly well in this test. It is not reproducible on the HPC systems we tested with various compilers. Normally, do no use tasks if an algorithm is already well supported by OpenMP, such as standard do/for loop.

\*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

# The depend clause

- › Set a variable to act as **placeholder** for the dependency



```
#pragma omp parallel
{
 #pragma omp single
 {
 // Dependency is represented as a var
 int x = 0;

 #pragma omp task depend(in:x)
 {
 t0();
 }

 #pragma omp task depend(out:x)
 {
 t1();
 }
 } // bar&TSO
} // parreg end
```

# Using Tasks

- Don't use tasks for things already well supported by OpenMP
- For example standard do/for loops
- The overhead of using tasks is greater
- Don't expect miracles from the runtime
  - Best results usually obtained where the user controls the number and granularity of tasks

# Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- **OpenMP and Performance**
- Recap, Q&A

# The Truth



**Bad OpenMP  
Does Not Scale**

# OpenMP and Performance



*Can Get Good Performance And Scalability*

*If You Do Things Right*

***Easy ≠ Stupid***

# The Basics for All Users



***Do Not Parallelize What Does Not Matter  
(never tune a code without profile)***

***Do Not Share Data Unless You Have To  
(exploit private data as much as you can)***

***One “Parallel For” Is Fine,  
Multiple Back to Back Is Evil  
(merge them where you can)***

***Think BIG  
(maximize the size of the parallel regions)***

# The wrong and right way

```
#pragma omp parallel for
{ code block #1}

pragma omp parallel for
{ code block #n}
```

```
#pragma omp parallel
{
 #pragma omp for
 { code block #1}
 .
 .
 .
 #pragma omp for nowait
 { code block #n}
} // End of par region
```

Overhead of parallel region is repeated <n> times  
No potential for “nowait” clause

Overhead of parallel region only once  
Potential for “nowait” clause

# Parallel in inner loops

```
for (i=0; i<n; i++)
 for (j=0; j<n; j++)
 #pragma omp parallel for
 for (k=0; k<n; k++)
 {
```

Bad:  $n^2$  overheads of parallel

```
#pragma omp parallel
 for (i=0; i<n; i++)
 for (j=0; j<n; j++)
 #pragma omp for
 for (k=0; k<n; k++)
 {
```

Better: 1 overhead of parallel

# The Basics (Yes, Also for You)



*Every Barrier Matters*  
*(but no more than necessary, please)*

*Do Not Lock Yourself Out*  
*(use atomic constructs where possible)*

*Everything Matters*  
*(minor bottlenecks add up too)*

*Why?*

*Amdahl's Law!*

# When Do Things Get Harder



***Memory access “just happens”***

***There are however 2 cases to watch out for***

***NUMA and False Sharing***

***They have nothing to do with OpenMP though and are a characteristic of using a shared memory system***

***With NUMA, the data access time varies***

***The time depends on where the data is***

***There are techniques to avoid this***

***This is covered in the OpenMP books***

# False Sharing



***With false sharing, multiple threads access the same cache line in rapid succession***

***This quickly ruins parallel performance***

***There are techniques to avoid this***

***Throughout this tutorial this is covered***

# Outline

- Introduction
- Why Common Core?
- OpenMP Basics
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP and Performance
- **Recap, Q&A**

# The OpenMP Common Core: Most OpenMP Programs Only Use These 21 items

| OpenMP pragma, function, or clause                                                 | Concepts                                                                                                                                                                              |
|------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| #pragma omp parallel                                                               | Parallel region, teams of threads, structured block, interleaved execution across threads.                                                                                            |
| void omp_set_thread_num()<br>int omp_get_thread_num()<br>int omp_get_num_threads() | Default number of threads and internal control variables.<br>SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID. |
| double omp_get_wtime()                                                             | Speedup and Amdahl's law.<br>False sharing and other performance issues.                                                                                                              |
| setenv OMP_NUM_THREADS N                                                           | Setting the internal control variable for the default number of threads with an environment variable                                                                                  |
| #pragma omp barrier<br>#pragma omp critical                                        | Synchronization and race conditions.<br>Revisit interleaved execution.                                                                                                                |
| #pragma omp for<br>#pragma omp parallel for                                        | Worksharing, parallel loops, loop carried dependencies.                                                                                                                               |
| reduction(op:list)                                                                 | Reductions of values across a team of threads.                                                                                                                                        |
| schedule (static [,chunk])<br>schedule(dynamic [,chunk])                           | Loop schedules, loop overheads, and load balance.                                                                                                                                     |
| shared(list), private(list), firstprivate(list)                                    | Data environment.                                                                                                                                                                     |
| default(None)                                                                      | Force explicit definition of each variable's storage attribute                                                                                                                        |
| nowait                                                                             | Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).                                                   |
| #pragma omp single                                                                 | Workshare with a single thread.                                                                                                                                                       |
| #pragma omp task<br>#pragma omp taskwait                                           | Tasks including the data environment for tasks.                                                                                                                                       |

# Beyond OpenMP Common Core



- **Synchronization mechanisms**
  - locks, flush and several forms of atomic
- **Data environment**
  - lastprivate, threadprivate, default(private|shared)
- **Fine grained task control**
  - dependencies, tied vs. untied tasks, task groups, task loops ...
- **Vectorization constructs**
  - simd, uniform, simdlen, inbranch vs. nobranch, ....
- **Map work to do onto an attached device**
  - target, teams distribute parallel for, target data ...
- ... and much more. OpenMP 5.0 specification is over 600 pages!!!

Don't become overwhelmed. Master the common core and move on to other constructs when you encounter problems that require them.

# Shared and private data

You have already seen some of the basics:

- Data declared outside a parallel region is shared.
- Data declared in the parallel region is private.  
(Fortran does not have this block scope mechanism)

```
int i;
#pragma omp parallel
{ double i; }
```

- You can change all this with clauses:

```
int i;
#pragma omp parallel private(i)
```

# Variables in loops

```
int i; double t;
#pragma omp parallel for
for (i=0; i<N; i++) {
 t = sin(i*pi*h);
 x[i] = t*t;
}
```

- The loop variable is automatically private.
- The temporary `t` is shared, but conceptually private to each iteration:  
needs to be declared private.  
(What happens if you don't?)

# Copying to/from private data

- Private data is uninitialized

```
int i = 3;
#pragma omp parallel private(i)
 printf("%d\n", i); // undefined!
```

- To import a value:

```
int i = 3;
#pragma omp parallel firstprivate(i)
 printf("%d\n", i); // undefined!
```

- lastprivate to preserve value of last iteration.

# Default behaviour

- `default(shared)` or `default(private)`
- useful for debugging: `default(none)`  
because you have to specify everything as shared/private

# Persistent thread data

- Private data disappears after the parallel region.

What if you want data to persist?

- Directive `threadprivate`

```
double seed;
#pragma omp threadprivate(seed)
```

- Standard application: random number generation.
- Tricky: has to be global or static.

# Arrays

- Statically allocated arrays can be made private.
- Dynamically allocated ones can not: the pointer becomes private.

# Need for synchronization

- The loop and sections directives do not specify an ordering, sometimes you want to force an ordering.
- Barriers: global synchronization.
- Critical sections: only one process can execute a statement this prevents race conditions.
- Locks: protect data items from being accessed.

# Barriers

- Every workshare construct has an implicit barrier:

```
#pragma omp parallel
{
 #pragma omp for
 for (.. i ..)
 x[i] = ...
 #pragma omp for
 for (.. i ..)
 y[i] = .. x[i] .. x[i+1] .. x[i-1] ...
}
```

First loop is completely finished before second.

- Explicit barrier:

```
#pragma omp parallel
{
 x = f();
#pragma omp barrier
 x
```

# Critical sections

- Critical section: One update at a time.

```
#pragma omp parallel
{
 double x = f();
#pragma omp critical
 global_update(x);
}
```

- `atomic`: special case for simple operations, possible hardware support

```
#pragma omp atomic
 t += x;
```

# Warning

- Critical sections are not cheap! The operating system takes thousands of cycles to coordinate the threads.
- Use only if minor amount of work.
- Do not use if a reduction suffices.
- Name your critical sections.
- Explore locks if there may not be a data conflict.

# Locks

- Critical sections are coarse:  
they dictate exclusive access to a *statement*
- Suppose you update a big table  
updates to non-conflicting locations should be allowed
- Locks protect a single data item.

# LOCKS

```
omp_lock_t writelock;

omp_init_lock(&writelock);

#pragma omp parallel for
for (i = 0; i < x; i++)
{
 // some stuff
 omp_set_lock(&writelock);
 // one thread at a time stuff
 omp_unset_lock(&writelock);
 // some stuff
}

omp_destroy_lock(&writelock);
```

```
#include <stdio.h>
#include <omp.h>

int main()
{
 int var = 0;
 // init lock
 omp_lock_t lock;

 omp_init_lock(&lock);

 #pragma omp parallel num_threads(1024) shared(lock)
 {
 // set lock
 omp_set_lock(&lock);

 // increment var
 var++;

 // unset lock
 //omp_unset_lock(&lock);
 } // barrier

 printf("var is %d (should be 1024)\n", var);

 // destroy lock
 omp_destroy_lock(&lock);

 return 0;
}
```

```
#pragma omp parallel default(none) shared(buffer) private(tid)
{
 tid = omp_get_thread_num();
 if (tid == 0) consumer(buffer);
 else producer(buffer);
}

int get_value(buffer_type &buffer) {
 while (true) {
 omp_set_lock(&buffer.lock);
 if (buffer.n > 0) {
 buffer.n--;
 int ret = buffer.value[buffer.n];
 omp_unset_lock(&buffer.lock);
 return ret;
 }
 omp_unset_lock(&buffer.lock);
 }
}

void consumer(buffer_type &buffer)
{
 while (true) {
 int value = get_value(buffer);
 solve_problem(value);
 }
}
```

# Producer and consumer with openmp

```
void put_value(buffer_type &buffer, int val)
{
 while (true) {
 omp_set_lock(&buffer.lock);
 if (buffer.n < buffer.size) {
 buffer.value[buffer.n] = val;
 buffer.n++;
 omp_unset_lock(&buffer.lock);
 return;
 }
 omp_unset_lock(&buffer.lock);
 }
}

void producer(buffer_type &buffer)
{
 while (true) {
 // Compute a value
 int value = compute_problem();
 // Insert in buffer
 put_value(buffer, value);
 }
}
```

# Advanced OpenMP Topics

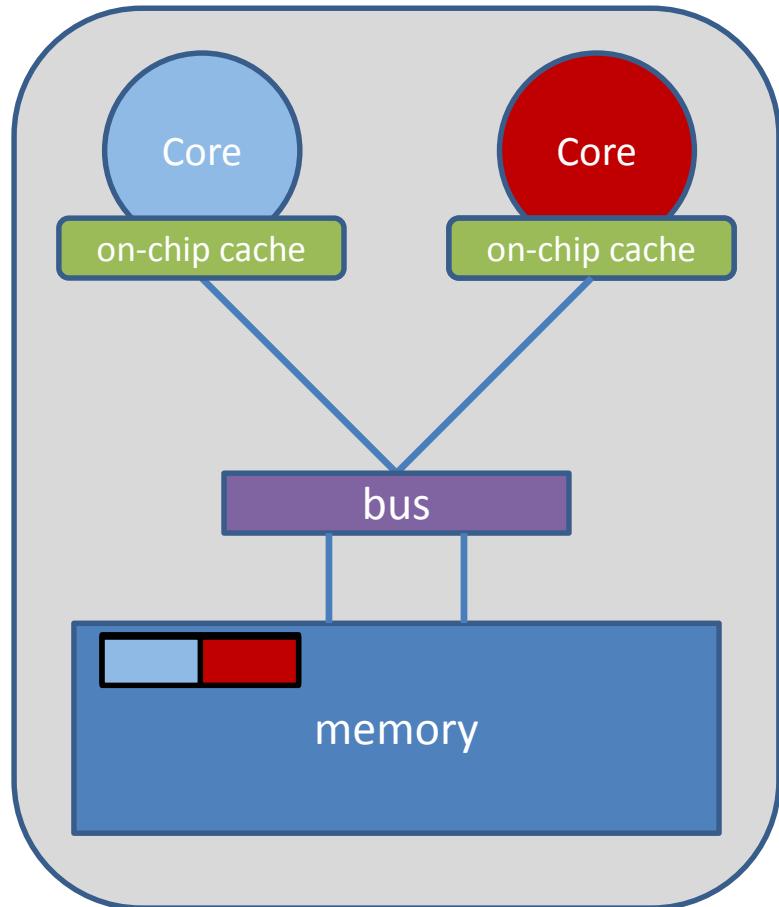
Joachim Protze

IT Center, RWTH Aachen University

[protze@itc.rwth-aachen.de](mailto:protze@itc.rwth-aachen.de)

Many slides provided by: Christian Terboven

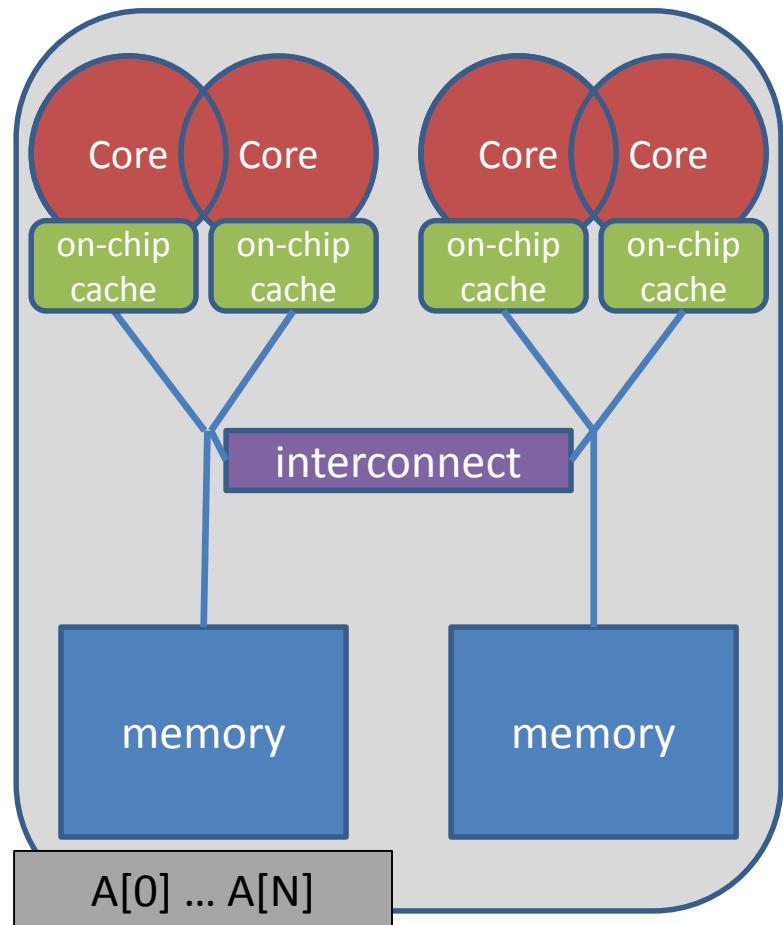
- **False Sharing occurs when**
  - different threads use elements of the same cache-line
  - one of the threads writes to the cache-line
- **As a result the cache line is moved between the threads, although there is no real dependency**
- **Note: False Sharing is a performance problem, not a correctness issue**



## How To Distribute The Data ?

```
double* A;
A = (double*)
 malloc(N * sizeof(double));
```

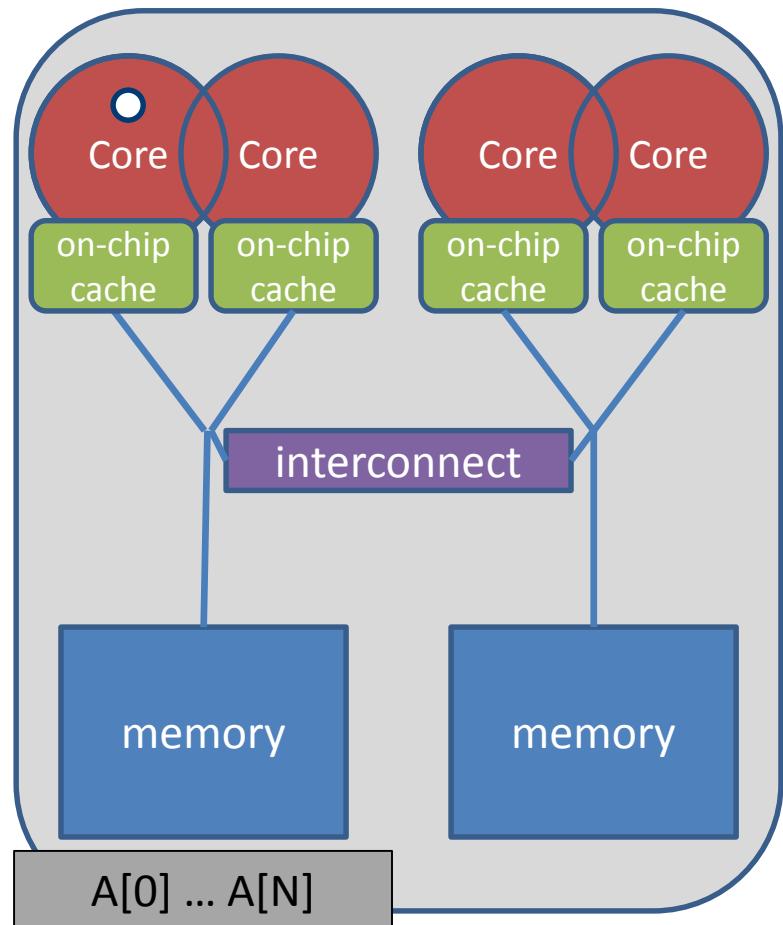
```
for (int i = 0; i < N; i++) {
 A[i] = 0.0;
}
```



- Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread

```
double* A;
A = (double*)
 malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {
 A[i] = 0.0;
}
```

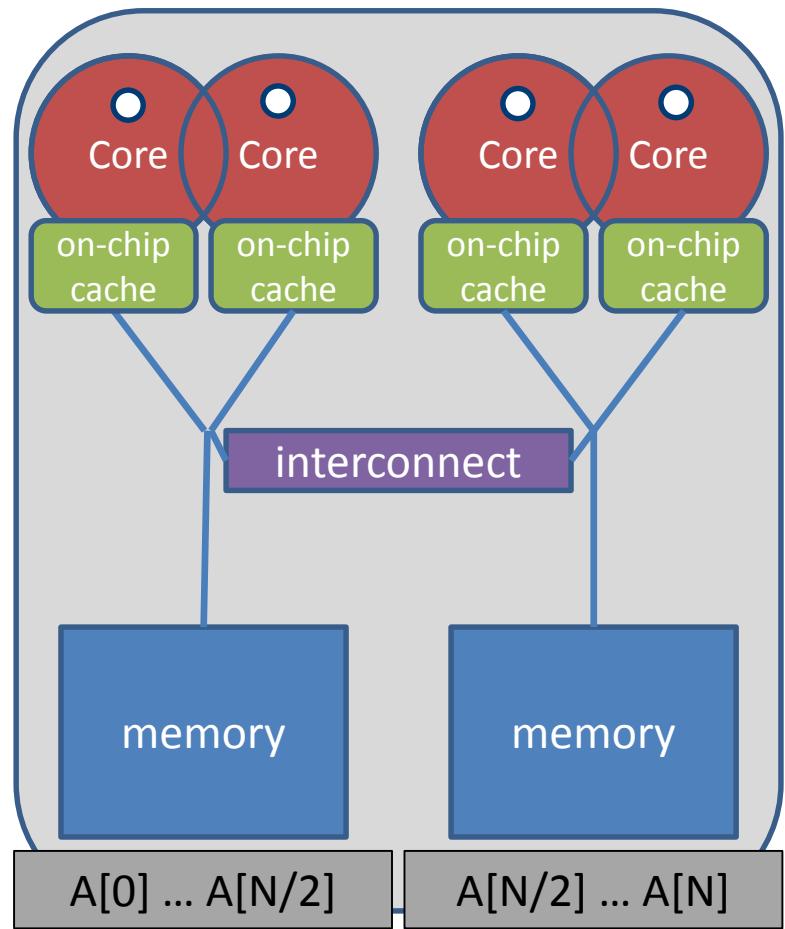


- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

```
double* A;
A = (double*)
 malloc(N * sizeof(double));

omp_set_num_threads(4);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
 A[i] = 0.0;
}
```



- Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.
  - Putting threads far apart, i.e. on different sockets
    - May improve the aggregated memory bandwidth available to your application
    - May improve the combined cache size available to your application
    - May decrease performance of synchronization constructs
  - Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
    - May improve performance of synchronization constructs
    - May decrease the available memory bandwidth and cache size
- If you are unsure, just try a few options and then select the best one.

## ■ Define OpenMP Places

- set of OpenMP threads running on one or more processors
- can be defined by the user, i.e. `OMP_PLACES=cores`

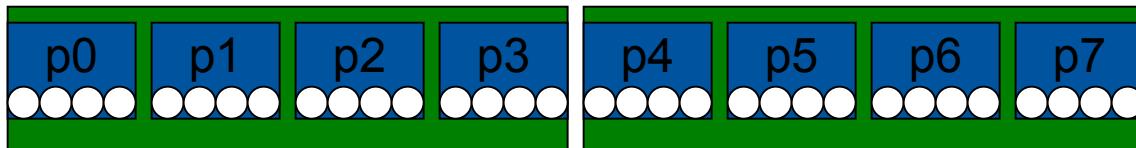
## ■ Define a set of OpenMP Thread Affinity Policies

- SPREAD: spread OpenMP threads evenly among the places
- CLOSE: pack OpenMP threads near master thread
- MASTER: collocate OpenMP thread with master thread

## ■ Goals

- user has a way to specify where to execute OpenMP threads for
- locality between OpenMP threads / less false sharing / memory bandwidth

## ■ Assume the following machine:



→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

## ■ Abstract names for OMP\_PLACES:

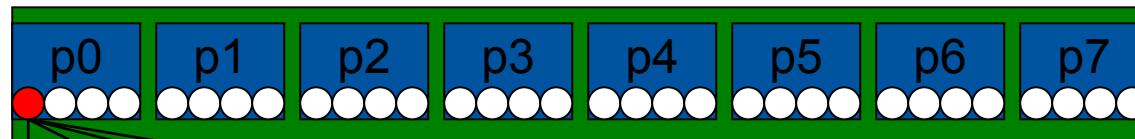
- threads: Each place corresponds to a single hardware thread on the target machine.
- cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
- sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

## Example

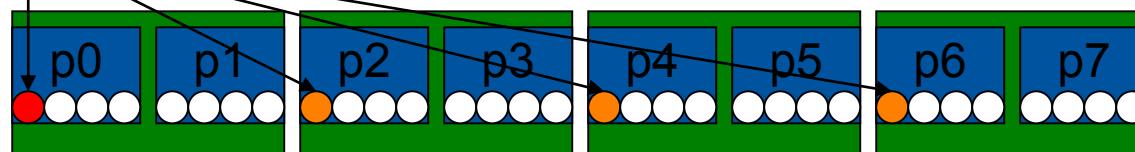
```
OMP_PLACES = cores
OMP_NUM_THREADS = 4
```

```
#pragma omp parallel proc_bind(spread)
```

→ initial



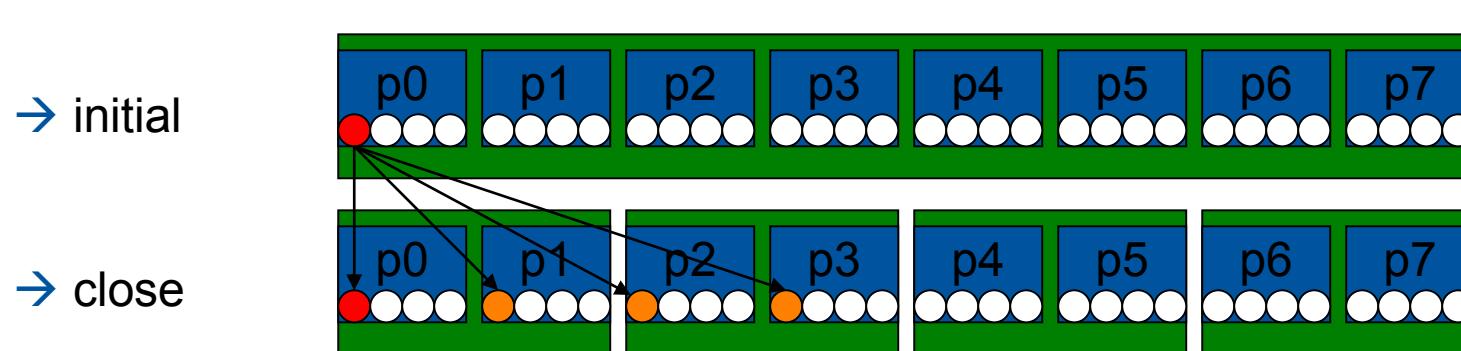
→ spread



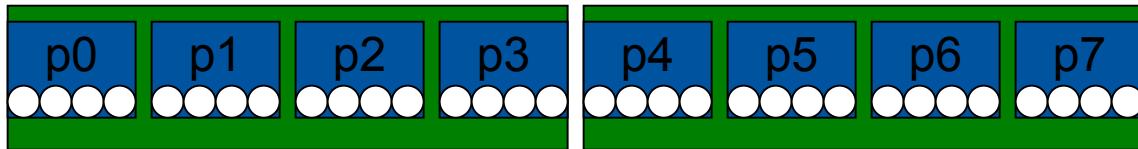
## Example

```
OMP_PLACES = cores
OMP_NUM_THREADS = 4
```

```
#pragma omp parallel proc_bind(close)
```



## ■ Assign different teams to each socket



```
OMP_PLACES = cores
```

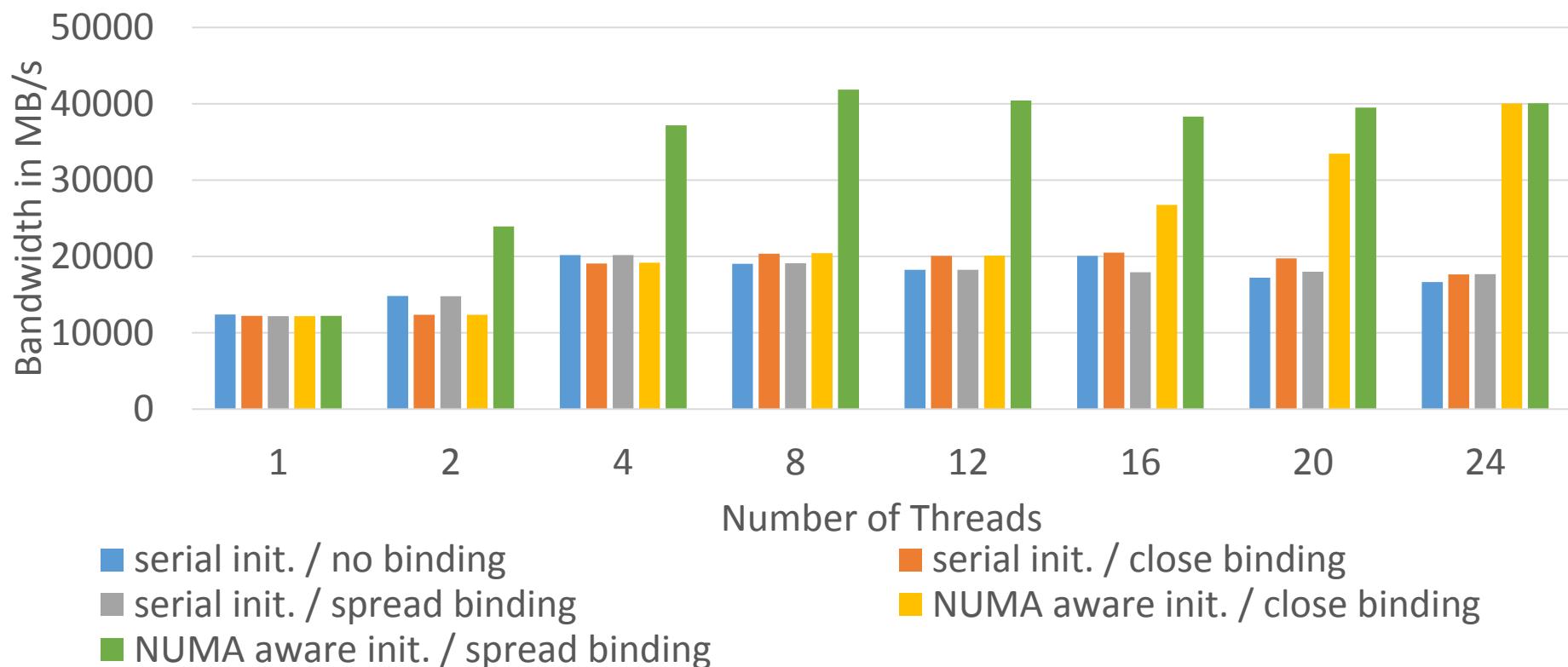
```
#pragma omp teams num_teams(NUM_SOCKETS) proc_bind(spread)
#pragma omp parallel num_threads(4) proc_bind(spread)
```

## ■ OpenMP synchronization is limited to a team

- Reduce the latency of synchronization
- Allow to explicitly partition a problem into independent domains

## ■ Current alternative: If you use hybrid MPI+OpenMP,

- Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:



# If I'm writing new code, how do I choose between Intel Cilk Plus, TBB, OpenMP, and MPI?

It's not an exclusive choice. However let us focus on "how to multi-thread" question. First, you'll want to look at the **development environment**. If the code is written in FORTRAN, use OpenMP. TBB works only for C++, and makes heavy use of templates. If the code is C, or you are not comfortable with templates, your choice is narrowed to Cilk Plus or OpenMP. If you need parallelism with non-Cilk Plus enabled compilers, your choice is narrowed to OpenMP or TBB. If you need parallelism in C++ with compilers that do not support Cilk Plus or OpenMP, then TBB is the way to go, because it does not require compiler support.

If the code makes heavy use of **recursion**, then Cilk Plus or TBB is likely a better fit than OpenMP. Cilk Plus has particularly low cost "spawning", so it's a particularly good fit for highly recursive code. If the code is C++, it's likely that TBB is the best fit. TBB matches especially well with the code that is highly object oriented, and makes heavy use of C++ templates and user defined types. If the code is written in C or FORTRAN, OpenMP may be the better solution because it fits better than TBB into a structured coding style and for simple cases, it introduces less coding overhead.

# If I'm writing new code, how do I choose between Intel Cilk Plus, TBB, OpenMP, and MPI?

How much are you willing to make your code stray from plain serial code? If you use Cilk Plus without array notation, then the code still **looks like serial code** with some markup. In contrast, conversion to TBB will require restructuring certain parts to fit the TBB templates.

Next, look at **what you want to make parallel**. Use OpenMP if the parallelism is primarily for bounded loops over built-in types, or if it is flat do-loop centric parallelism. OpenMP works especially well with large and predictable data parallel problems. It can be very challenging to match OpenMP performance with Cilk Plus or TBB for such problems. Cilk Plus and TBB excels at less structured or consistent parallelism.

Consider using TBB if you need off-the-shelf **patterns** that go beyond loop-based and recursion-based parallelism, since TBB provides generic parallel patterns for parallel while-loops, data-flow pipeline models, parallel sort, and prefix-scan.

# Programming GPUs

A collage of several tutorials

# Introduction to General Purpose GPU Computing

HPC Course @CINECA  
9-11 June 2021

***Sergio Orlandini***

[s.orlandini@cineca.it](mailto:s.orlandini@cineca.it)

***Luca Ferraro***

[l.ferraro@cineca.it](mailto:l.ferraro@cineca.it)



# GPGPU Introduction

Alan Gray

EPCC

The University of Edinburgh

---

# What is a GPU

- **Graphics Processing Unit**

a device equipped with an highly parallel microprocessor (*thousands of cores*) and a private memory with very high bandwidth (about 900GB/s)



- born in '90 as a response to the growing demand for high definition 3D rendering graphic applications (gaming, animations, etc)

# GPU are specialized for parallel intensive computation

GPUs are designed to render complex 3D scenes composed of ***millions of data*** points/vertex at high frame rates (60-120 FPS)

The rendering process requires a set of transformations based on linear algebra operations and (mostly local) filters

- the ***same set of operations*** are applied ***on each data point*** of the scene
- each operation is ***independent*** with respect to data
- all operations are performed ***in parallel*** using a ***huge number of threads*** which process all data independently



# Graphics Logical Pipeline



- The 3D application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons.
- Vertex shader programs map the position of triangle vertices onto the screen, altering their position, color, or orientation.
- Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives.
- Pixel fragment shaders each “shade” one pixel.
- The GPU hardware creates a new independent thread to execute a vertex, geometry, or pixel shader program for every vertex, every primitive, and every pixel fragment.

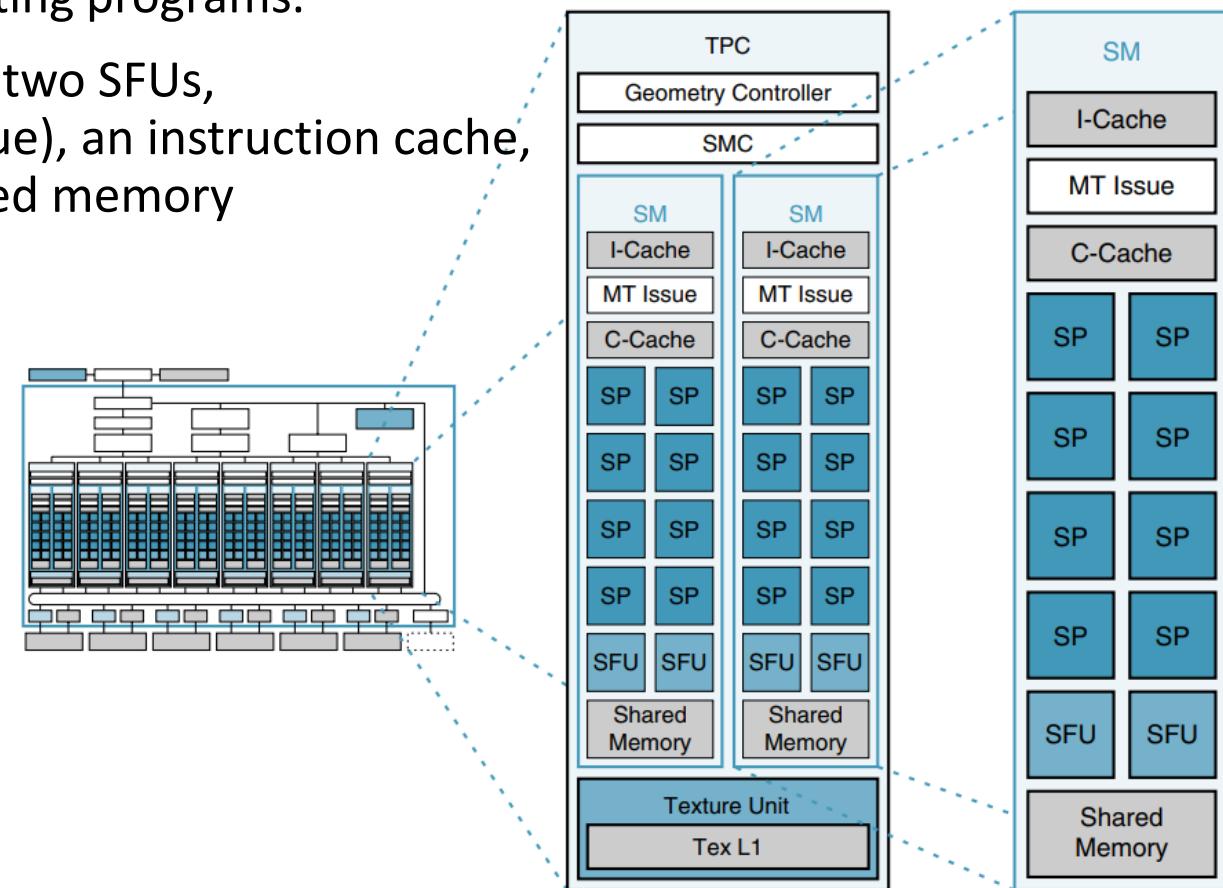
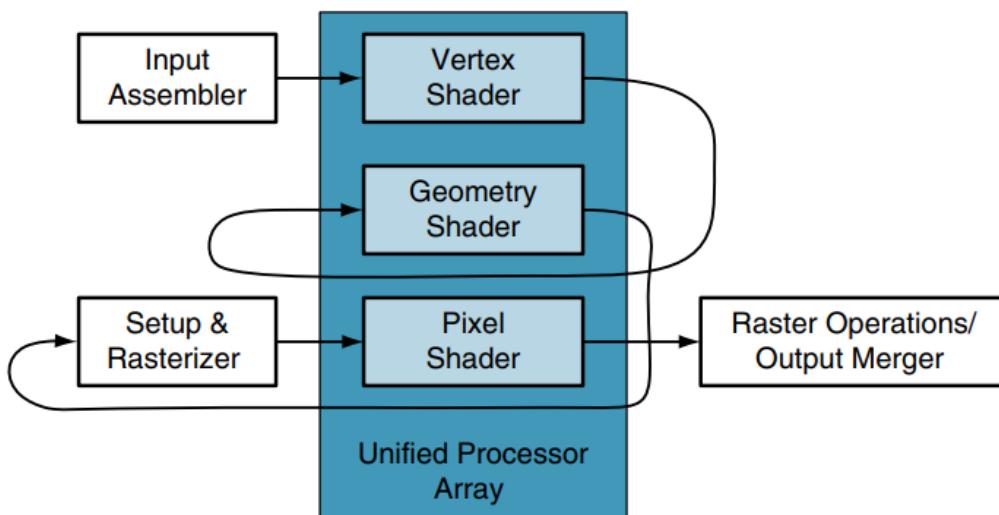
# Unified GPU architectures

Unified GPU architectures are based on a parallel array of many programmable processors.

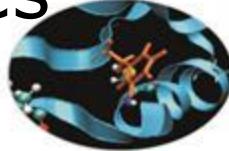
They unify vertex, geometry, and pixel shader processing and parallel computing on the same processors, unlike earlier GPUs which had separate processors dedicated to each processing type.

The Texture Processing Cluster is composed by two Streaming Multiprocessors that execute vertex, geometry, and pixel-fragment shader programs and parallel computing programs.

Each SM consists of eight SP, i.e. thread processor cores, two SFUs, a multithreaded instruction fetch and issue unit (MT issue), an instruction cache, a read-only constant cache, and a 16KB read/write shared memory



# GPU vs CPU: different philosophies



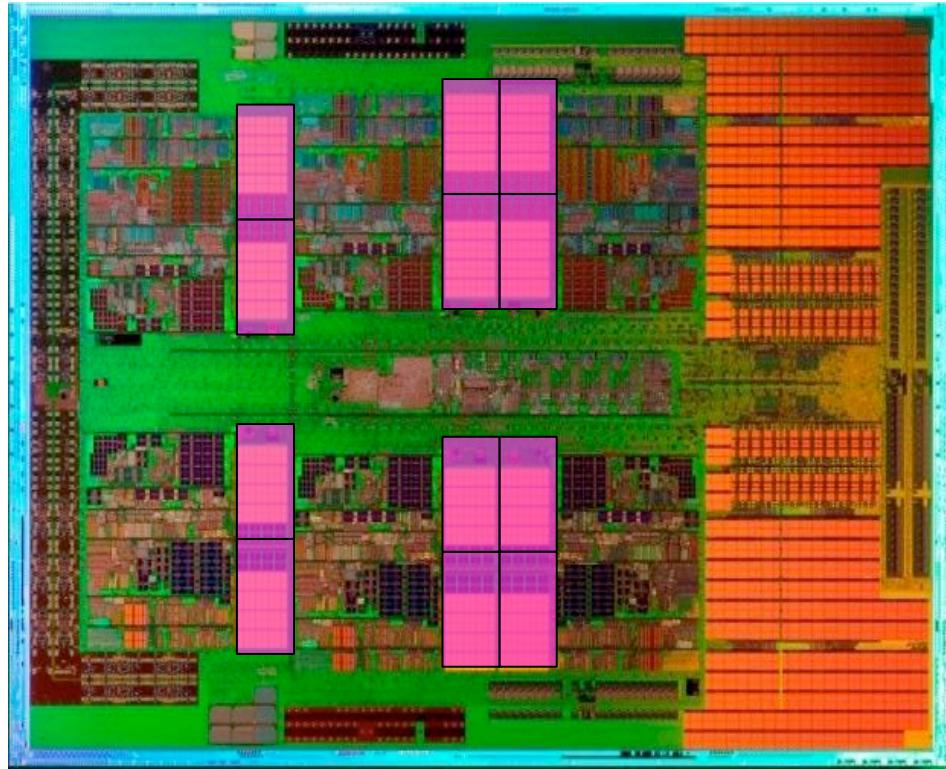
- ⌚ Design of CPUs optimized for sequential code performance:
- ⌚ multi-core
- ⌚ sophisticated control logic unit
- ⌚ large cache memories to reduce access latencies

Design of GPUs optimized for the execution of large number of threads dedicated to floating-points calculations:

- ⌚ many-cores (several hundreds)
- ⌚ minimized the control logic in order to manage lightweight threads and maximize execution throughput
- ⌚ taking advantage of large number of threads to overcome long-latency memory accesses

# AMD 12-core CPU

|epcc|

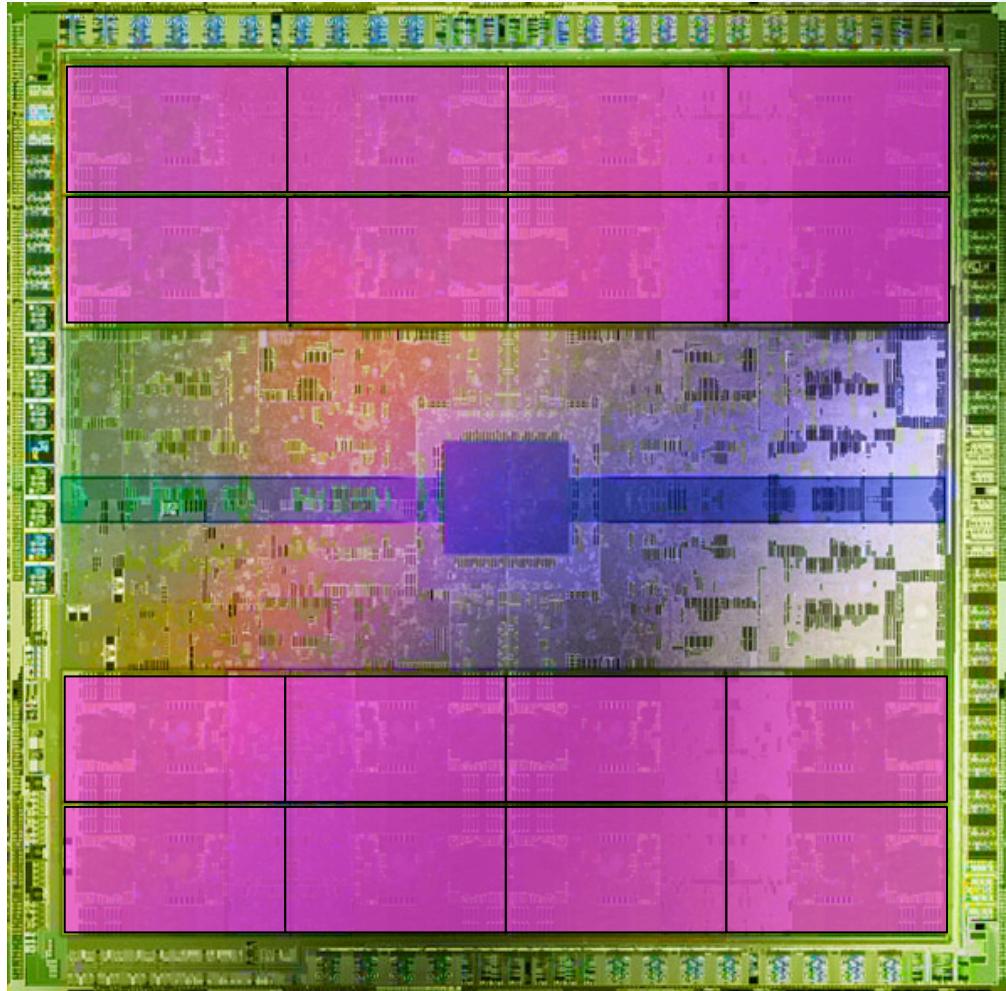


 = compute unit  
(= core)

- Not much space on CPU is dedicated to compute

# NVIDIA Fermi GPU

|epcc|



 = compute unit  
 (= SM)  
 = 32 CUDA cores)

- GPU dedicates much more space to compute
  - At expense of caches, controllers, sophistication etc

# NVIDIA HPC GPU Solutions

| Model       | FP32<br>[TFlops] | cores | RAM<br>[GB] | Bandwidth<br>[GB/s] | Link                    |
|-------------|------------------|-------|-------------|---------------------|-------------------------|
| Kepler K40  | 4.3              | 2280  | 12 GDDR5    | 240                 | PCIe 3.0<br>(15.8 GB/s) |
| Pascal P100 | 10.6             | 3584  | 16 HBM2     | 720                 | PCIe 3.0<br>(15.8 GB/s) |
| Volta V100  | 15.7             | 5120  | 16/32 HBM2  | 900                 | PCIe 3.0<br>(15.8 GB/s) |
| Ampere A100 | 19.5             | 8192  | 40 HBM2     | 1500                | PCIe 4.0<br>(31.6 GB/s) |

- Tesla serie is the NVIDIA top gamma GPU solution for HPC
  - the GeForce series are for gaming
- HBM2: gen2 high-performance RAM interface for 3D-stacked DRAM with Error-correcting (ECC)

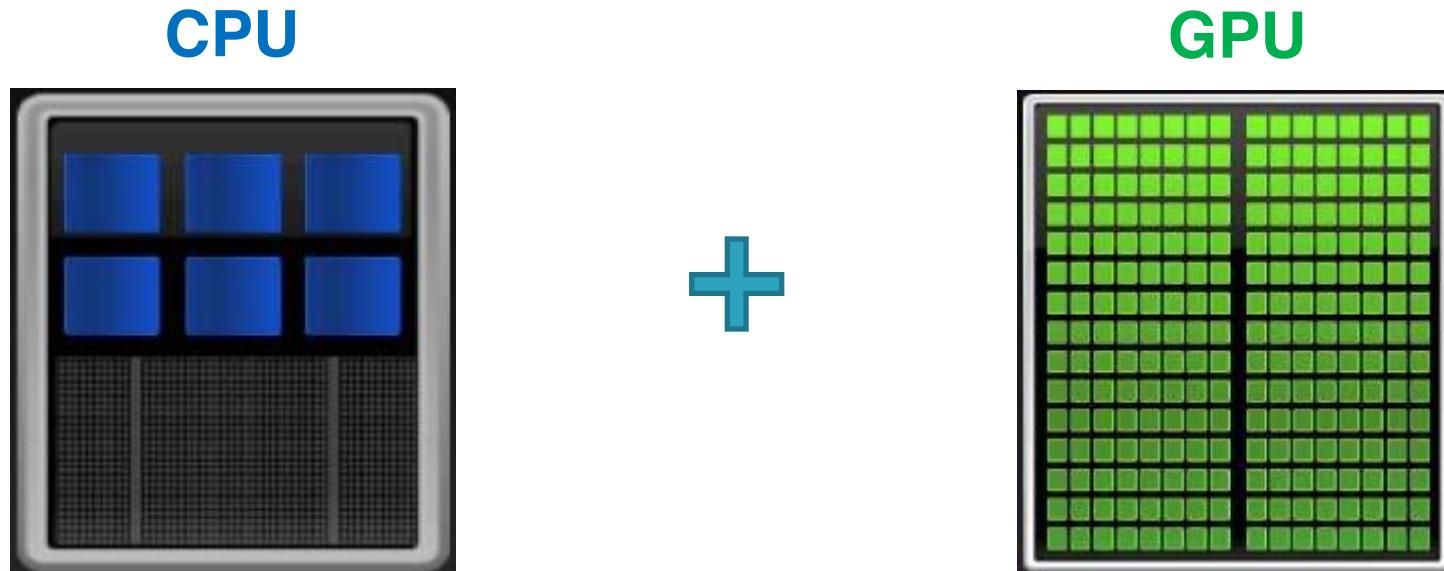
# AMD HPC GPU Solutions

| Model        | FP32<br>[TFlops] | cores | RAM<br>[GB] | Bandwidth<br>[GB/s] | Link                    |
|--------------|------------------|-------|-------------|---------------------|-------------------------|
| Radeon MI8   | 8.2              | 4096  | 4 HBM       | 512                 | PCIe 3.0<br>(15.8 GB/s) |
| Radeon MI25  | 12.3             | 4096  | 16 HBM2     | 484                 | PCIe 3.0<br>(15.8 GB/s) |
| Radeon MI50  | 13.4             | 3840  | 16 HBM2     | 1024                | PCIe 3.0<br>(15.8 GB/s) |
| Radeon MI100 | 32.1             | 7680  | 32 HBM2     | 1200                | PCIe 4.0<br>(31.6 GB/s) |

- VEGA Processor is the AMD top gamma GPU solution for HPC
  - the Radeon RX VEGA/500/400 series are for gaming
- HBM2: gen2 high-performance RAM interface for 3D-stacked DRAM with Error-correcting (ECC)
- Infinity Fabric™ Links per GPU deliver up to 200 GB/s of peer-to-peer bandwidth
- very high TDP factor sustainable on HPC server blades

# GPGPU Programming Model

- General Purpose GPU Programming relates to use GPU computational power to solve problems other than graphics
- CPU and GPU are **separate devices** with **separate memory space addresses**
- GPU is seen as an auxiliary coprocessor equipped with thousands of cores and a high bandwidth memory
- They should work together for best benefit and performances



# GPGPU Programming Model

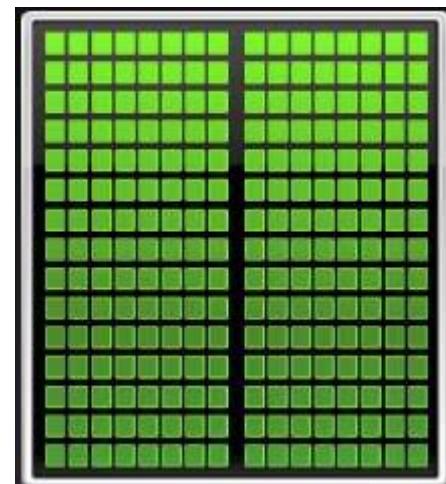
- Optimized for low-latency accesses to caches data sets
- Control logic for out-of-order and speculative execution
- Best for serial or event driven tasks

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- Best for data-parallel tasks

**CPU**

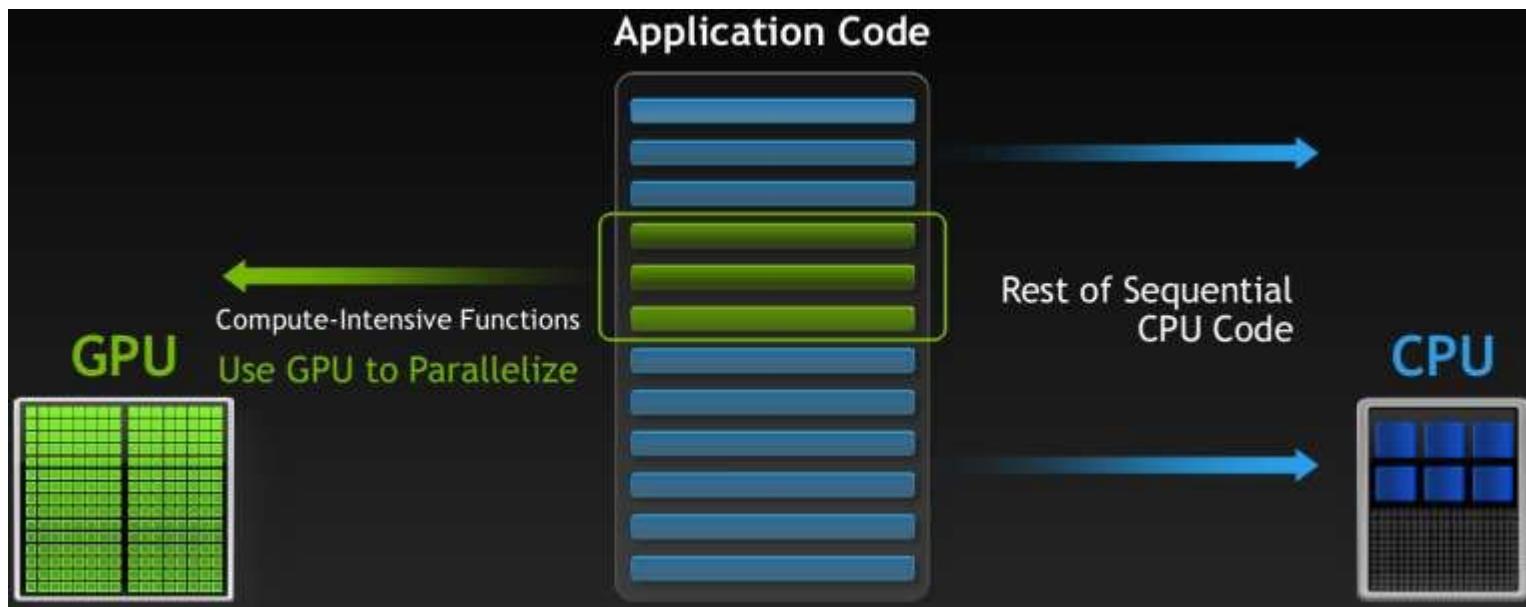


**GPU**

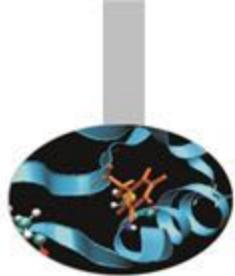


# GPGPU Programming Model

- **serial parts** of a program, or those with low level of parallelism, keep running **on the CPU** (host)
- computational-intensive **data-parallel** regions are executed **on the GPU** (device)
- required data is moved on GPU memory and back to HOST memory



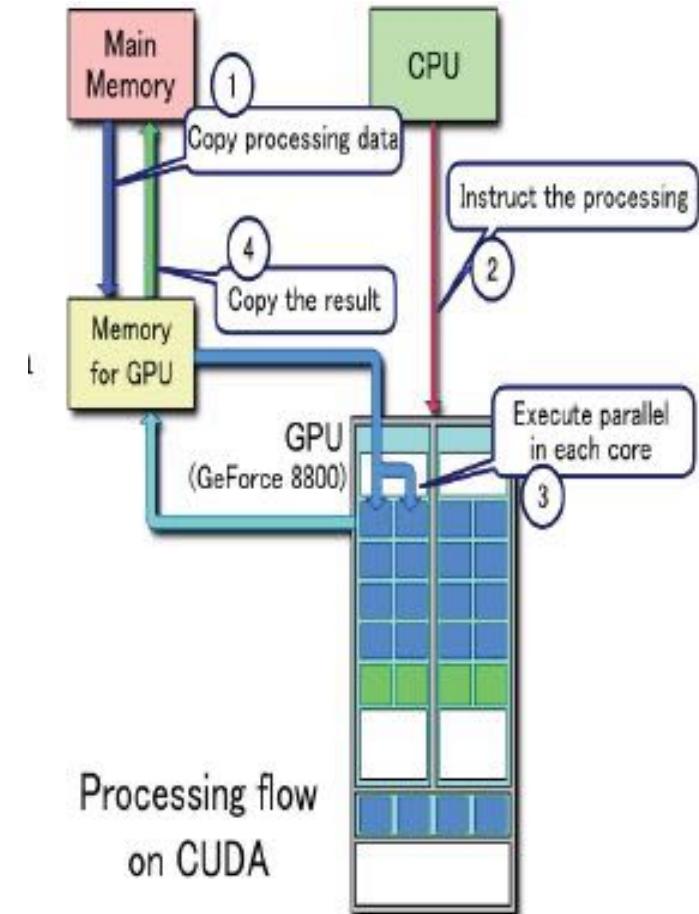
# There cannot be a GPU without a CPU



GPUs are designed as numeric computing engines, therefore they will not perform well on other tasks.

Applications should use both CPUs and GPUs, where the latter is exploited as a coprocessor in order to speed up numerically intensive sections of the code by a massive fine grained parallelism.

CUDA programming model introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.



# DIY GPU Workstation

|epcc|

Do It Yourself



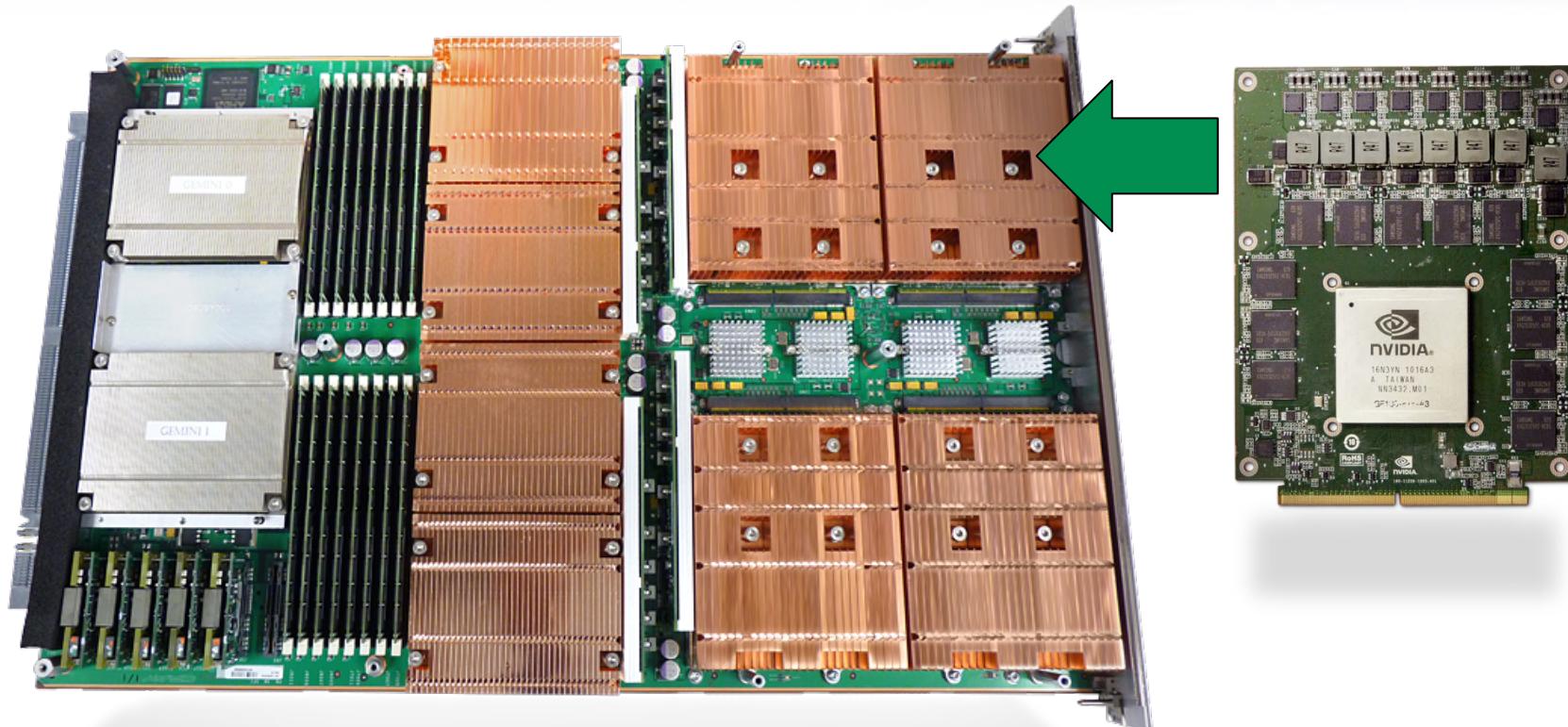
- Just need to slot GPU card into PCI-e
- Need to make sure there is enough space and power in workstation



- Several vendors offer GPU Servers
- Example Configuration:
  - 4 GPUs plus 2 (multi-core) CPUs
- Multiple servers can be connected via interconnect

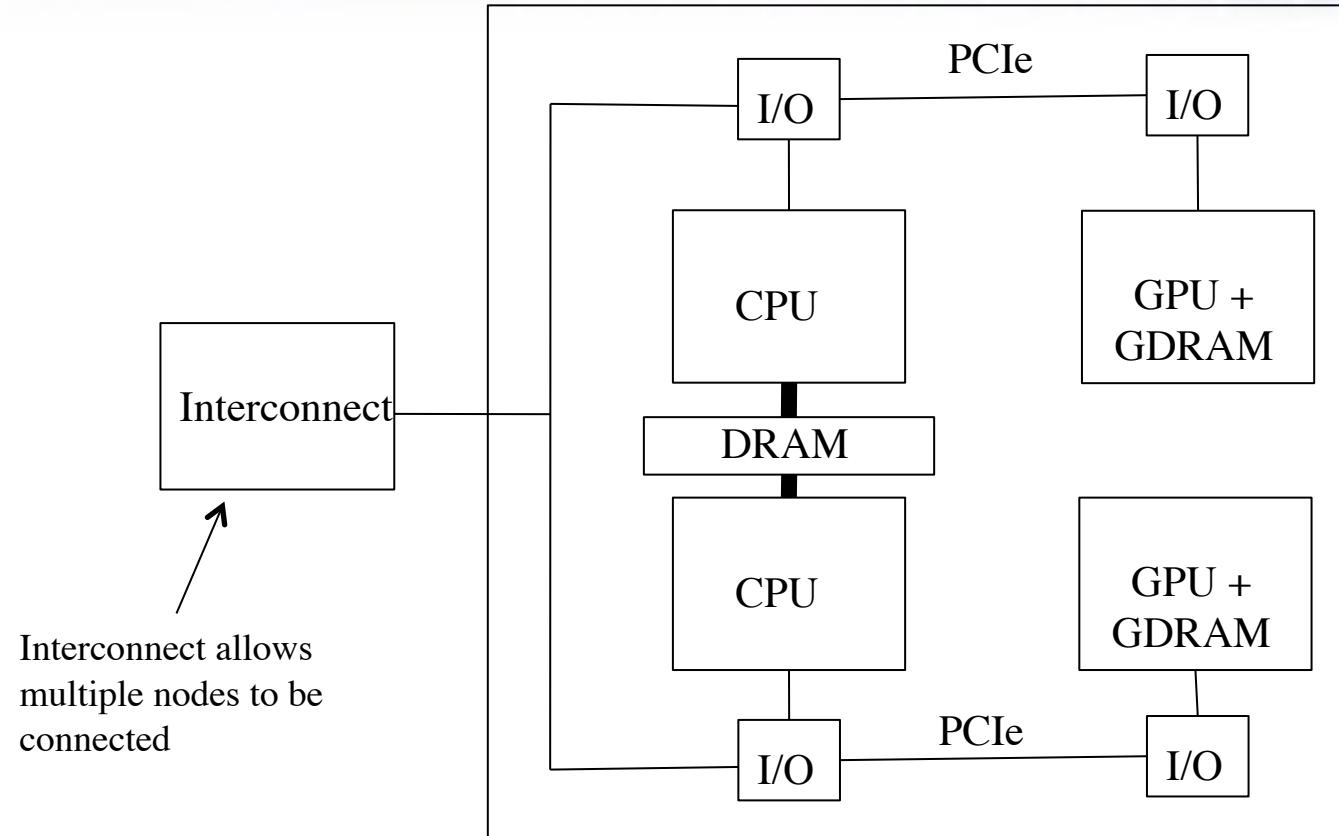
# Cray XK6 Compute Blade

|epcc|



- Compute Blade: 4 Compute Nodes
  - 4 CPUs (middle) + 4 GPUs (right)
    - + 2 interconnect chips (left) (2 compute nodes share a single interconnect chip)

# Scaling to larger systems



- Can have multiple CPUs and GPUs within each “workstation” or “shared memory node”
  - E.g. 2 CPUs +2 GPUs (above)
  - CPUs share memory, but GPUs do not



November 2021

A total of 151 systems on the list are using accelerator/co-processor technology, up from 147 six months ago. 84 of these use NVIDIA Volta chips, 43 use NVIDIA Ampere, and 8 systems with NVIDIA Pascal.

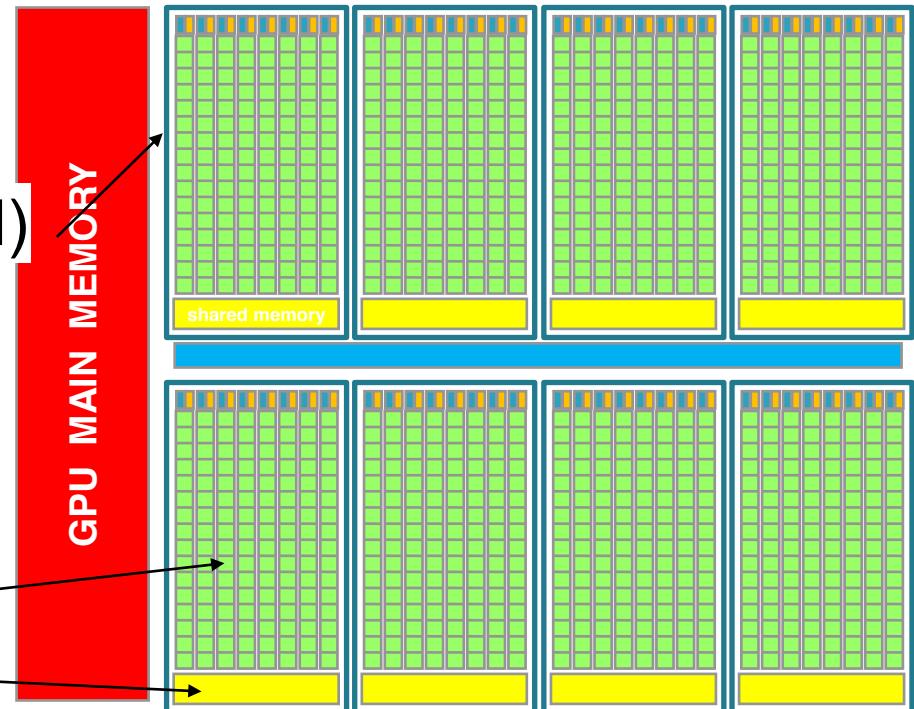
|    |                        | Count | System Share (%) | Rmax (TFlops) | Rpeak (TFlops) | Cores     |
|----|------------------------|-------|------------------|---------------|----------------|-----------|
| 1  | NVIDIA Tesla V100      | 68    | 13.6             | 226,796       | 443,631        | 4,688,680 |
| 2  | NVIDIA A100            | 18    | 3.6              | 237,246       | 344,051        | 2,260,896 |
| 3  | NVIDIA Tesla V100 SXM2 | 11    | 2.2              | 90,370        | 180,163        | 2,031,440 |
| 4  | NVIDIA A100 80GB       | 9     | 1.8              | 121,225       | 160,208        | 1,026,944 |
| 5  | NVIDIA A100 40GB       | 8     | 1.6              | 52,766        | 84,264         | 555,588   |
| 6  | NVIDIA A100 SXM4 40 GB | 8     | 1.6              | 115,838       | 160,747        | 1,229,272 |
| 7  | NVIDIA Tesla P100      | 7     | 1.4              | 46,445        | 68,784         | 944,960   |
| 8  | NVIDIA Volta GV100     | 4     | 0.8              | 269,439       | 362,565        | 4,408,096 |
| 9  | NVIDIA Tesla K40       | 3     | 0.6              | 8,824         | 14,612         | 201,328   |
| 10 | Matrix-2000            | 1     | 0.2              | 61,445        | 100,679        | 4,981,760 |

...

# GPU Architecture Scheme

A typical GPU architecture consists of

- Main Global Memory
  - medium size (8-16 GB)
  - very high bandwidth (250-800 GB/s)
- Streaming Multiprocessors (SM)
  - grouping independent cores and control units
- each SM unit has
  - many ALU cores (> 100 cores)
  - lots of registers (32K-64K)
  - instruction scheduler dispatchers
  - a shared memory with very fast access to data



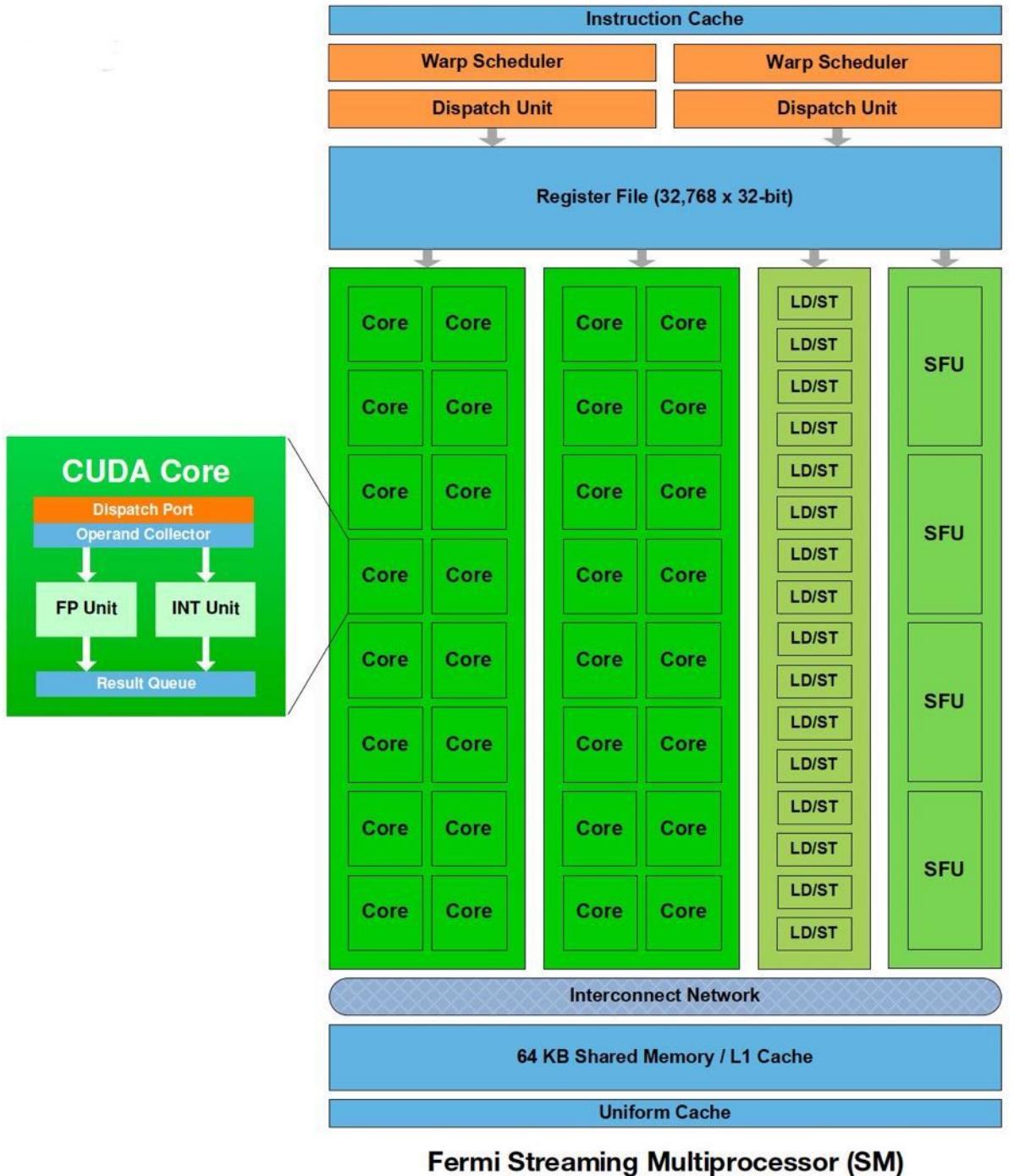
# GPU Functional Unit Types

- FP32: performs 32-bit floating point add, multiply, multiply/add, and similar instructions.
- INT32: performs 32-bit add, multiply, multiply-add, and maybe some logical operations.
- FP64: executes 64-bit FP operations
- Special Functional Unit (SFU): performs reciprocal (1x) and transcendental instructions such as sine, cosine, and reciprocal square root.
- Load/Store (LS): performs loads and stores from shared, constant, local, and global memory address spaces.
- Tensor Core: specialized units to compute  $A^T B + C$  matrix product.



# Nvidia SM

- Less scheduling units than cores
- Threads are scheduled in groups of 32, called a *warp*
- Threads within a warp always execute the same instruction in lock-step (on different data elements)
- Configurable L1 Cache/  
Shared Memory



# NVIDIA Volta V100 Architecture (2017)

- A full GV100 GPU unit contains 6 Compute Graphic Clusters (CGC) with 14 SM each, total 84 SMs
- 5376 FP32 cores
- 5376 INT32 cores
- 6MB L2 cache
- High Bandwidth Memory
  - 16 GB HBM2 SDRAM
  - 732 GB/s bandwidth
- NVLink tecnology
  - 300GB/s bandwidth to host data transfers
  - 12X respect PCIe Gen3 16x

<https://developer.nvidia.com/blog/inside-volta>



Peak Performance:  
15,7 FP32 TFlops  
Max Power Consumption: 300W

# Streaming Multiprocessor of nVIDIA Volta (2017)

- SM composed of 4 independent blocks
- each block sports:
  - 1 warps x 2 dispatchers
  - 16FP32 + 16INT32 ALU units
  - separate FP32 and INT32 cores, allowing simultaneous execution of FP32 and INT32 operations at full throughput
  - 8FP64 ALU units
  - 2 Tensor Core units (HW matmul)
  - 8 Load/Store units
  - 4 SFU units
  - 32768 32bits registers
- each block accesses:
  - 128KB for L1/shared memory
  - 4 texture units



# NVIDIA Ampere A100 Architecture (2020)

- A full GA100 GPU unit contains 8 Compute Graphic Clusters (CGC) with 16 SM each, total 128 SMs
- 8192 FP32 cores
- 8192 INT32 cores
- **40MB L2 cache**
- High Bandwidth Memory
  - **40GB HBM2**
  - 732 GB/s bandwidth
- NVLink tecnology
  - 600GB/s bandwidth to host data transfers
  - 24X respect PCIe Gen3 16x

[developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth](https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth)



Peak Performance:  
19,5 FP32 TFlops

Max Power Consumption: 400W

# Streaming Multiprocessor of nVIDIA Ampere (2020)

- SM composed of 4 independent blocks
- each block sports:
  - 1 warps x 2 dispatchers
  - 16FP32 + 16INT32 ALU units
  - separate FP32 and INT32 cores, allowing simultaneous execution of FP32 and INT32 operations at full throughput
  - 8FP64 ALU units
  - 2 Tensor Core units (HW matmul)
  - 8 Load/Store units
  - 4 SFU units
  - 32768 32bits registers
- each block accesses:
  - 192KB for L1/shared memory
  - 4 texture units



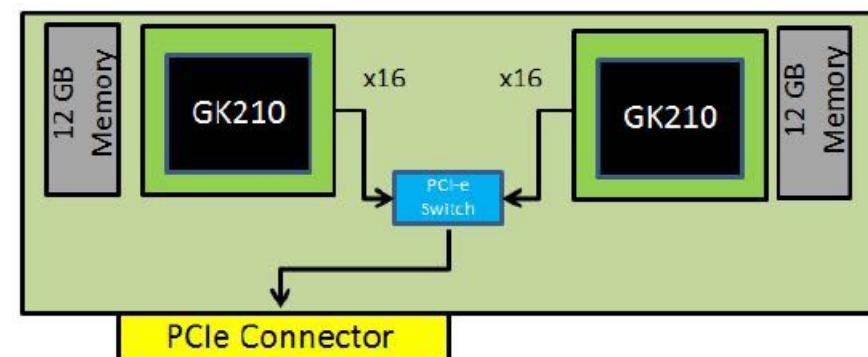
# GPU nVIDIA K80 (2013)

- Two GPUs (K40) per device

- 12GB RAM per GPU
- 480 GB/s memory bandwidth

- 15 SM per GPU
- 192 CUDA cores/SM
  - total of 2880 cuda cores

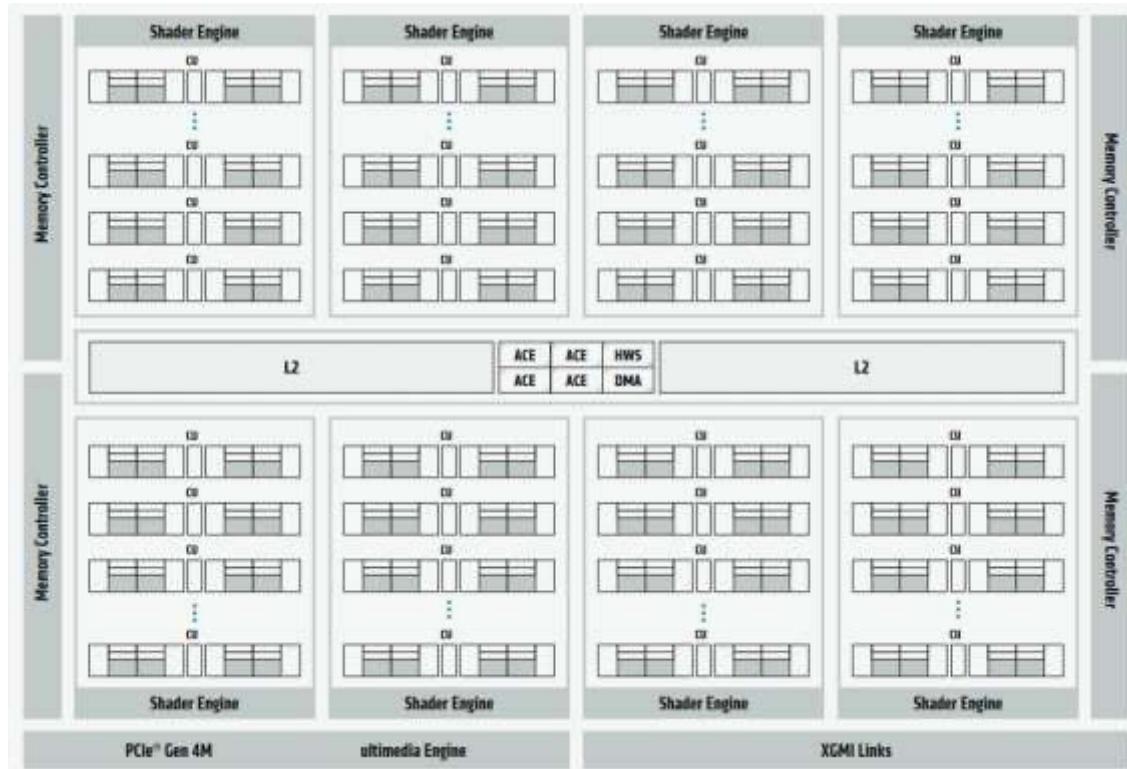
- 500-800 MHz clock
- 250W



# AMD Radeon MI100 Architecture (2020)

- A full MI100 GPU unit contains a total of 120 Compute Unit (like SMs)
- 7680 FP32 cores
- 8MB L2 cache
- High Bandwidth Memory
  - **32GB HBM2**
  - 1200 GB/s bandwidth
- **AMD Infinity Fabric**  
500GB/s bandwidth to host data transfers
  - 24X respect PCIe Gen3  
16x

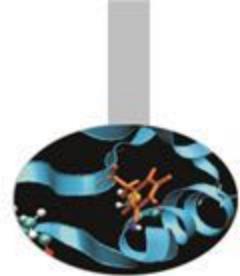
[www.amd.com](http://www.amd.com) MI100 microarchitecture



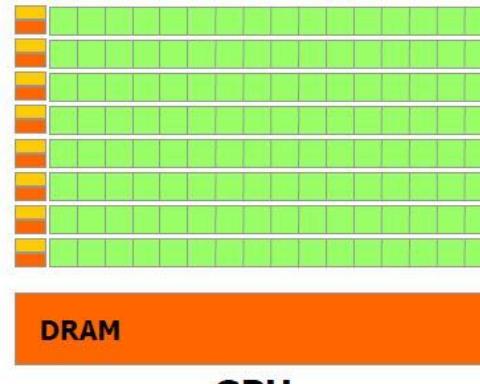
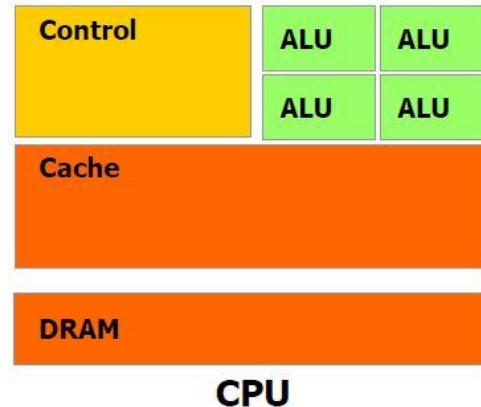
Peak Performance:  
23.0 FP32 TFlops  
Max Power Consumption: 400W

- To utilise a GPU, programs must
  - Contain parts targeted at host CPU (most lines of source code)
  - Contain parts targeted at GPU (key computational kernels)
  - Manage data transfers between distinct CPU and GPU memory spaces
  - Traditional language (e.g C/Fortran) does not provide these facilities
- To run on multiple GPUs in parallel
  - Normally use one host CPU core (thread) per GPU
  - Program manages communication between host CPUs in the same fashion as traditional parallel programs
    - e.g. MPI and/or OpenMP (latter shared memory node only)

# Different worlds: host and device



|                     | Host                                                                                                                                                  | Device                                                                                                                               |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| Threading resources | 2 threads per core (SMT), 24/32 threads per node. The thread is the atomic execution unit.                                                            | e.g.: $1536 \text{ (thd x sm)} * 14 \text{ (sm)} = 21504$ .<br>The Warp (32 thd) is the atomic execution unit.                       |
| Threads             | «Heavy» entities, context switches and resources management.                                                                                          | Extremely lightweight, managed grouped into warps, fast context switch, no resources management (statically allocated once).         |
| Memory              | e.g.: $48 \text{ GB} / 32 \text{ thd} = 1.5 \text{ GB/thd}$ , 300 cycles lat., 6.4 GB/s band (DDR3), 3 caching levels with lots of speculation logic. | e.g.: $6 \text{ GB} / 21504 \text{ thd} = 0.3 \text{ MB/thd}$ , 600 cycles lat*, 144 GB/s band (GDDR5)*, fake caches.<br>* coalesced |



# How do I program GPUs?

The situation is changing rapidly but possibilities include:

## Declarative languages

- OpenMP
  - v 4.0+ allows offloading of tasks onto GPUs
- OpenAcc
  - High-level model, particularly suited for devices such as GPUs.

## Languages

- CUDA
  - Extension to C developed by NVIDIA. With PGI compilers, FORTRAN extension also possible.
- OpenCL
  - General framework for writing programs across heterogenous devices. Often used for non-NVIDIA GPUs and FPGAs.

# GPU Programming Languages

- **CUDA (Compute Unified Device Architecture)**
  - a set of extensions to higher level programming language to use GPU as a coprocessor for heavy parallel task
  - a developer toolkit to compile, debug, profile programs and run them easily in a heterogeneous systems
- **OpenCL (Open Computing Language):**
  - a standard open-source programming model developed by major brands of hardware manufacturers (Apple, Intel, AMD/ATI, nVIDIA).
    - like CUDA, provides extenions to C/C++ and a developer toolkit
    - extensions for specific hardware (GPUs, FPGAs, MICs, etc)
    - it's very low level (verbose) programming

There are many other approaches and solutions such as SYCL (Khronos), HIP (AMD), OneAPI (Intel), DirectCompute (Microsoft), ... but current market is basically dominated by CUDA and some OpenCL

# GPU Programming Languages

Numerical analytics ➤

MATLAB Mathematica, LabVIEW

Fortran ➤

CUDA Fortran

C ➤

CUDA C

C++ ➤

CUDA C++

Python ➤

PyCUDA, Copperhead, Numba

F# ➤

Alea.cuBase



## Compute Unified Device Architecture:

- extends ANSI C language with minimal extensions
- provides application programming interface (API) to manage host and device components

## CUDA program:

- Serial sections of the code are performed by CPU (**host**)
- The parallel ones (that exhibit rich amount of *data parallelism*) are performed by GPU (**device**) in the SIMD mode as **CUDA kernels**.
- host and device have separate memory spaces: programmers need to transfer data between CPU and GPU in a manner similar to “one-sided” message passing.

# CUDA: Compute Unified Device Architecture

CUDA is a general purpose parallel computing platform and programming model that easy GPU programming, which provides:

- a hierarchical multi-threaded programming paradigm that matches GPU hardware structure
- an extensions to higher level programming languages for C/C++ and Fortran to express thread parallelism within a familiar programming environment
- a new architecture instruction set called PTX (Parallel Thread eXecution) to match GPU typical hardware
- a complete mature SDK: compiler (nvcc), debugger (cuda-gdb), profiler (nvvp), IDE (insight/eclipse/VS plugins)
- a set of GPU accelerated libraries for common scientific algorithms and requirements ...

# CUDA GPU ready scientific libraries

- dense/sparse linear algebra single/multi-GPU:  
cuBLAS, nvBLAS, cuSparse
- dense/sparse direct solver and factorizations:  
cuSOLVER
- Fast Fourier Transform (and related): cuFFT
- Random number generator: cuRAND
- Common primitives for digital signal processing and imaging elaboration: NPP (nVIDIA Performance Primitives)
- Deep Learning libraries
- ... and many, many more

# GPU Accelerated Libraries

Linear Algebra  
FFT, BLAS,  
SPARSE, Matrix



cULA tools



C U S P

Numerical & Math  
RAND, Statistics



ArrayFire



Data Struct. & AI  
Sort, Scan, Zero Sum



Visual Processing  
Image & Video



NVIDIA  
Video  
Encode



# CUDA - C

Applications

Libraries

Compiler  
Directives

Programming  
Languages

Easy to use  
Most Performance

Easy to use  
Portable code

**Most Performance**  
**Most Flexibility**

# GPGPU Programming Model

- A function which runs on a GPU is called “**kernel**”
  - when a kernel is launched on a GPU thousands of threads will execute its code
  - programmer chooses the number of threads to run
  - **each thread acts on a different data element independently**
  - the GPU parallelism is very close to the SPMD paradigm

```
void vecAddCPU (int N, const float *A,
 const float *B, float *C)
{
 for (int i = 0; i < N; i++)
 c[i] = a[i] + b[i];
}
...
// call vecAddCPU on N elements
vecAddCPU (N, a, b, c);
```

```
void vecAddGPU (int N, const float *A,
 const float *B, float *C)
{
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 if (i < N) c[i] = a[i] + b[i];
}
...
// call vecAddGPU on N elements
vecAddGPU<<<1, N>>>(N, a, b, c);
```

# Asynchronous execution

By default, GPU operations are **asynchronous**.

- When you call a function that uses the GPU, the operations are enqueued to the particular device, but not necessarily executed until later.
- This allows us to execute more computations in parallel, including operations on CPU or other GPUs.

Instead they are **synchronous** if

- The environment variable CUDA\_LAUNCH\_BLOCKING equals to 1.
- using a profiler(nvprof), without enabling concurrent kernel profiling
- memcpy that involve host memory which is not page-locked.

# A first program

```
#include <stdio.h>

void CPUFunction() {
 printf("Hello world from the CPU.\n");
}

__global__ void GPUFunction() {
 printf("Hello world from the GPU.\n");
}

int main() {
 CPUFunction();
 GPUFunction<<<1, 1>>>();
 cudaDeviceSynchronize();
}
```

Try to remove one of the following at a time and see what happens

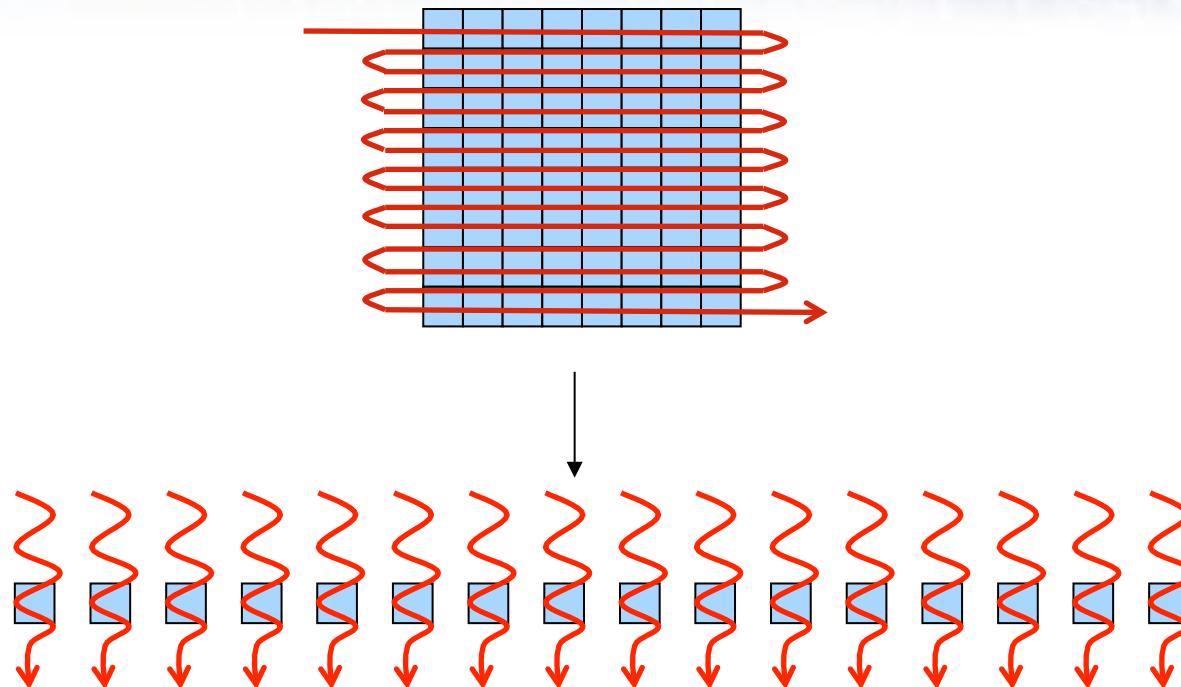
- `__global__`
- `<<<1,1>>>`
- `cudaDeviceSynchronize();`

COMPILE with

```
nvcc -o first first.cu -run
```

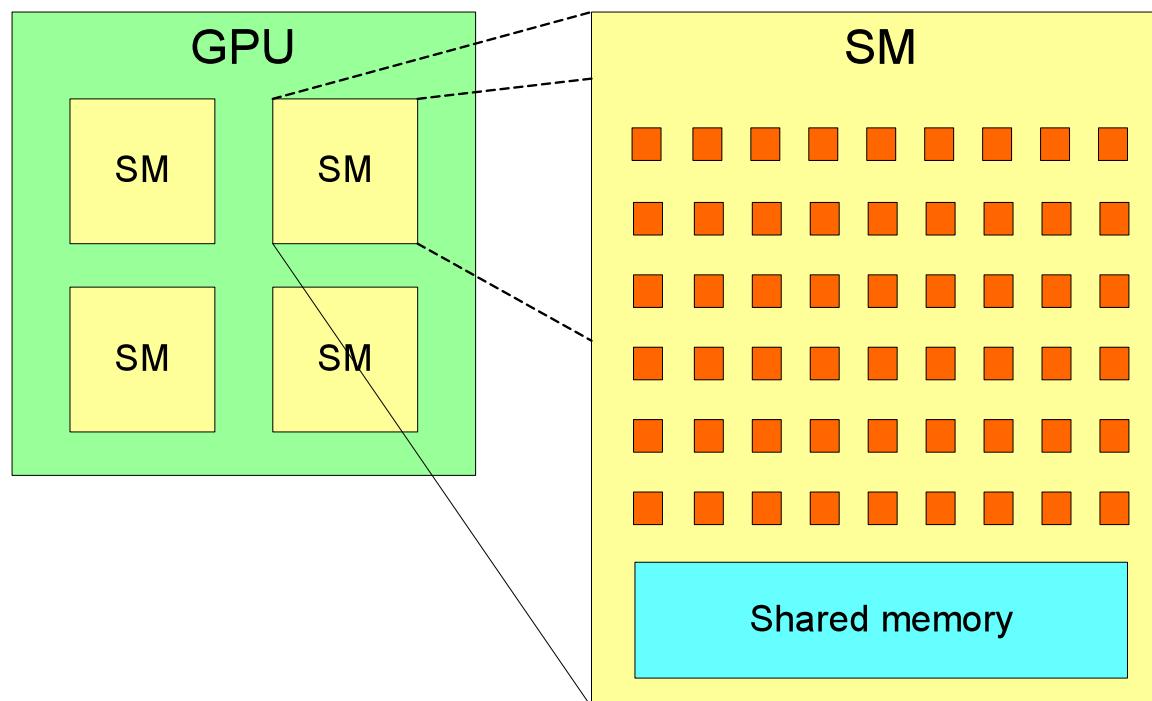
# GPGPU: Stream Computing

|epcc|



- Data set decomposed into a *stream* of elements
- A single computational function (*kernel*) operates on each element
  - “thread” defined as execution of kernel on one data element
- Multiple cores can process multiple elements in parallel
  - i.e. many threads running in parallel
- Suitable for data-parallel problems

- NVIDIA GPUs have a 2-level hierarchy:
  - Multiple Stream Multiprocessors SMs
    - each with multiple cores



- In CUDA, this is abstracted as *Grid of Thread Blocks*
  - The multiple **blocks** in a grid map onto the multiple **SMs**
    - Each block in a grid contains multiple **threads**, mapping onto the **cores** in an SM
- We don't need to know the exact details of the hardware (number of SMs, cores per SM).
  - Instead, *oversubscribe*, and system will perform scheduling automatically
    - Use more blocks than SMs, and more threads than cores
  - Same code will be portable and efficient across different GPU versions.

# CUDA Kernel Launch Parameters Syntax

Triple chevron launch syntax <<< >>> contains  
“kernel launch parameters”

vecAddGPU<<< 1, 1024 >>>( N, a, b, c )

1° parameter defines the number of **blocks** to use

2° parameter defines the number of **threads per block**

```
void vecAddGPU (int N, const float *A,
 const float *B, float *C)
{
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 if (i < N) c[i] = a[i] + b[i];
}
...
// call vecAddGPU on N elements
vecAddGPU<<<1, N >>>(N, a, b, c);
```

# A second program

```
#include <stdio.h>

__global__ void GPUfunction()
{
 printf("This is running in parallel.\n");
}

int main()
{
 GPUfunction <<<5, 5>>>();
 cudaDeviceSynchronize();
}
```

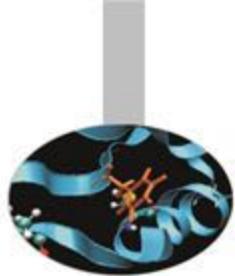
Try

- <<<1, 1>>>
- <<<1,10>>>
- <<<10, 1>>>
- <<<10, 10>>>
- and, again, remove cudaDeviceSynchronize();

COMPILE with

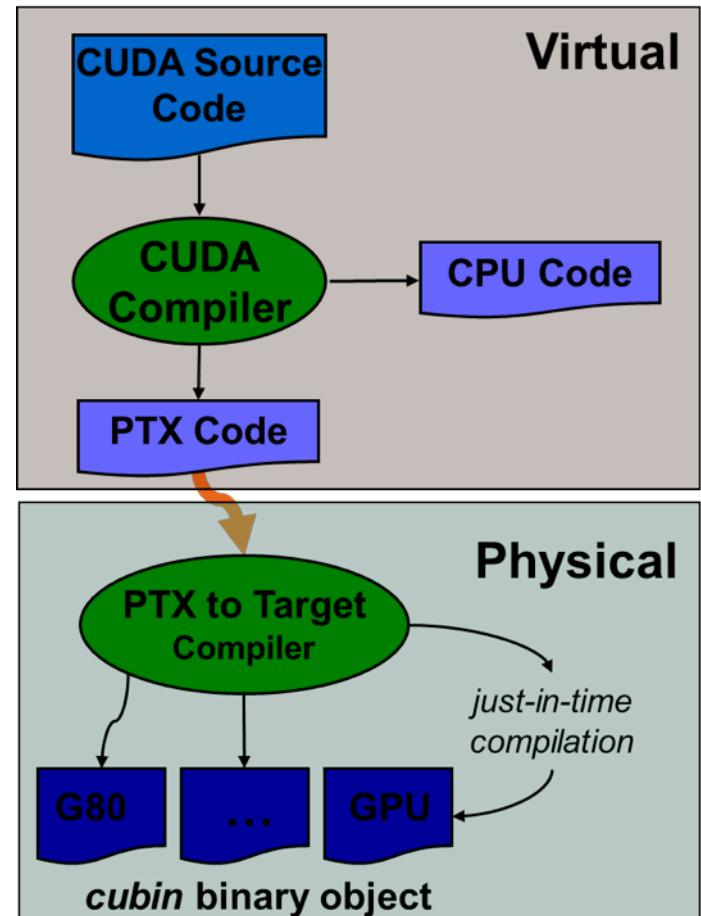
```
nvcc -o second second.cu -run
```

# NVIDIA C compiler



nvcc front-end for compilation:

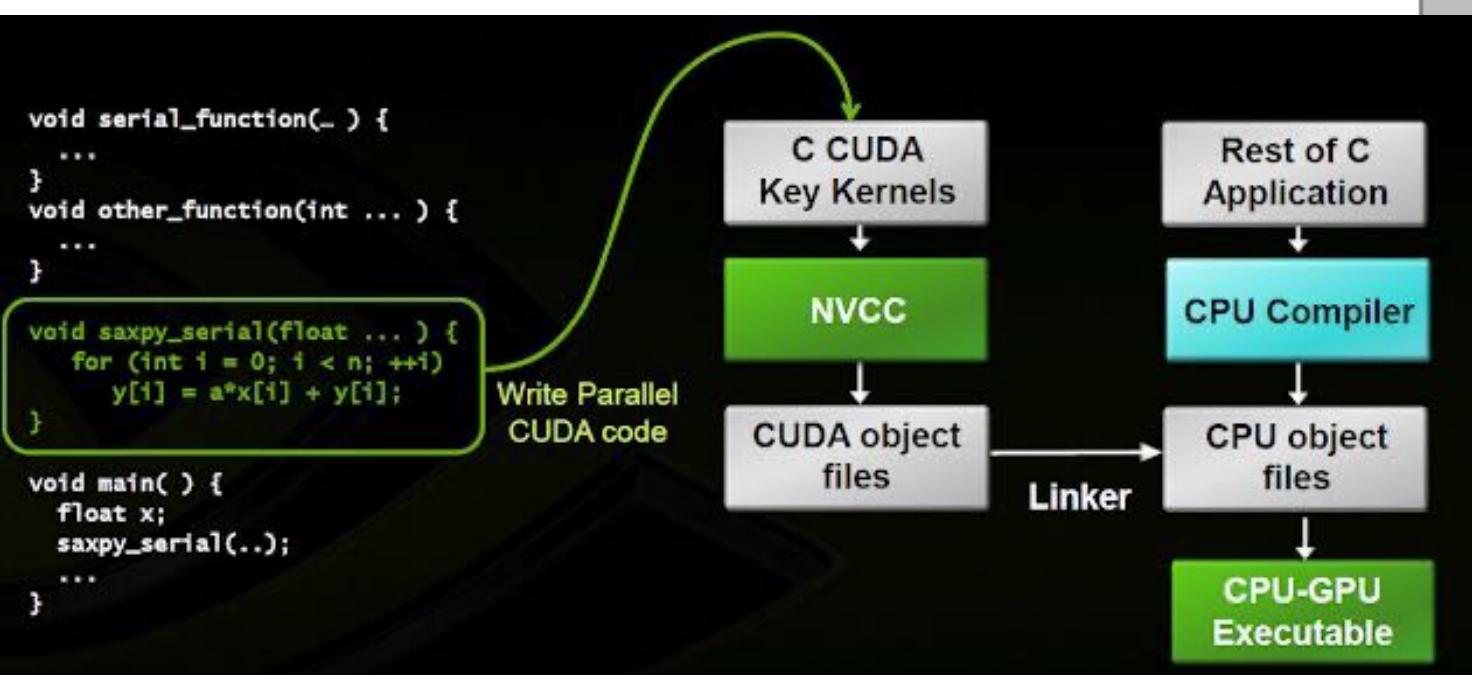
- separates GPU code from CPU code
- CPU code -> C/C++ compiler (Microsoft Visual C/C++, GCC, ecc.)
- GPU code is converted in an intermediate assembly language: PTX, then in binary form (the *cubin* object)
- link all executables



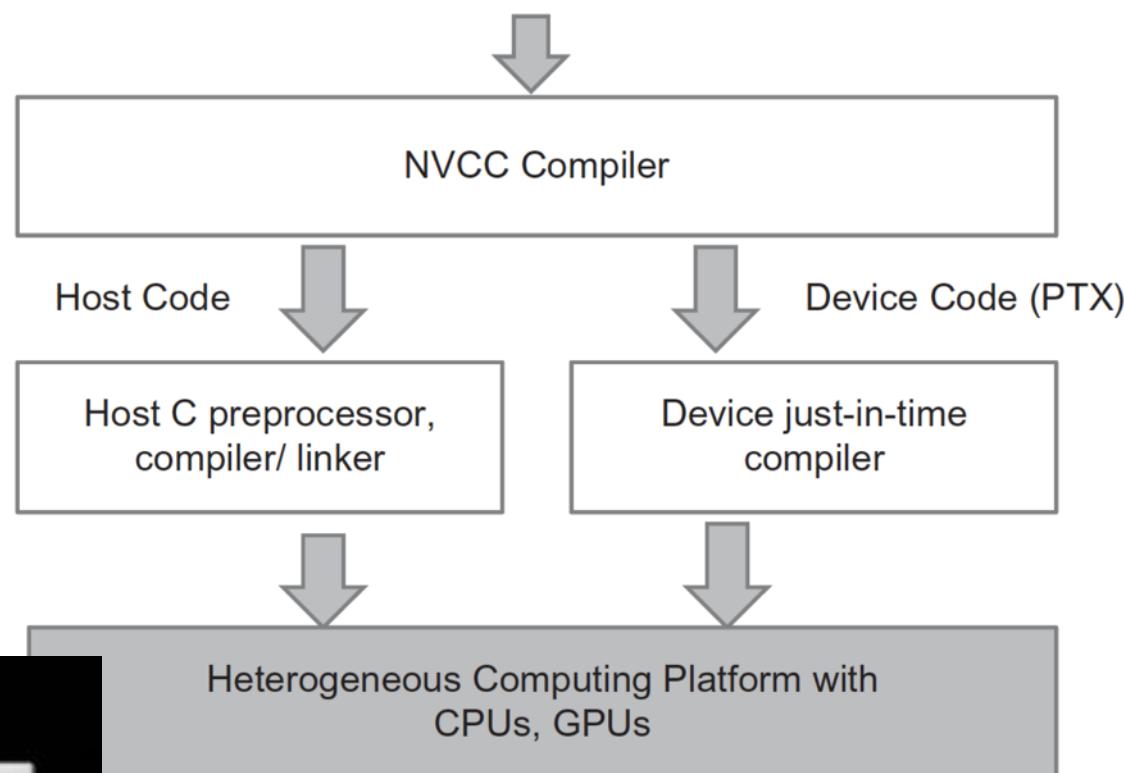
# How to compile

```
nvcc myprog.cu -o myprog
```

*Nvcc only parses .cu files for CUDA*



Integrated C programs with CUDA extensions

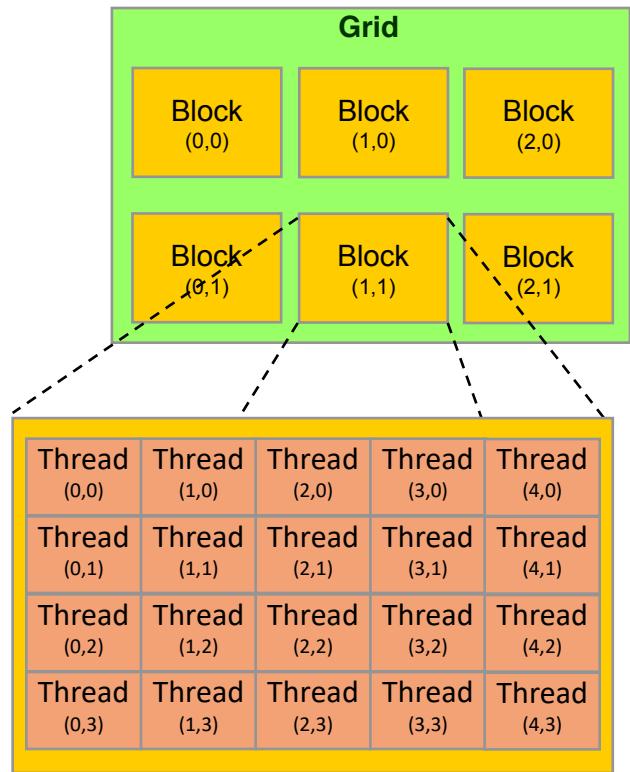


# Device Code

- CUDA keyword `__global__` indicates a kernel function that:
  - Runs on the device.
  - Called from the host.
- CUDA keyword `__device__` indicates a device function that:
  - Runs on the device.
  - Called from a kernel function or another device function.
- Triple angle brackets `<<< >>>` indicate a call from host code to device code.
  - Kernel launch
- `nvcc` separates source code into two components:
  - Device functions are processed by NVIDIA compiler.
  - Host functions are processed by standard host compiler.
  - `$ nvcc hello.cu`

# GPU Thread Hierarchy

- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device
- GPU threads are grouped together in teams or blocks of threads
- Threads belonging to the same block or team can cooperate together exchanging data through a shared memory cache area
- each block of threads will be executed independently
- no assumption is made on the blocks execution order



# GPU Thread Hierarchy

- threads are organized into blocks of threads
  - blocks can be 1D, 2D, 3D sized in threads
  - blocks can be organized into a 1D, 2D, 3D grid of blocks
- Blocks are organized in a grid of blocks
- each block or thread has a unique ID
  - use .x, .y, .z to access its components

## threadIdx:

thread coordinates inside the block

## blockIdx:

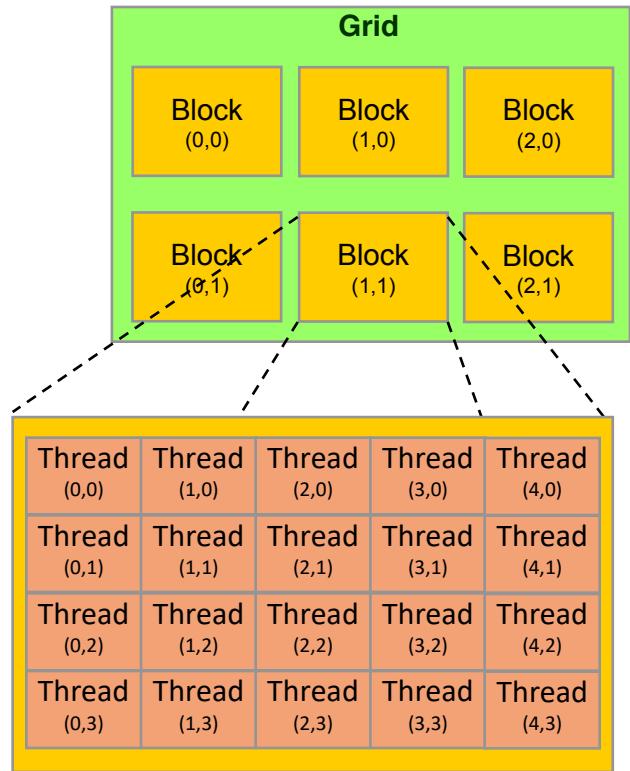
block coordinates inside the grid

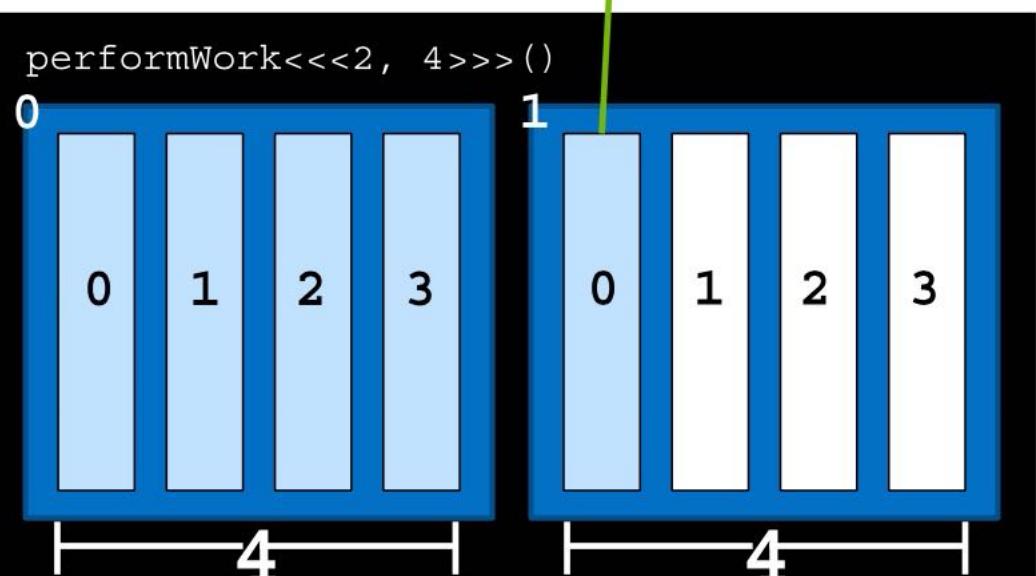
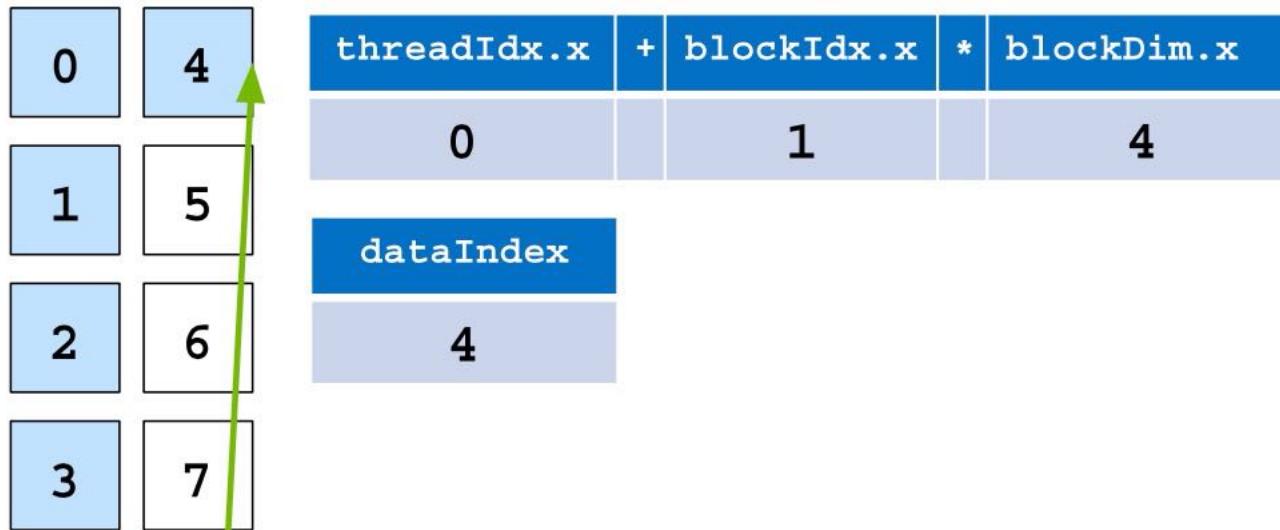
## blockDim:

block dimensions in thread units

## gridDim:

grid dimensions in block units

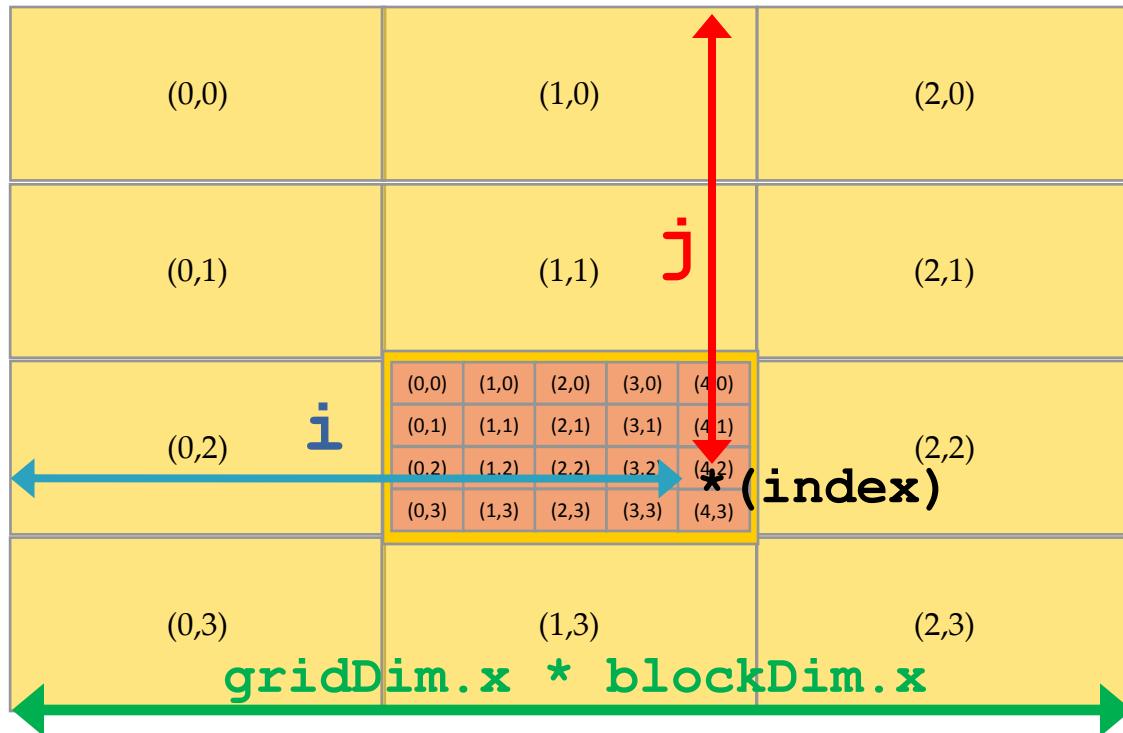




This idiomatic expression gives each thread a unique index within the entire grid.

`int i = blockIdx.x * blockDim.x + threadIdx.x;`

# CUDA Thread Grid



threadIdx:  
thread coordinates inside a block

blockIdx:  
block coordinates inside the grid

blockDim:  
block dimensions in thread units

gridDim:  
grid dimensions in block units

```
i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

index = j * gridDim.x * blockDim.x + i;
```

- You can think of this as restructuring the original loop

```
for (i=0;i<N;i++) {
 c[i] = a[i] + b[i];
}
```

as a set of ( $N/32$ ) blocks composed by 32 threads each

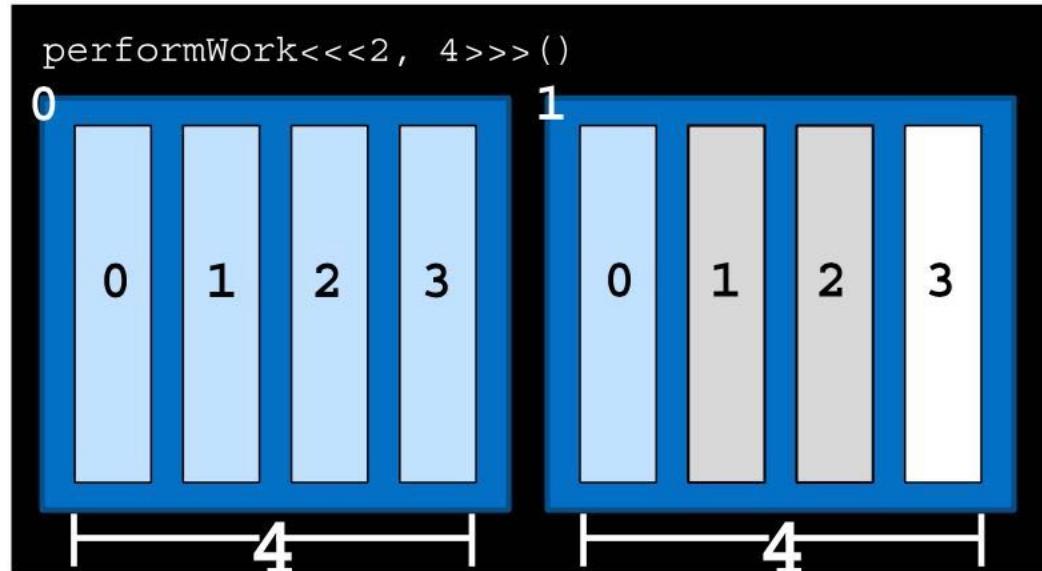
```
for (i0=0;i0<((N+31)/32) ;i0++) {
 for (i1=0;i1<32 ;i1++) {
 i = i0*32 + i1;
 c[i] = a[i] + b[i];
 }
}
```

If  $N \% 32 \neq 0$ , e.g.  $100 \% 32 == 4$ , you need  $N/32 + 1$  blocks, with this technique you avoid to check %  
-  $100 / 32 == 3,125$ , so 4 blocks  
-  $(100+31) / 32 == 4,09 \Rightarrow 4$  blocks

But we must check that  $i < N$ , because  $4 * 32 == 128$

and parallelising inner loop over threads in a block, outer loop over blocks.

|   |   |             |   |            |   |            |
|---|---|-------------|---|------------|---|------------|
| 0 | 4 | threadIdx.x | + | blockIdx.x | * | blockDim.x |
|   |   | 2           |   | 1          |   | 4          |
| 1 |   | dataIndex   | < | N          | = | Can work   |
| 2 |   | 6           |   | 5          |   | false      |
| 3 |   |             |   |            |   |            |



# CUDA Kernel Launch Parameters Syntax

```
void vecAddGPU (int N, const float *A,
 const float *B, float *C)
{
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 if (i < N) c[i] = a[i] + b[i];
}

...
// call vecAddGPU on N elements
dim3 threads(32);
dim3 blocks ((N+threads.x-1)/threads.x);
vecAddGPU<<< blocks, threads >>>(N, a, b, c);
```

Threads 99-127 will not compute anything as expected

Es. N = 100  
 $100/32 = 3.125$ , but  $3*32 = 96 < 100$   
 $(100+31)/32 = 4.09$ , and  $4*32 = 128 > 100$

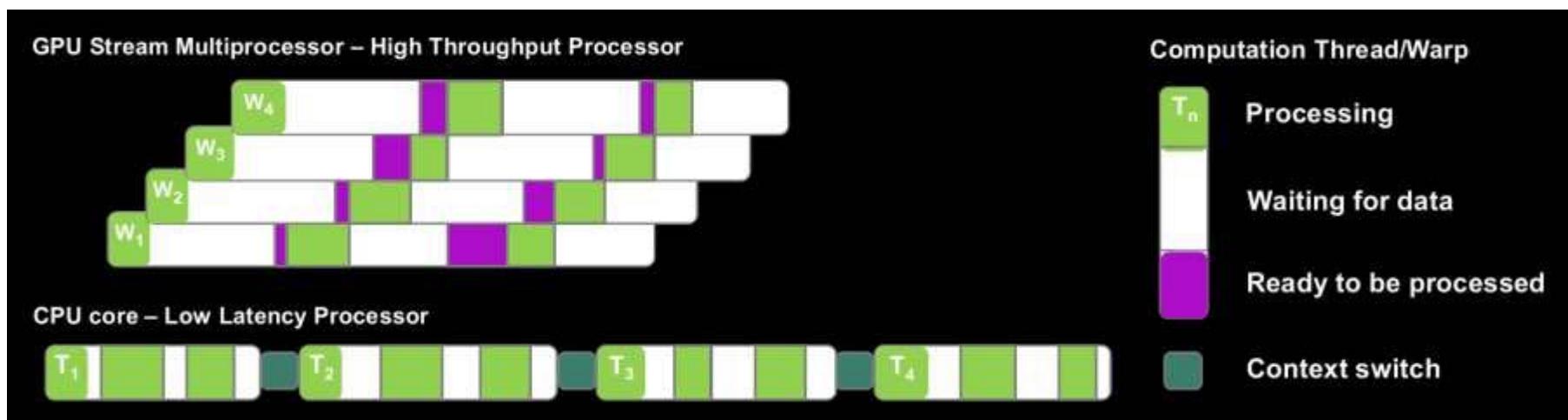
for **blockIdx.x** = 0  
 $i = 0 * 32 + \text{threadIdx.x} = \{ 0, 1, 2, \dots, 31 \}$

for **blockIdx.x** = 1  
 $i = 1 * 32 + \text{threadIdx.x} = \{ 32, 33, 34, \dots, 63 \}$

for **blockIdx.x** = 2  
 $i = 2 * 32 + \text{threadIdx.x} = \{ 64, 65, 66, \dots, 95 \}$

# CUDA Programming Model

- GPU threads are extremely *light weight*
  - no penalty in case of a *context-switch*
  - each thread has its own registers
- the more are the threads *in flight*, the more the GPU hardware is able to hide memory or computational latencies



## 2D Example

- The previous examples were one dimensional.
- Each thread block can be 1D, 2D or 3D to best fit the algorithm, e.g. for matrix addition:

```
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
 int i = threadIdx.x;
 int j = threadIdx.y;

 c[i][j] = a[i][j] + b[i][j];
}

int main()
{
 dim3 blocksPerGrid(1); /* 1 block per grid (1D) */
 dim3 threadsPerBlock(N, N); /* NxN threads per block (2D) */
 matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```

- **dim3** is a CUDA type, containing 3 integers (x,y and z components)

# Multiple Block 2D Example

- Grid can also be be 1D, 2D or 3D

```
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;

 c[i][j] = a[i][j] + b[i][j];
}

int main()
{
 dim3 blocksPerGrid(N/16,N/16); // (N/16)x(N/16) blocks/grid (2D)
 dim3 threadsPerBlock(16, 16); /. 16x16 threads/block (2D)
 matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```

# Grid-strided loops

- If more elements than threads

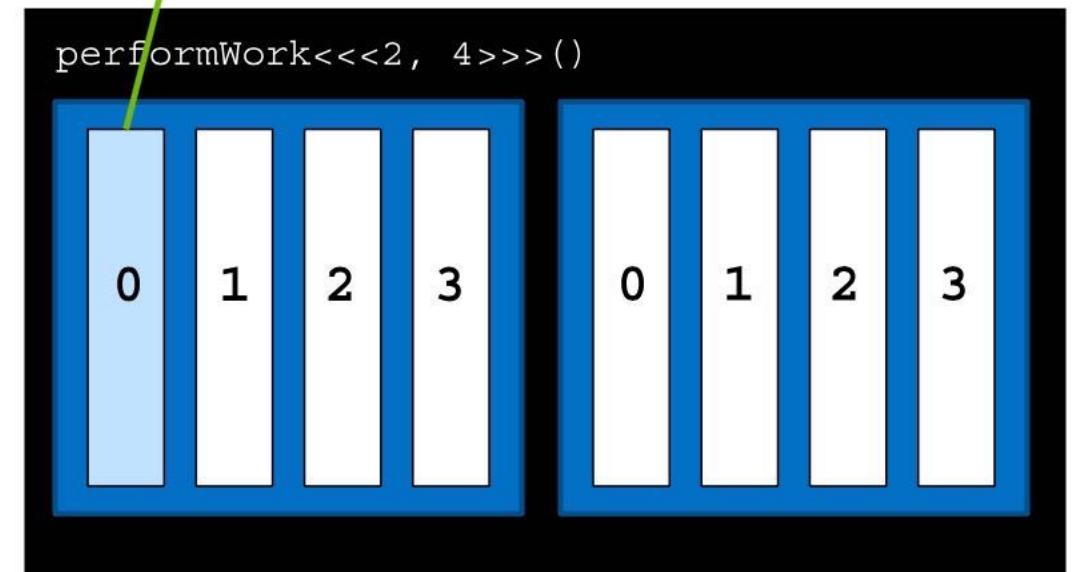
```
__global__ void kernel(int *a, int N) {

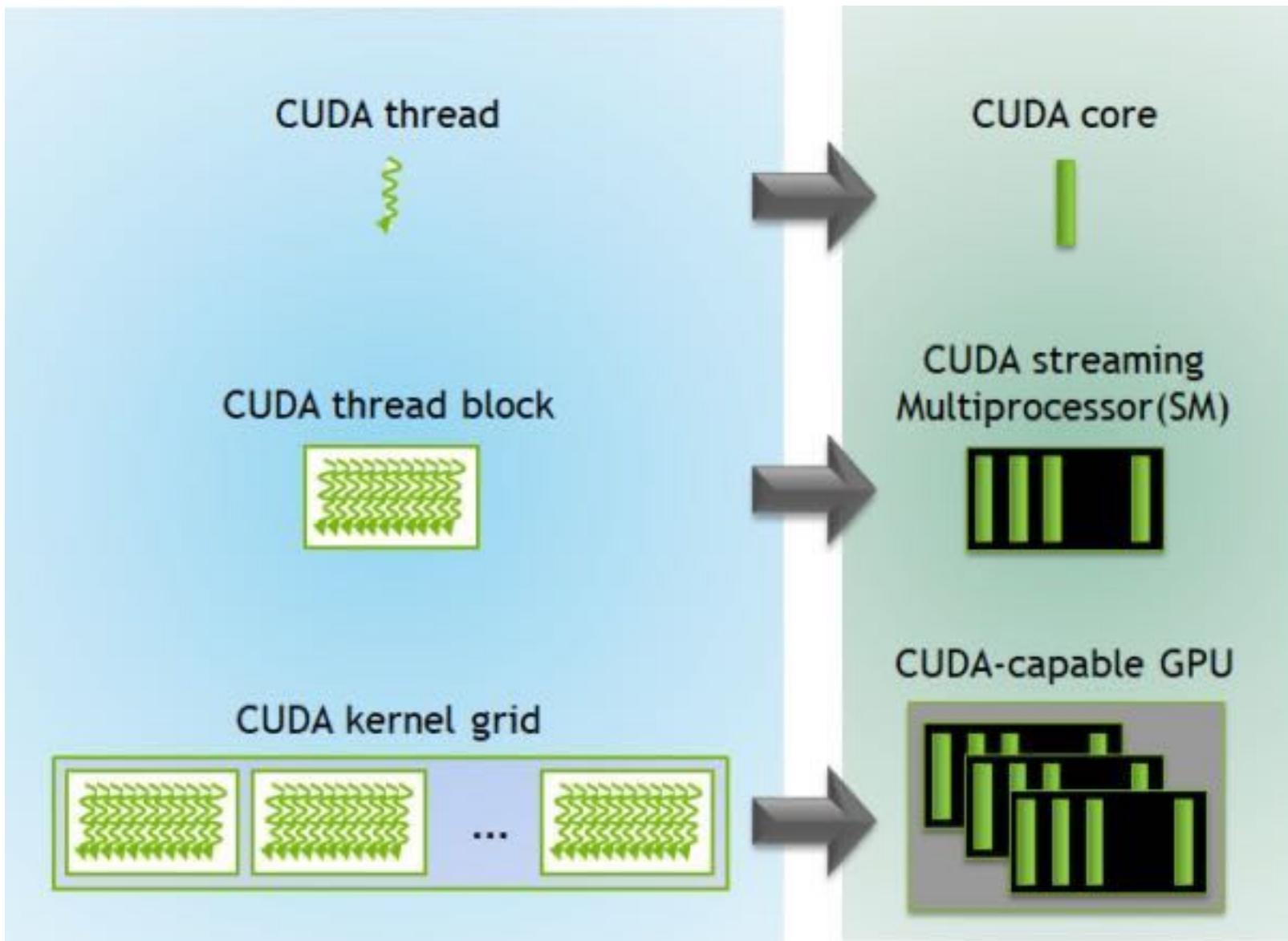
 int indexWithinTheGrid = threadIdx.x + blockIdx.x * blockDim.x;

 int gridStride = gridDim.x * blockDim.x;

 for (int i = indexWithinTheGrid; i < N; i += gridStride) {
 // do work on a[i];
 }
}
```

|   |   |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|
| 0 | 4 | 8  | 12 | 16 | 20 | 24 | 28 |
| 1 | 5 | 9  | 13 | 17 | 21 | 25 | 29 |
| 2 | 6 | 10 | 14 | 18 | 22 | 26 | 30 |
| 3 | 7 | 11 | 15 | 19 | 23 | 27 | 31 |





Device 0: "GeForce GTX 1650"

|                                                |                                                                                          |
|------------------------------------------------|------------------------------------------------------------------------------------------|
| CUDA Driver Version / Runtime Version          | 11.1 / 11.1                                                                              |
| CUDA Capability Major/Minor version number:    | 7.5                                                                                      |
| Total amount of global memory:                 | 3904 MBytes (4093509632 bytes)                                                           |
| (14) Multiprocessors, ( 64) CUDA Cores/MP:     | 896 CUDA Cores                                                                           |
| GPU Max Clock rate:                            | 1695 MHz (1.70 GHz)                                                                      |
| Memory Clock rate:                             | 4001 Mhz                                                                                 |
| Memory Bus Width:                              | 128-bit                                                                                  |
| L2 Cache Size:                                 | 1048576 bytes                                                                            |
| Maximum Texture Dimension Size (x,y,z)         | 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)                                |
| Maximum Layered 1D Texture Size, (num) layers  | 1D=(32768), 2048 layers                                                                  |
| Maximum Layered 2D Texture Size, (num) layers  | 2D=(32768, 32768), 2048 layers                                                           |
| Total amount of constant memory:               | 65536 bytes                                                                              |
| Total amount of shared memory per block:       | 49152 bytes                                                                              |
| Total shared memory per multiprocessor:        | 65536 bytes                                                                              |
| Total number of registers available per block: | 65536                                                                                    |
| Warp size:                                     | 32                                                                                       |
| Maximum number of threads per multiprocessor:  | 1024                                                                                     |
| Maximum number of threads per block:           | 1024                                                                                     |
| Max dimension size of a thread block (x,y,z):  | (1024, 1024, 64)                                                                         |
| Max dimension size of a grid size (x,y,z):     | (2147483647, 65535, 65535)                                                               |
| Maximum memory pitch:                          | 2147483647 bytes                                                                         |
| Texture alignment:                             | 512 bytes                                                                                |
| Concurrent copy and kernel execution:          | Yes with 3 copy engine(s)                                                                |
| Run time limit on kernels:                     | No                                                                                       |
| Integrated GPU sharing Host Memory:            | No                                                                                       |
| Support host page-locked memory mapping:       | Yes                                                                                      |
| Alignment requirement for Surfaces:            | Yes                                                                                      |
| Device has ECC support:                        | Disabled                                                                                 |
| Device supports Unified Addressing (UVA):      | Yes                                                                                      |
| Device supports Managed Memory:                | Yes                                                                                      |
| Device supports Compute Preemption:            | Yes                                                                                      |
| Supports Cooperative Kernel Launch:            | Yes                                                                                      |
| Supports MultiDevice Co-op Kernel Launch:      | Yes                                                                                      |
| Device PCI Domain ID / Bus ID / location ID:   | 0 / 1 / 0                                                                                |
| Compute Mode:                                  | < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) > |

Nvidia TURING

- arch=sm\_75
- compute\_75

So, for example,

nvcc -arch=sm\_75 -o first first.cu -run

- The GPU has a separate memory space from the host CPU
- We cannot simply pass normal C pointers to CUDA threads
- Need to manage GPU memory and copy data to and from it explicitly
- `cudaMalloc` is used to allocate GPU memory
- `cudaFree` releases it again

```
float *a;

cudaMalloc (&a, N*sizeof(float));

...

cudaFree (a);
```

- Once we've allocated GPU memory, we need to be able to copy data to and from it
- cudaMemcpy does this:

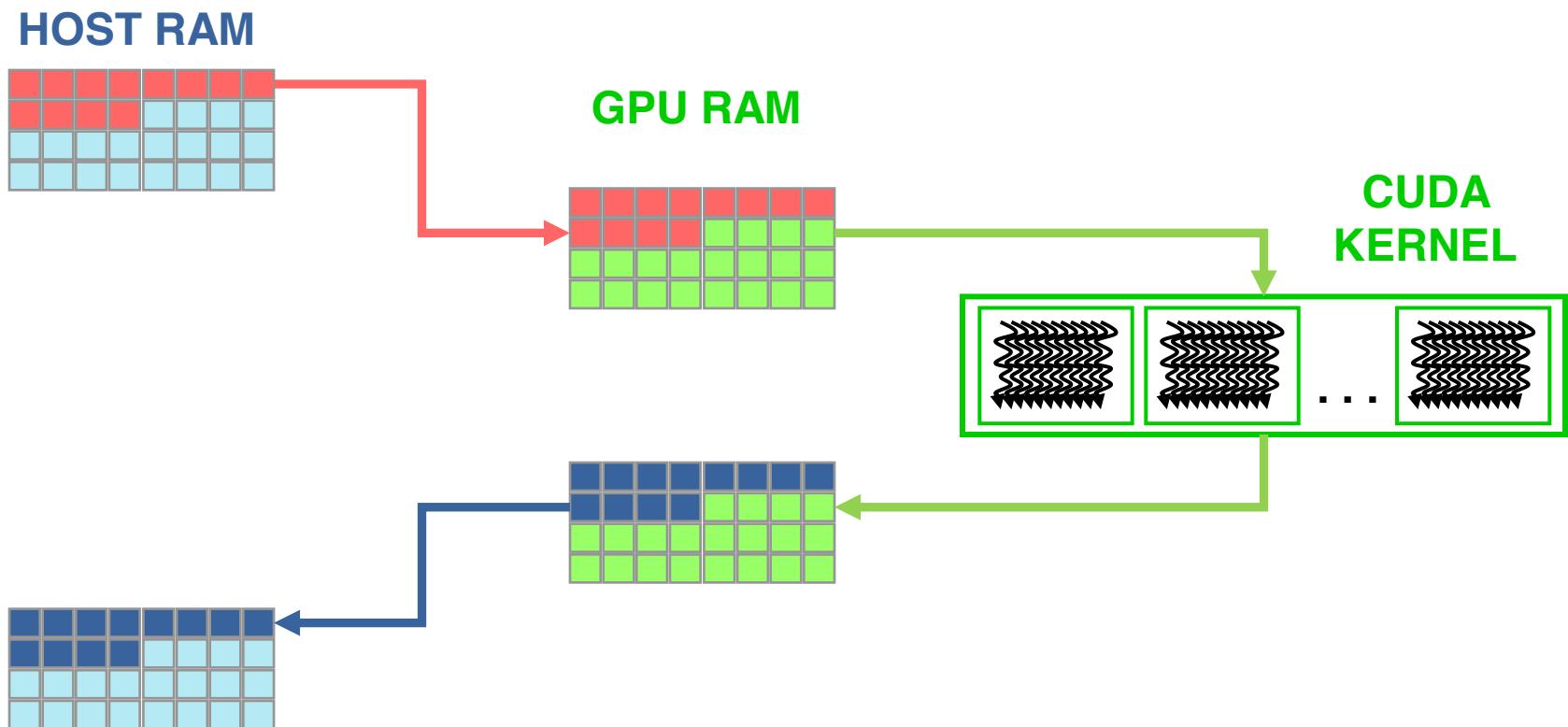
```
cudaMemcpy(array_device, array_host, N*sizeof(float),
 cudaMemcpyHostToDevice);

cudaMemcpy(array_host, array_device, N*sizeof(float),
 cudaMemcpyDeviceToHost);
```

- The first argument always corresponds to the *destination* of the transfer.
- Transfers between host and device memory are relatively slow and can become a bottleneck, so should be minimised when possible

# Data movement

- data must be moved from HOST to DEVICE memory in order to be processed by a CUDA kernel
- when data is processed, and no more needed on the GPU, it is transferred back to HOST



# The full example

```
#include <stdio.h>
#include <sys/time.h>
#define N (32 * 1024)

__global__ void add(int *a, int *b, int *c) {
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 if (tid < N) c[tid] = a[tid] + b[tid];
}

int main(void) {
 int *a, *b, *c, *dev_a, *dev_b, *dev_c;
 struct timeval t1, t2;
 dim3 threads(32);
 dim3 blocks ((N+threads.x-1)/threads.x);

 a = (int*)malloc(N * sizeof(int)); //the same for b and c
 cudaMalloc((void**)&dev_a, N * sizeof(int));
 //the same for dev_b and dev_c

 for (int i=0; i<N; i++) { a[i] = i; b[i] = 2 * i; }
 gettimeofday(&t1, 0);

 // copy the arrays 'a' and 'b' to the GPU
 cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, N * sizeof(int),
 cudaMemcpyHostToDevice);

 add<<<blocks,threads>>>(dev_a, dev_b, dev_c);

 // copy the array 'c' back from the GPU to the CPU
 cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
 cudaDeviceSynchronize();
 gettimeofday(&t2, 0);

 printf("We did it!\n");
 double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-
 t1.tv_usec)/1000.0;
 printf("Time to generate: %3.1f ms \n", time);

 // free the memory
 cudaFree(dev_a);
 free(a); //the same for b and c
 return 0;
}
```

# CUDA 6.x - Unified Memory

- Unified Memory creates a pool of memory with an address space that is shared between the CPU and GPU. In other word, a block of Unified Memory is accessible to both the CPU and GPU by using the same pointer;
- the system automatically *migrates* data allocated in Unified Memory mode between the host and device memory
  - no need to explicitly declare device memory regions
  - no need to explicitly copy back and forth data between CPU and GPU devices
  - greatly simplifies programming and speeds up CUDA ports
- REM: it can result in performances degradation with respect to an explicit, finely tuned data transfer.

# SINGLE POINTER

## Explicit vs Unified Memory

### Explicit Memory Management

```
void *data, *d_data;
data = malloc(N);
cudaMalloc(&d_data, N);
cpu_func1(data, N);
cudaMemcpy(d_data, data, N, ...)
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N, ...)
cudaFree(d_data);
cpu_func3(data, N);

free(data);
```

### GPU code w/ Unified Memory

```
void *data;
data = malloc(N);

cpu_func1(data, N);

gpu_func2<<<...>>>(data, N);
cudaDeviceSynchronize();

cpu_func3(data, N);

free(data);
```

# Sample code using CUDA Unified Memory

## CPU code

```
void sortfile (FILE *fp, int N) {
 char *data;

 data = (char *) malloc (N);

 fread(data, 1, N, fp);

 qsort(data, N, 1, compare);

 use_data(data);

 free(data)
}
```

## GPU code

```
void sortfile(FILE *fp, int N) {
 char *data;

 cudaMallocManaged(&data, N);

 fread(data, 1, N, compare);

 qsort<<< ... >>> (data, N, 1, compare);

 cudaDeviceSynchronize();

 use_data(data);

 cudaFree(data);
}
```

# Checking CUDA Errors

- All CUDA API returns an error code of type **cudaError\_t**
  - Special value **cudaSuccess** means that no error occurred
- CUDA runtime has a convenience function that translates a CUDA error into a readable string with a human understandable description of the type of error occurred

```
char* cudaGetString(cudaError_t code)
```

```
cudaError_t cerr = cudaMalloc(&d_a, size);

if (cerr != cudaSuccess)
 fprintf(stderr, "%s\n", cudaGetString(cerr));
```

- CUDA Asynchronous API returns an error which refers only on errors which may occur during the call on host
- CUDA kernels are asynchronous and void type so they don't return any error code

# Checking Errors for CUDA kernels

- The error status is also held in an internal variable, which is modified by each CUDA API call or kernel launch.
- CUDA runtime has a function that returns the status of internal error variable.

```
cudaError_t cudaGetLastError(void)
```

1. Returns the status of internal error variable (`cudaSuccess` or other)
2. Resets the internal error status to `cudaSuccess`
  - Error code from `cudaGetLastError` may refers to any other preceding CUDA API runtime calls
  - To check the error status of a CUDA kernel execution, we have to wait for kernel completion using the following synchronization API:

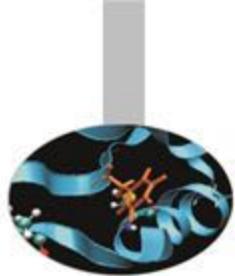
```
cudaDeviceSynchronize()
```

```
// reset internal state
cudaError_t cerr = cudaGetLastError();
// launch kernel
kernelGPU<<<dimGrid, dimBlock>>>(....);
cudaDeviceSynchronize();
cerr = cudaGetLastError();
if (cerr != cudaSuccess)
 fprintf(stderr, "%s\n",
 cudaGetStringError(cerr));
```

# Checking CUDA Errors

- Error checking is strongly encouraged during developer phase
- Error checking may introduce overhead and unpleasant synchronizations during production run
- Error check code can become very verbose and tedious
  - A common approach is to define a assert style preprocessor macro which can be turned on/off in a simple manner

```
#define CUDA_CHECK(X) { \
 cudaError_t _m_cudaStat = X; \
 if(cudaSuccess != _m_cudaStat) { \
 fprintf(stderr, "\nCUDA_ERROR: %s in file %s line %d\n", \
 cudaGetErrorString(_m_cudaStat), __FILE__, __LINE__); \
 exit(1); \
 } \
} \
... \
CUDA_CHECK(cudaMemcpy(d_buf, h_buf, bufferSize, cudaMemcpyHostToDevice));
```



# Development tools

## Common

- ⋮ Memory Checker
- ⋮ Built-in profiler
- ⋮ Visual Profiler

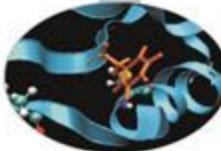
## Linux

- ⋮ CUDA GDB
- ⋮ Parallel Nsight for Eclipse

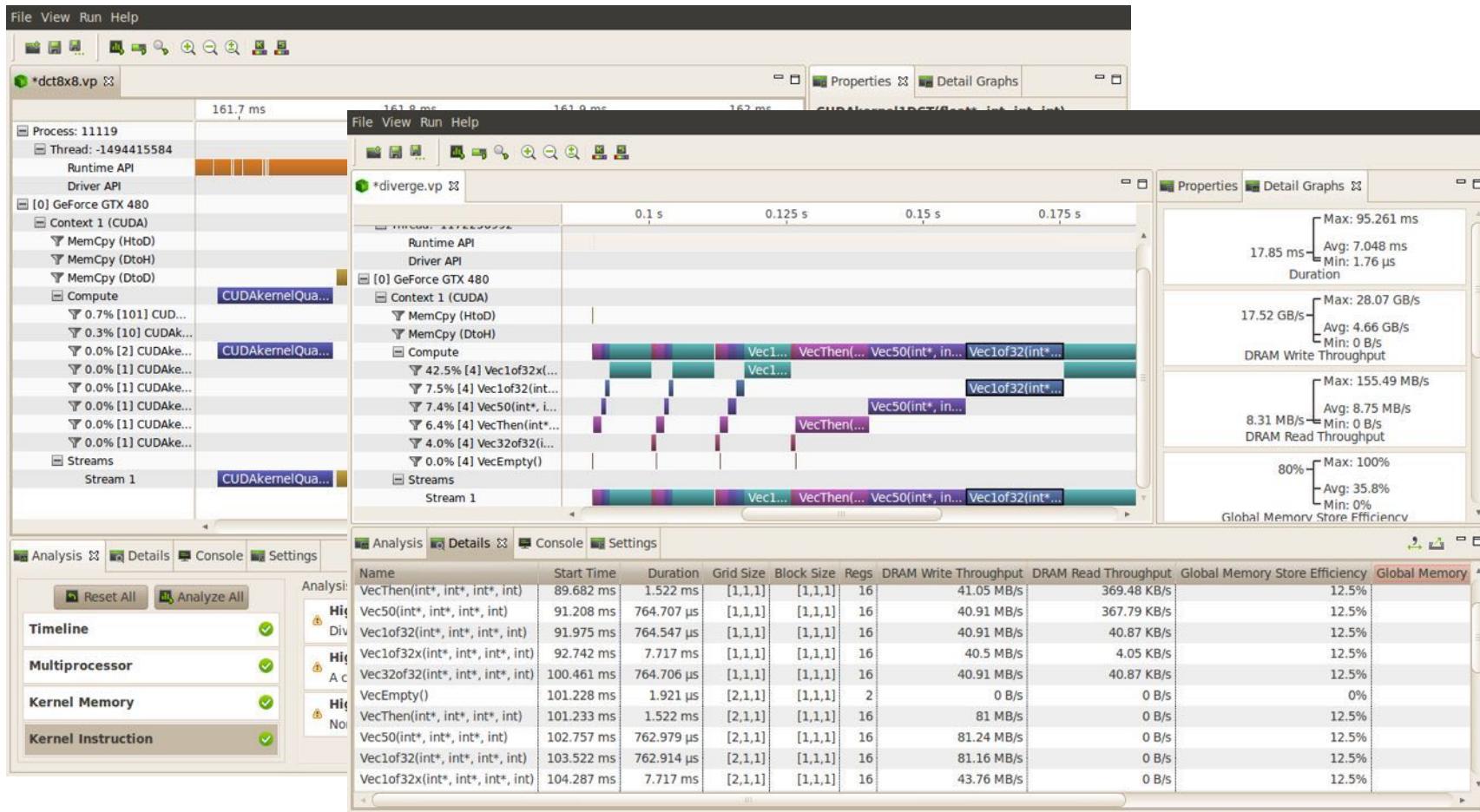
## Windows

- ⋮ Parallel Nsight for VisualStudio

# Profiling: Visual Profiler

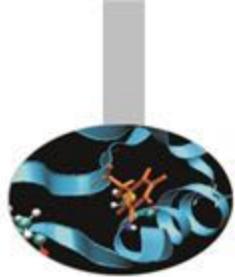


- Traces execution at host, driver and kernel levels (unified timeline)
- Supports automated analysis (hardware counters)



# Parallel NSight

<https://developer.nvidia.com/tools-overview>

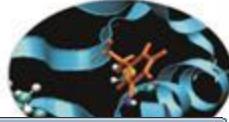


- ➊ Plug-in for major IDEs (Eclipse and VisualStudio)
- ➋ Aggregates all external functionalities:
  - ➌ Debugger (fully integrated)
  - ➌ Visual Profiler
  - ➌ Memory correctness checker
- ➌ As a plug-in, it extends all the convenience of IDEs to CUDA

On Windows systems:

- ➊ Now works on a single GPU
- ➋ Supports remote debugging and profiling
- ➌ Latest version (2.2) introduced live PTX assembly view, warp inspector and expression lamination

# Parallel NSight



oxelpipe\_demo\_vc10 (Debugging) - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Team Nsight Data Tools Test Analyze Window Help

Process: [1840] voxelpipe\_demo.exe Thread: [2874912] <No Name> Stack Frame: CUmodule 05508fe0 - [2] trace - Line 148

Connections: localhost

**CUDA Info 1**

Warp Watch 1

**CUDA Info 1**

**Modules**

rt\_render.cu

(Unknown Scope)

143                    node\_index = node.get\_index(); // jump to child

144        }

145        else

146        {

147                // leaf intersection

148                const uint32 leaf\_index = node.get\_index();

149                const Bvh\_leaf leaf = geometry.m\_bvh\_leaves[ leaf\_index ];

150                const uint32 leaf\_end = leaf.get\_index() + leaf.get\_size();

151                const uint32 leaf\_begin = leaf.get\_index();

152                for (uint32 tri\_index = leaf\_begin; tri\_index < leaf\_end; ++tri\_index)

153                                  !!!

100 %

**Locals**

| Name       | Type    | Value                                             |
|------------|---------|---------------------------------------------------|
| leaf       | _local_ | {m_size = 67106176, m_index = 0}                  |
| leaf_index |         | 'leaf_index' has no value at the target location. |
| leaf_end   |         | 'leaf_end' has no value at the target location.   |
| leaf_begin |         | 'leaf_begin' has no value at the target location. |
| node       | _local_ | {m_packed_data = 2147484877, m_skip_node = 248}   |
| _T21609    | _local_ | {x = -1.4394605, y = -1.8220775, z = -2.150774}   |
| ray_inv    | _local_ | {x = -1.4394605, y = -1.8220775, z = -2.150774}   |
| node_index |         | 'node_index' has no value at the target location. |

**Call Stack**

| Name                                                       | Language |
|------------------------------------------------------------|----------|
| CUmodule 05508fe0 - [2] trace - Line 148                   | CUDA     |
| CUmodule 05508fe0 - [1] render_pixel - Line 409            | CUDA     |
| CUmodule 05508fe0 - [0] rt_trace_primary_kernel - Line 493 | CUDA     |

**Disassembly**

Address: 0x003e1298

148:                    const uint32 leaf\_index = node.get\_index( )

0x003e12a0 2800000010019de4 MOV R6, c[0x0][0x4];

0x003e12a8 28000000fc01dde4 MOV R7, R2;

0x003e12b0 2800000018019de4 MOV R6, R6;

0x003e12b8 4801000018411c03 IADD R4.CC, R4, R6;

0x003e12c0 480000001c515c43 IADD.X R5, R5, R7;

0x003e12c8 2800000010011de4 MOV R4, R4;

0x003e12d0 2800000014015de4 MOV R5, R5;

0x003e12d8 2800000014015de4 MOV R5, R5;

!!!

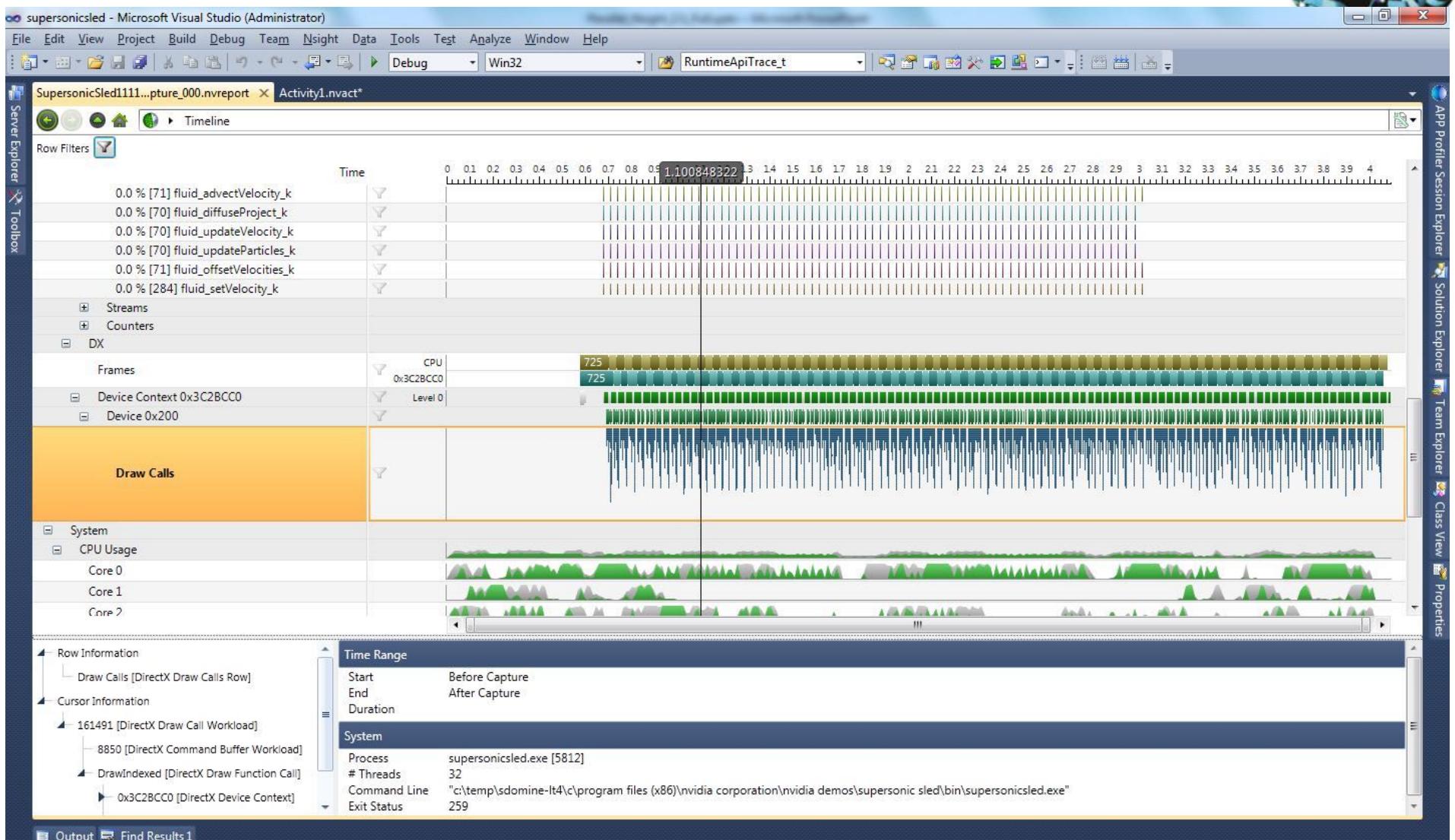
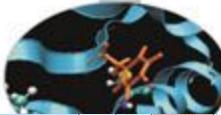
**CUDA WarpWatch 1**

| Name | ray_inv.x  | ray_inv.y   | ray_inv.z |
|------|------------|-------------|-----------|
| 0    | -1.444908  | -1.7955524  | -2.17     |
| 1    | -1.44425   | -1.7967783  | -2.17     |
| 2    | -1.4440092 | -1.7980076  | -2.17     |
| 3    | -1.4437686 | -1.7992405  | -2.17     |
| 4    | -1.4435281 | -1.800477   | -2.17     |
| 5    | -1.4432876 | -1.8017174  | -2.17     |
| 6    | -1.4430474 | -1.8029615  | -2.17     |
| 7    | -1.4428074 | -1.8042094  | -2.16     |
| 8    | -1.4425675 | -1.8054608  | -2.16     |
| 9    | -1.4423276 | -1.8067161  | -2.16     |
| 10   | -1.4420878 | -1.8079749  | -2.16     |
| 11   | -1.4418485 | -1.8092378  | -2.16     |
| 12   | -1.4416089 | -1.8105046  | -2.16     |
| 13   | -1.4413697 | -1.8117749  | -2.16     |
| 14   | -1.4411306 | -1.8130492  | -2.16     |
| 15   | -1.4408917 | -1.8143274  | -2.15     |
| 16   | -1.4406527 | -1.8156093  | -2.15     |
| 17   | -1.4404141 | -1.8168953  | -2.15     |
| 18   | -1.4401754 | -1.818185   | -2.15     |
| 19   | -1.439937  | -1.8194786  | -2.15     |
| 20   | -1.4396986 | -1.820776   | -2.15     |
| 21   | -1.4394605 | -1.8220775  | -2.15     |
| 22   | -1.4392225 | -1.8233831  | -2.14     |
| 23   | -1.4389844 | -1.8246926  | -2.14     |
| 24   | -1.4387469 | -1.8260059  | -2.14     |
| 25   | -1.4385092 | -1.8273233  | -2.14     |
| 26   | -1.4382718 | -1.828645   | -2.14     |
| 27   | -1.4380344 | -1.8299706  | -2.14     |
| 28   | -1.4377974 | -1.8313001  | -2.14     |
| 29   | -1.4375603 | -1.832634   | -2.13     |
| 30   | -1.4373236 | -1.8339716  | -2.13     |
| 31   | -1.4370868 | -1.83553136 | -2.13     |

CUDA WarpWatch 1 Output

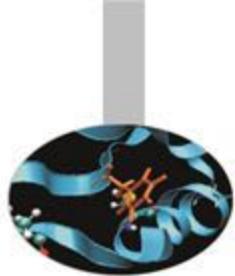


# Parallel NSight



# Other CUDA command line programs

- **nvidia-smi**
  - Shows which GPUs are available and gives information about them
  - Can be used in scrolling mode when running CUDA programs
- **nvprof**
  - Quick profiler, useful for showing memory transfers between host and device.
  - More sophisticated profiling can be done with nvp.
- **cuda-memcheck**
  - Ideal for spotting memory leaks in the CUDA program. Will considerably slow execution.
- **cuda-gdb**
  - CUDA debugger



# Debugging: CUDA-MEMCHECK

- It's able to detect buffer overflows, misaligned global memory accesses and leaks
- Device-side allocations are supported
- Standalone or fully integrated in CUDA-GDB

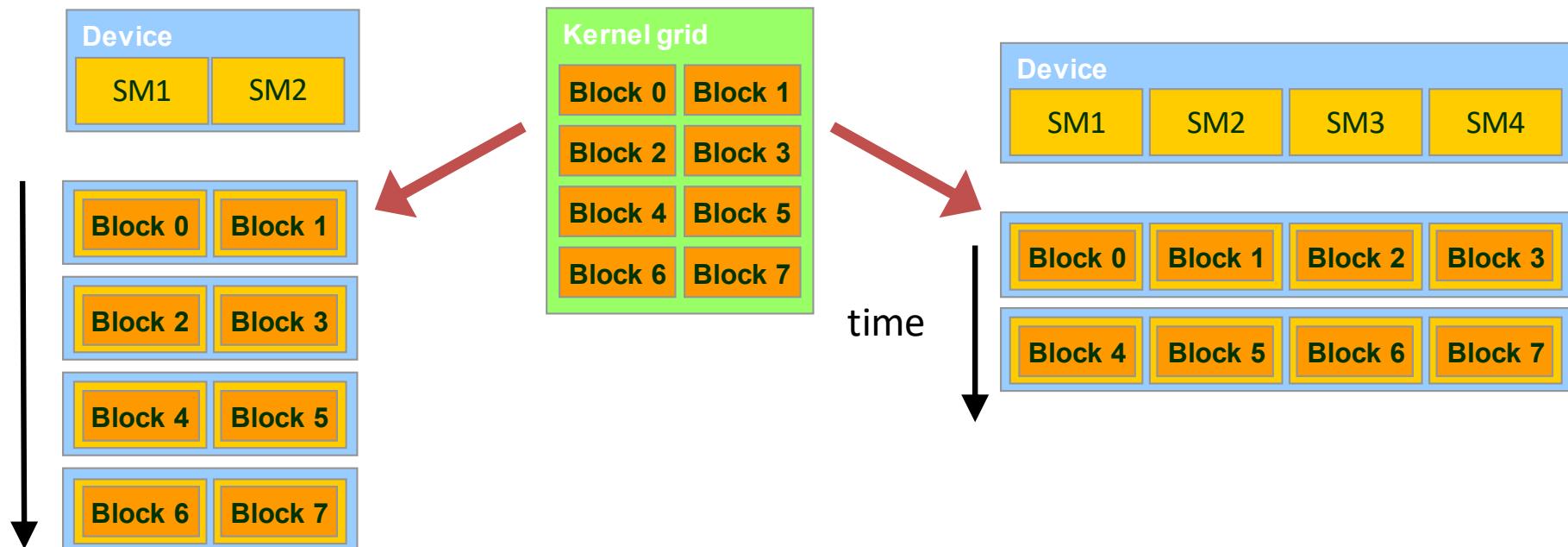
```
$ cuda-memcheck --continue ./memcheck_demo
=====
 CUDA-MEMCHECK
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
=====
 Invalid __global__ write of size 4
 at 0x00000038 in memcheck_demo.cu:5:unaligned_kernel
 by thread (0,0,0) in block (0,0,0)
 Address 0x200200001 is misaligned
=====
 Invalid __global__ write of size 4
 at 0x00000030 in memcheck_demo.cu:10:out_of_bounds_kernel
 by thread (0,0,0) in block (0,0,0)
 Address 0x87654320 is out of bounds
=====
=====
=====
 ERROR SUMMARY: 2 errors
```

Some more details

# Transparent Scalability

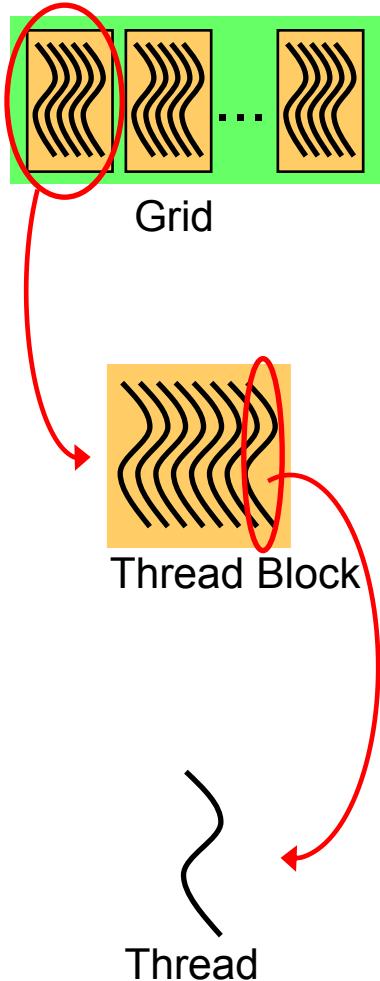
The GPU runtime system can execute blocks in any order relative to each other

This flexibility enables to execute the same application code on hardware with different numbers of SMs.

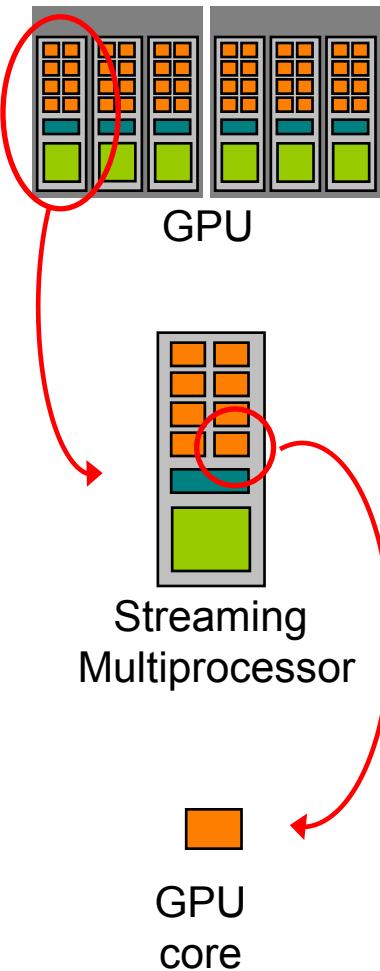


# more on the GPU Execution Model

## Software



## Hardware



when a GPU kernel is invoked:

- each thread block is assigned to a SM in a round-robin mode
  - a maximum number of blocks can be assigned to each SM, depending on hardware generation and on how many resources each requires (registers, shared memory, etc)
  - the runtime system maintains a list of active blocks and assigns new blocks to SMs as they complete
  - once a block is assigned to a SM, it remains on that SM until the work for all threads in the block is completed
  - each block execution is independent from the other (no synchronization is possible among them)
- threads of each block are partitioned into warps of consecutive *threads*
- the scheduler selects for execution a warp from one of the residing blocks in each SM
- A warp executes one common set of instructions at a time
  - each GPU core takes care of one thread in the warp
  - full efficiency when all threads agree on their execution path

# CUDA and NVIDIA GPUs

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

How many threads and blocks can I use?

Depends on the *compute capability* of the device which describes the GPU features available.

| Code name      | Product name | Compute capability | SMM units | Max threads/block | Max thread blocks/sm | #cores (FP32) |
|----------------|--------------|--------------------|-----------|-------------------|----------------------|---------------|
| Kepler (GK210) | Tesla K40    | 3.7                | 15        | 1024              | 16                   | 2496          |
| Maxwell        | Tesla M40    | 5.2                | 24        | 1024              | 32                   | 3072          |
| Pascal         | Tesla P100   | 6.0                | 56        | 1024              | 32                   | 3584          |
| Volta          | Tesla V100   | 7.0                | 80        | 1024              | 32                   | 5120          |

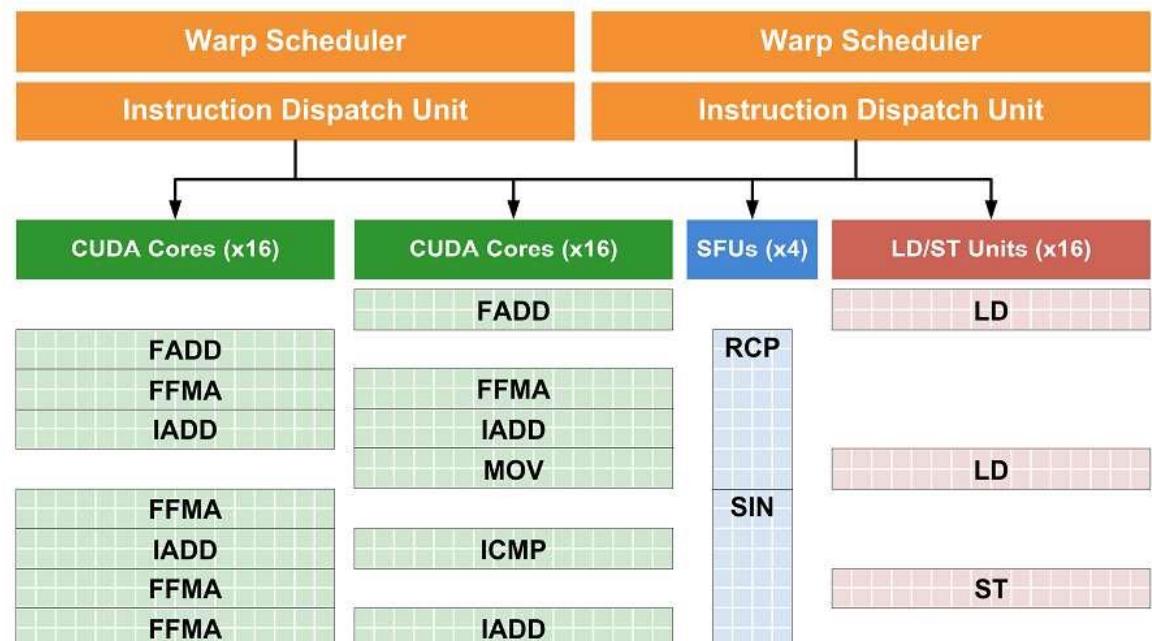
But often makes sense to set `threads/block=1024` and make the number of blocks = `problem_size/1024`

|                                                                                     | Compute Capability      |      |     |        |       |       |            |                         |       |       |             |             |
|-------------------------------------------------------------------------------------|-------------------------|------|-----|--------|-------|-------|------------|-------------------------|-------|-------|-------------|-------------|
| Technical Specifications                                                            | 3.0                     | 3.2  | 3.5 | 3.7    | 5.0   | 5.2   | 5.3        | 6.0                     | 6.1   | 6.2   | 7.0         | 7.5         |
| Maximum number of resident grids per device<br><i>(Concurrent Kernel Execution)</i> | 16                      | 4    |     |        | 32    |       |            | 16                      | 128   | 32    | 16          | 128         |
| Maximum dimensionality of grid of thread blocks                                     |                         |      |     |        |       |       | 3          |                         |       |       |             |             |
| Maximum x-dimension of a grid of thread blocks                                      |                         |      |     |        |       |       | $2^{31}-1$ |                         |       |       |             |             |
| Maximum y- or z-dimension of a grid of thread blocks                                |                         |      |     |        |       |       | 65535      |                         |       |       |             |             |
| Maximum dimensionality of thread block                                              |                         |      |     |        |       |       | 3          |                         |       |       |             |             |
| Maximum x- or y-dimension of a block                                                |                         |      |     |        |       |       | 1024       |                         |       |       |             |             |
| Maximum z-dimension of a block                                                      |                         |      |     |        |       |       | 64         |                         |       |       |             |             |
| Maximum number of threads per block                                                 |                         |      |     |        |       |       | 1024       |                         |       |       |             |             |
| Warp size                                                                           |                         |      |     |        |       |       | 32         |                         |       |       |             |             |
| Maximum number of resident blocks per multiprocessor                                | 16                      |      |     |        |       |       |            | 32                      |       |       |             | 16          |
| Maximum number of resident warps per multiprocessor                                 |                         |      |     |        |       |       | 64         |                         |       |       |             | 32          |
| Maximum number of resident threads per multiprocessor                               |                         |      |     |        |       |       | 2048       |                         |       |       |             | 1024        |
| Number of 32-bit registers per multiprocessor                                       | 64 K                    |      |     | 128 K  |       |       |            | 64 K                    |       |       |             |             |
| Maximum number of 32-bit registers per thread block                                 | 64 K                    | 32 K |     | 64 K   |       | 32 K  |            | 64 K                    | 32 K  |       | 64 K        |             |
| Maximum number of 32-bit registers per thread                                       | 63                      |      |     |        |       |       | 255        |                         |       |       |             |             |
| Maximum amount of shared memory per multiprocessor                                  | 48 KB                   |      |     | 112 KB | 64 KB | 96 KB | 64 KB      | 96 KB                   | 64 KB | 96 KB | 64 KB       |             |
| Maximum amount of shared memory per thread block <sup>27</sup>                      |                         |      |     | 48 KB  |       |       |            |                         |       | 96 KB | 64 KB       |             |
| Number of shared memory banks                                                       |                         |      |     |        |       |       | 32         |                         |       |       |             |             |
| Amount of local memory per thread                                                   |                         |      |     |        |       |       | 512 KB     |                         |       |       |             |             |
| Constant memory size                                                                |                         |      |     |        |       |       | 64 KB      |                         |       |       |             |             |
| Cache working set per multiprocessor for constant memory                            | 8 KB                    |      |     |        |       |       | 4 KB       | 8 KB                    |       |       |             |             |
| Cache working set per multiprocessor for texture memory                             | Between 12 KB and 48 KB |      |     |        |       |       |            | Between 24 KB and 48 KB |       |       | 32 ~ 128 KB | 32 or 64 KB |

# Warps

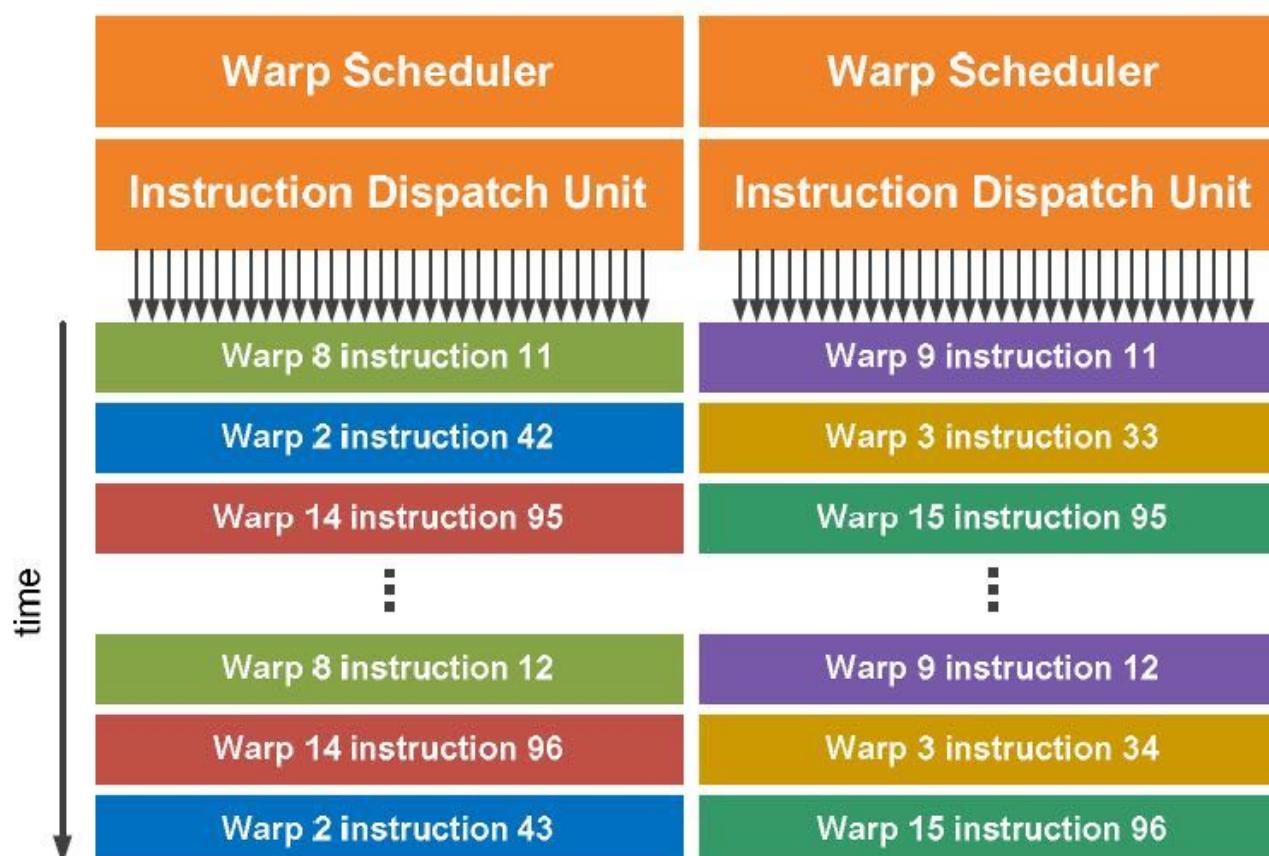
- The GPU multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.
- Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently

- each *warp* can execute instructions on
  - SM cores
  - load/store units
  - SFUs units



# The SM warp scheduler

- The NVIDIA SM schedules threads in groups of 32 threads, called *warps*
- Using 2 warp schedulers per SM allows two warps to be issued and executed concurrently if hardware resources are available



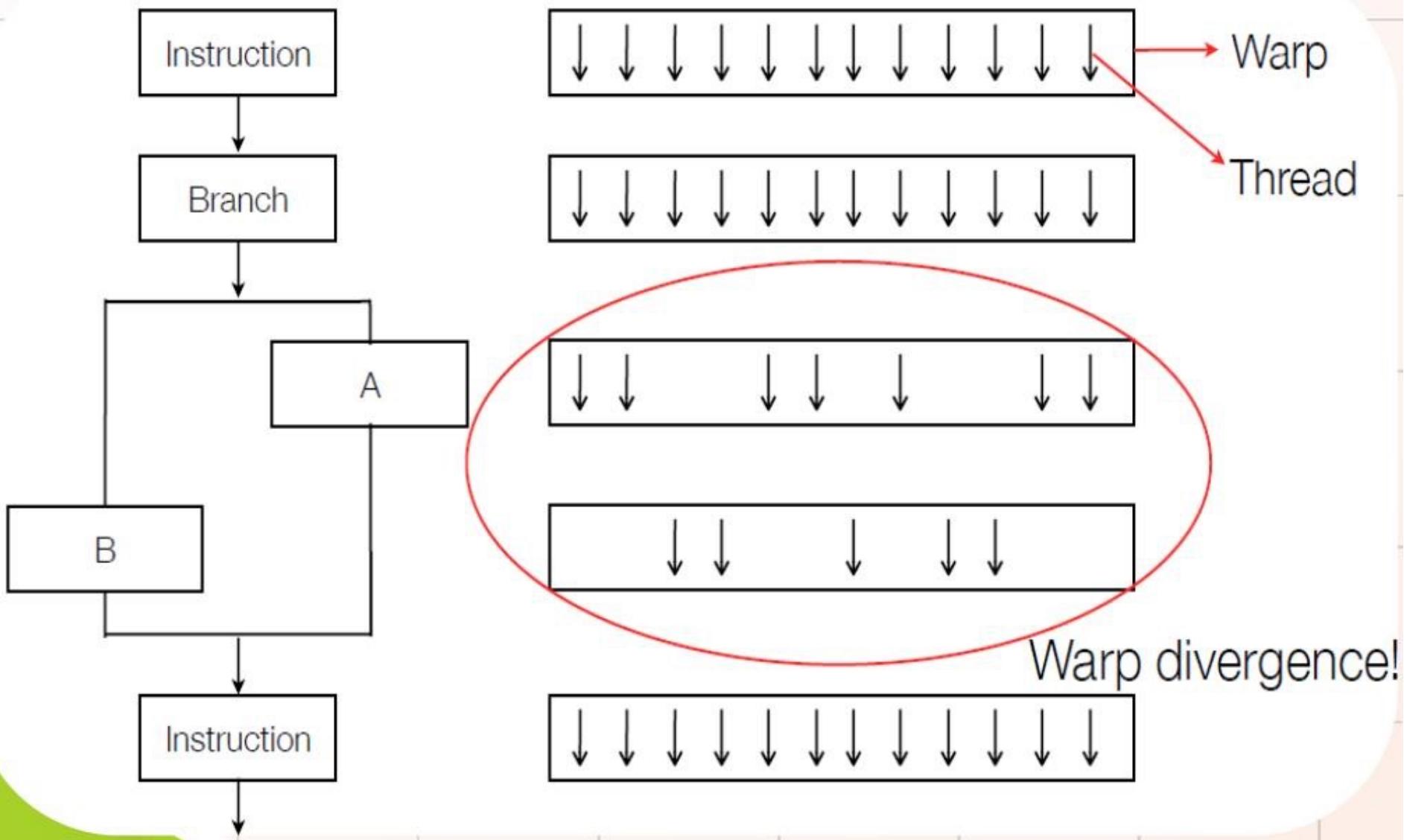
# Warps

- A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path.
- If threads of a warp diverge via a data-dependent conditional branch, **the warp serially executes each branch path taken**, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.
- **Branch divergence occurs only within a warp**; different warps execute independently regardless of whether they are executing common or disjointed code paths.
- Each single instruction **in a warp** is performed in a lockstep. The next instruction can be fetched only when the previous one has completed.
- An SM statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps (half and quarter-warp) that is ready to execute, if any.
- Volta is equipped with 4 warp-scheduler units. Instructions are performed over two cycles, and the schedulers can issue independent instructions every cycle. Dependent instruction issue latency for core FMA math operations are reduced to four clock cycles, so **execution latencies of core math operations can be hidden** by as few as 4 warps per SM, assuming 4-way instruction-level parallelism *ILP* per warp. Many more warps are, of course, recommended to cover the much greater latency of memory transactions and control-flow operations.

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture>

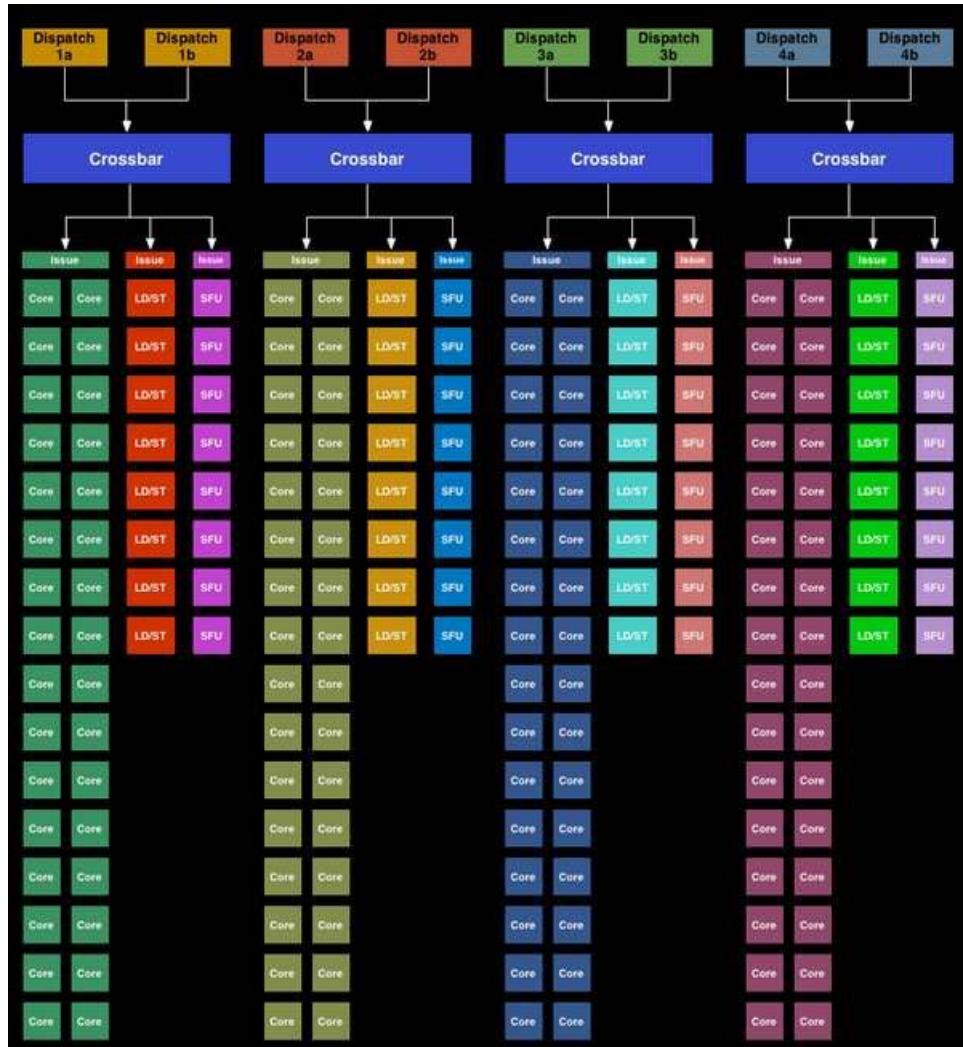
MANY DETAILS HERE <http://taylorlloyd.ca/gpu,/pascal,/cuda/2017/01/07/gpu-pipelines.html>

# Flow Divergence



# Volta SM Warp Scheduler

- Volta SM has 4 warp scheduler
- Each scheduler is responsible for
  - feeding 32 CUDA cores
  - 8 load/store units
  - 8 Special Function Units
- There are two dispatch ports per warp schedule
  - a warp scheduler can use little instruction level parallelism (ILP) by issuing a second instruction to an unused resource



# Instruction Execution

## Example

- a single Volta processing block has 16 FP32/INT32 and 8 FP64 ALU units
- a CUDA warps is 32 threads wide
- a FP32 operation on a warp will execute in 32 threads / 16 FP32 ALU = 2 cycles
- a FP64 operation on a warp will execute in 32 threads / 8 FP64 ALU = 4 cycles
- Each arithmetic operation has a pipe line stage so that, as soon as one warp has entered the first stage, a second independent warp can push its operand into the pipeline.
- FMA operations have four clock cycles on Volta: execution latencies of FMA core math operations can be hidden by as few as 4 warps per SM, assuming 4-way instruction-level parallelism ILP per warp

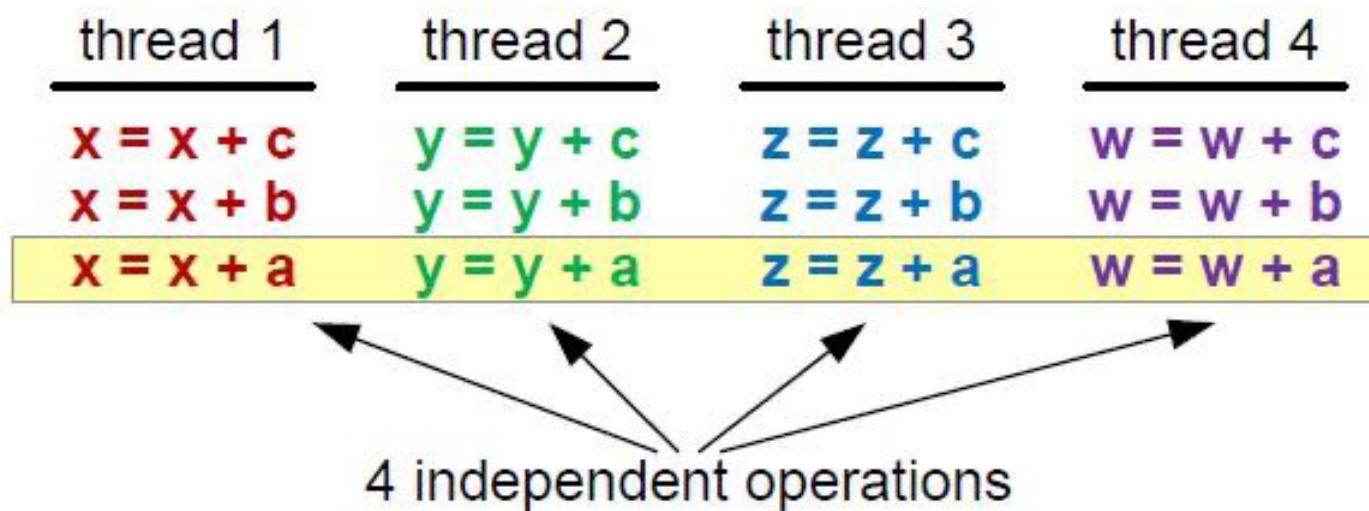


# Hiding Latencies

- What is latency?
  - the number of clock cycles needed to complete an instruction
  - ... that is, the number of cycles I need to wait for before another **dependent operation** can start
    - arithmetic latency (~ 18-24 cycles)
    - memory access latency (~ 400-800 cycles)
- We cannot discard latencies (it's an hardware design effect), but we can lessen their effect and hide them.
  - saturating computational pipelines in computational bound problems
  - saturating bandwidth in memory bound problems
- We can organize our code so to provide the scheduler a sufficient number of **independent operations**, so that the more the warp are available, the more content-switch can hide latencies and proceed with other useful operations
- There are two possible ways and paradigms to use (can be combined too!)
  - Thread-Level Parallelism (TLP)
  - Instruction-Level Parallelism (ILP)

# Thread-Level Parallelism (TLP)

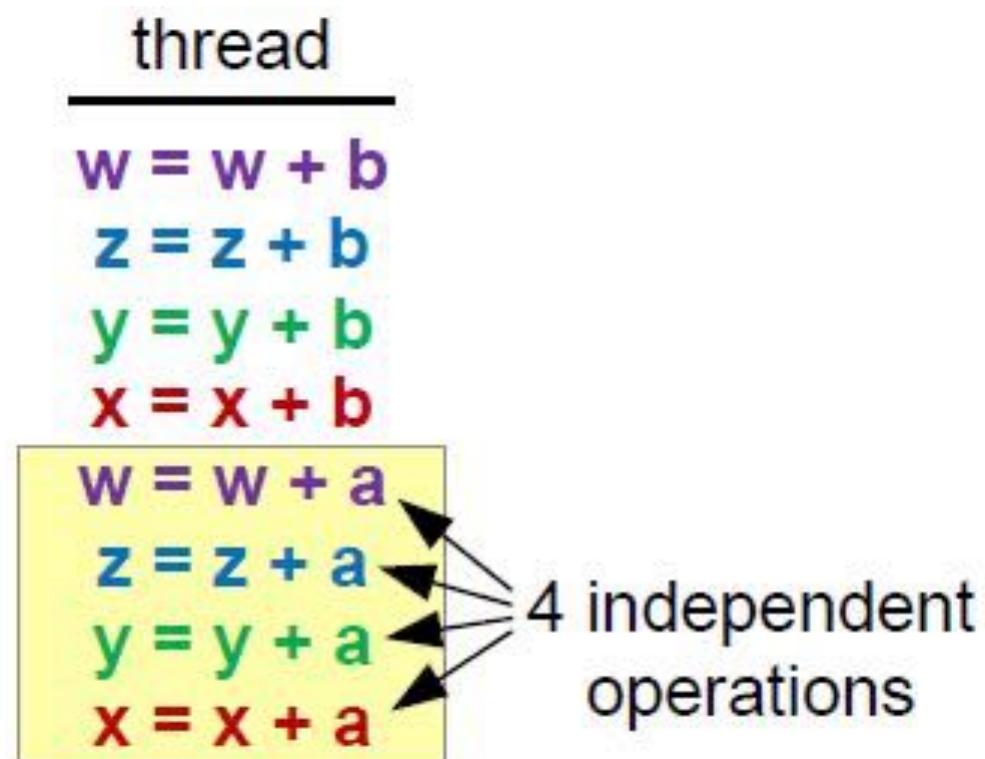
- Strive for high SM **occupancy**: that is try to provide as much threads per SM as possible, so to easy the scheduler find a warp ready to execute, while the others are still busy
- This kind of approach is effective when there is a low level of independent operations per CUDA kernels



# Instruction-Level Parallelism (ILP)

- Strive for multiple independent operations inside your CUDA kernel: that is, let your kernel act on more than one data
- this will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- note: the scheduler will not select a new warp until there are eligible instructions ready to execute on the current warp



# Branching example

- E.g you want to split your threads into 2 groups:

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if (i%2 == 0)
 ...
else
 ...
```



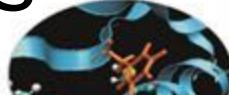
Threads within warp diverge

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if ((i/32)%2 == 0)
 ...
else
 ...
```



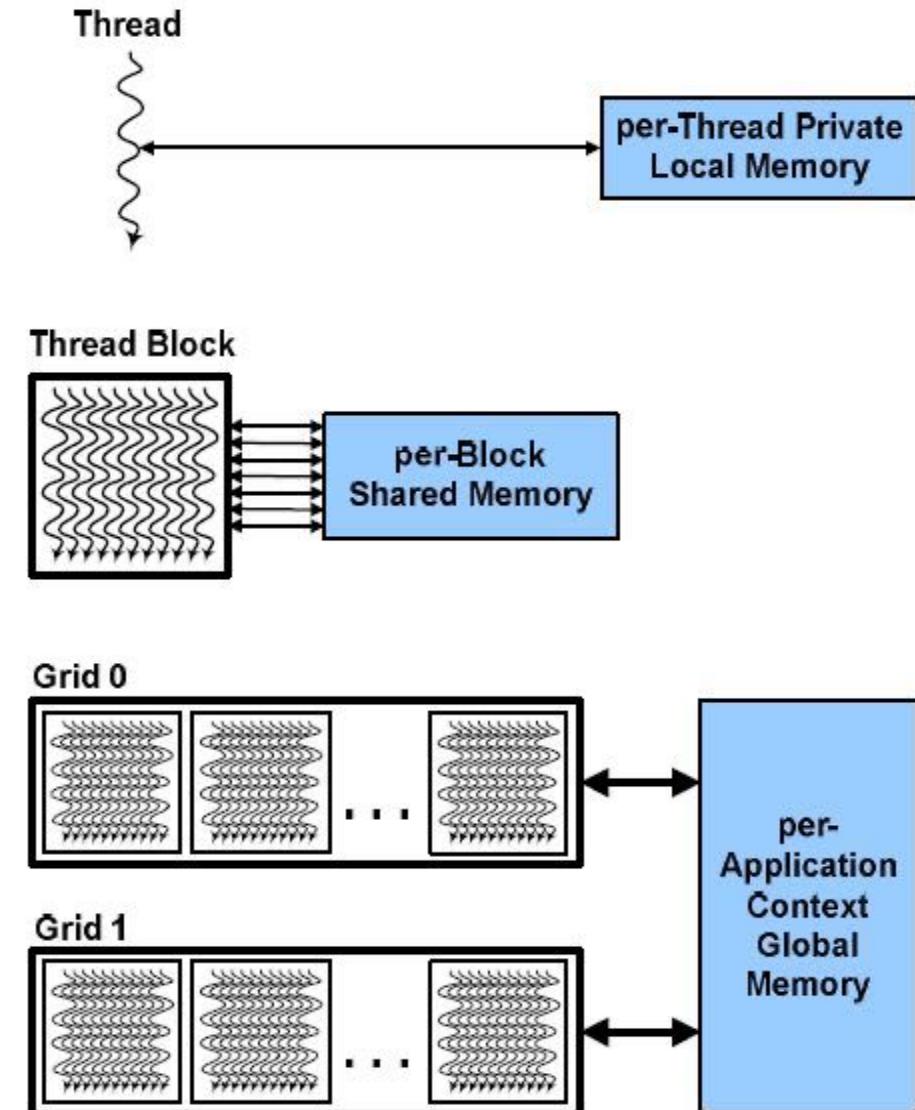
Threads within warp follow same path

# Hierarchy of device memories

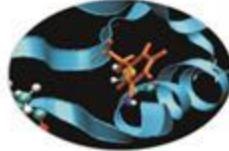


CUDA's hierarchy of threads maps to a hierarchy of memories on the GPU:

- Each thread has some **registers**, used to hold automatic scalar variables declared in kernel and device functions, and a **per-thread private memory space** used for register spills, function calls, and C automatic array variables
- Each thread block has a **per-block shared memory space** used for inter-thread communication, data sharing, and result sharing in parallel algorithms
- Grids of thread blocks share results in **global memory space**



# CUDA device memory model

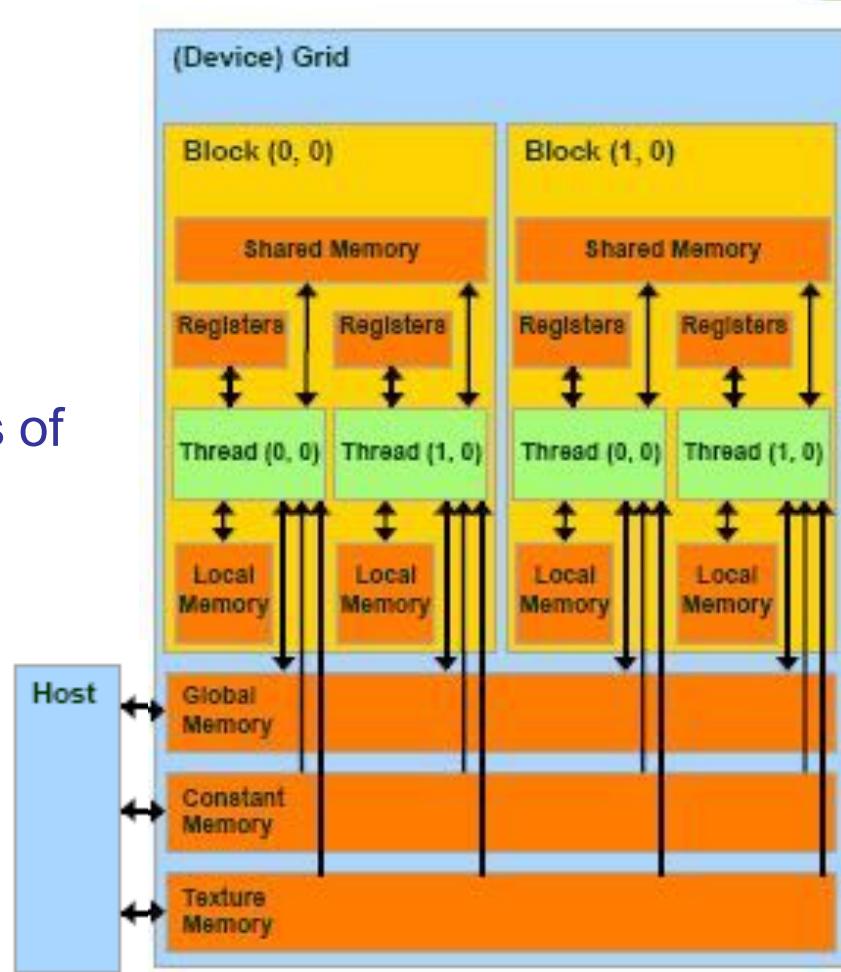


## on-chip memories:

- *registers (~8KB)* → SP
- *shared memory (~16KB)* → SM
- they can be accessed at very high speed in a highly parallel manner.

## per-grid memories:

- *global memory (~4GB)*
  - long access latencies (hundreds of clock cycles)
  - finite access bandwidth
- *constant memory (~64KB)*
  - read only
  - short-latency (cached) and high bandwidth when all threads simultaneously access the same location
- *texture memory (read only)*
- CPU can transfer data to/from all per-grid memories.

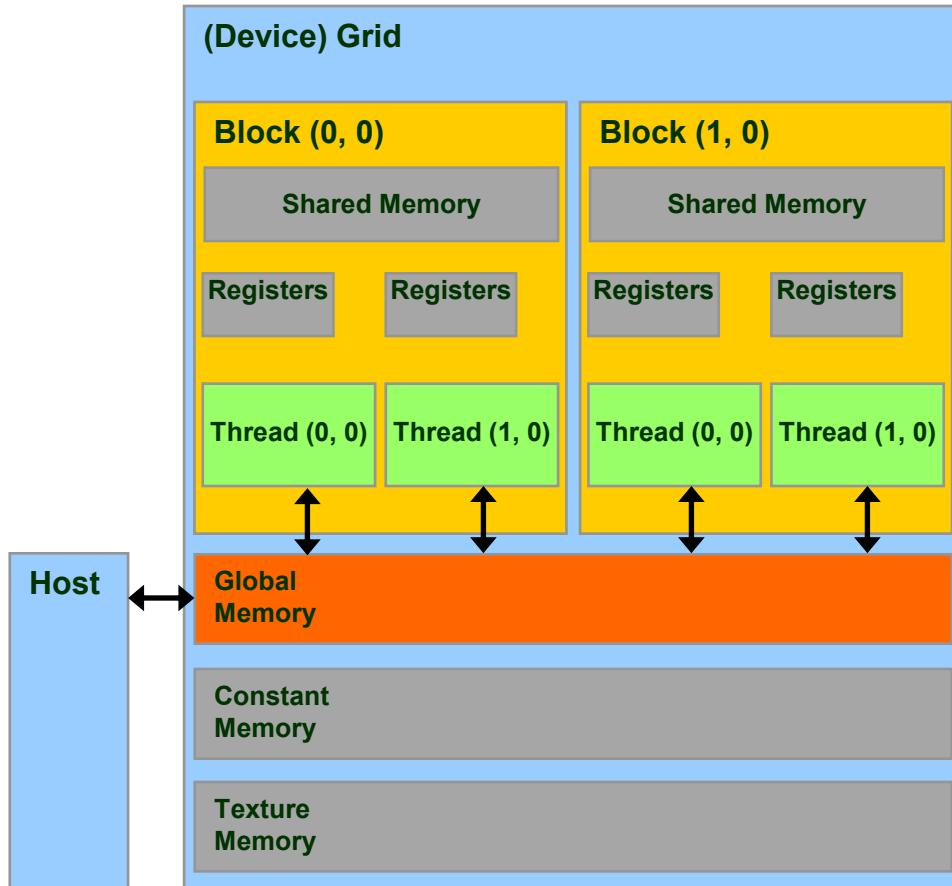


*Local memory* is implemented as part of the global memory, therefore has a long access latencies too.

# Global Memory

- **Global Memory** is the larger memory available on a *device*

- Comparable to a RAM for CPU
- Its status is maintained among different kernel launches
- Can be access both read/write from all threads of the kernel grid
- Unique memory that can be used in read/write access from the CPU
- **Very high bandwidth**  
Throughput > 900 GB/s
- **Very high latency**  
about 400-800 clock cycles



- Global memory bandwidth for graphics memory on GPU is high compared to CPU
  - But there are many data-hungry cores
  - Memory bandwidth is a bottleneck
- Maximum bandwidth achieved when data is loaded for multiple threads in a single transaction: coalescing
- This will happen when data access patterns meet certain conditions: 16 consecutive threads (half-warp) must access data from within the same memory segment
- E.g. condition met when consecutive threads read consecutive memory addresses within a warp.
- Otherwise, memory accesses are serialised, significantly degrading performance
- Adapting code to allow coalescing can dramatically improve performance

# Global Memory Load/Store

```
// strided data copy
__global__ void strideCopy (int N, float *odata, float* idata, int stride) {
 int xid = (blockIdx.x*blockDim.x + threadIdx.x) * stride;
 if (xid < N) odata[xid] = idata[xid];
}
```

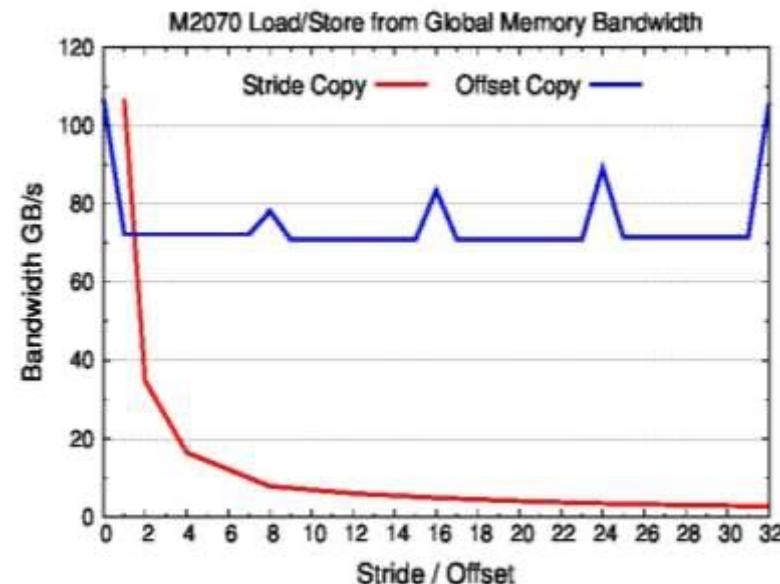
```
// offset data copy
__global__ void offsetCopy(int N, float *odata, float* idata, int offset) {
 int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
 if (idx < N) odata[xid] = idata[xid];
}
```

Strided based copy

| Stride | Bandwidth GB/s |
|--------|----------------|
| 1      | 106.6          |
| 2      | 34.8           |
| 8      | 7.9            |
| 16     | 4.9            |
| 32     | 2.7            |

Offset based copy

| Offset | Bandwidth GB/s |
|--------|----------------|
| 0      | 106.6          |
| 1      | 72.2           |
| 8      | 78.2           |
| 16     | 83.4           |
| 32     | 105.7          |



Measured on a M2070; Total elements = 16776960; Used Blocks = 65535; Block lenght = 256

# Data alignment in *Global Memory*

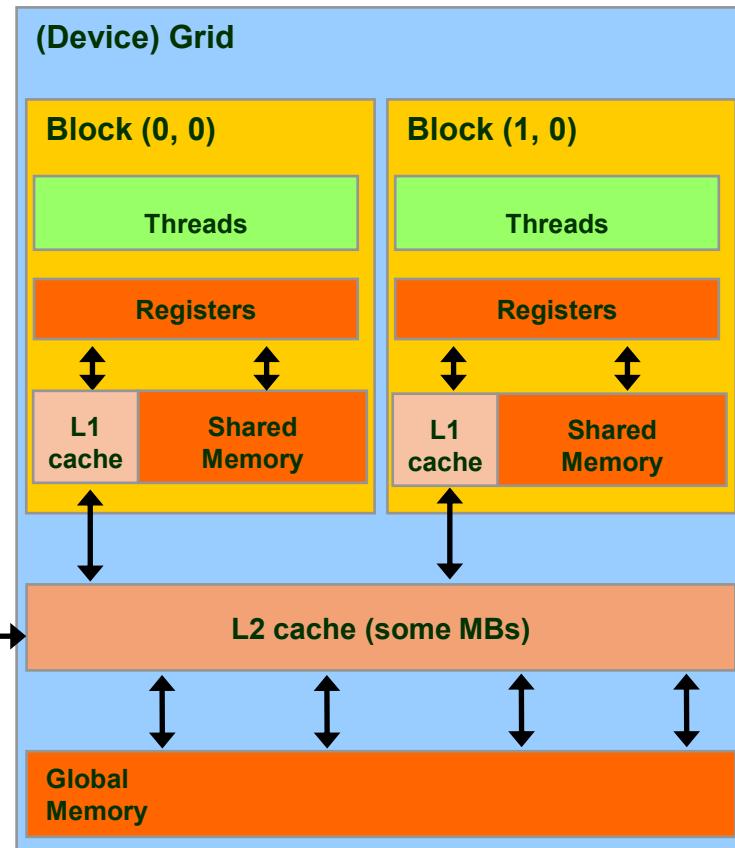
- It is very important to align data in memory so to have aligned accesses (*coalesced*) during load/store operation in global memory, reducing the number of segments moved across the bus
  - **cudaMalloc()** grants the alignment of first element in global memory, useful for one dimensional arrays
  - **cudaMallocPitch()** must be used to allocate 2d buffers
    - elements are padded so each row is aligned for coalescing accesses
    - returns an integer (pitch) which can be used as a stride to access row elements

```
// host code
int width = 64, height = 64; int pitch; float *devPtr;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);

// device code
__global__ myKernel(float *devPtr, int pitch, int width, int height)
{
 for (int r = 0; r < height; r++) {
 float *row = devPtr + r * pitch;
 for (int c = 0; c < width; c++)
 float element = row[c];
 }
 ...
}
```

# Cache Hierarchy for Global Memory Accesses

- GPU designs include cache hierarchy in order to ease the need for space and time data locality
- 2 Levels of cache:
  - **L2** : shared among all SM
    - Kepler 1MB, Pascal 4MB, Volta 6MB
    - 25% less latency than Global Memory
  - **L1** : private to each SM
    - [16/48 KB] configurable
    - L1 + Shared Memory = 64 KB
    - Kepler : configurable also as 32 KB



```
cudaFuncSetCacheConfig(kernel1, cudaFuncCachePreferL1); // 48KB L1 / 16KB ShMem
cudaFuncSetCacheConfig(kernel2, cudaFuncCachePreferShared); // 16KB L1 / 48KB ShMem
```

# Set cache configuration

`cudaDeviceSetCacheConfig ( cudaFuncCache cacheConfig )`

## Description:

- On devices where the L1 cache and shared memory use the same hardware resources, this sets through cacheConfig the preferred cache configuration for the current device. This is only a preference. **The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function.**
- Any function preference set via [cudaFuncSetCacheConfig \(\)](#) will be preferred over this device-wide setting. Launching a kernel with a different preference than the most recent preference setting, may insert a device-side synchronization point.
- The supported cache configurations are:
  - [cudaFuncCachePreferNone](#): no preference for shared memory or L1 (default)
  - [cudaFuncCachePreferShared](#): prefer larger shared memory and smaller L1 cache
  - [cudaFuncCachePreferL1](#): prefer larger L1 cache and smaller shared memory
  - [cudaFuncCachePreferEqual](#): prefer equal size L1 cache and shared memory

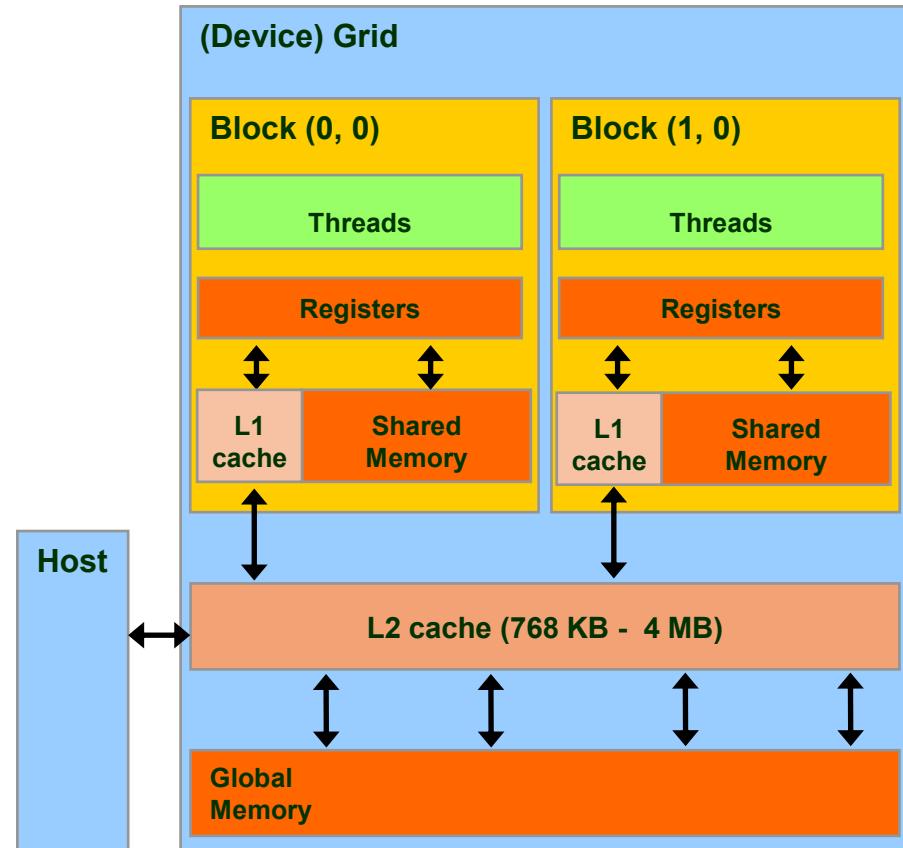
# Cache Hierarchy for Global Memory Accesses

Just one type of **store** operation:

- when data should be updated in global memory, its L1 copy is invalidated and updated the L2 cache value

Two different type of **load** operations:

- Caching (default mode)**
  - when data is requested by some thread, data is first searched in L1 cache, then in L2 cache, last in global memory
  - cache line lenght is **128-byte**
- Non-caching (compile time selected)**
  - the L1 cache is disabled
  - when data is requested by some thread, data is first searched in L2 cache, then in global memory
  - cache line lenght is **32-bytes**
  - this mode is activated at *compile time* using the compiler option:  
-Xptxas -dlcm=cg

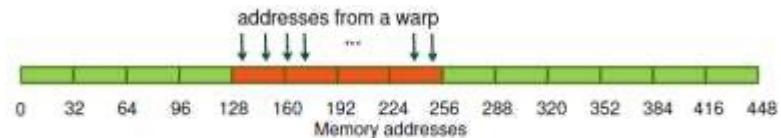
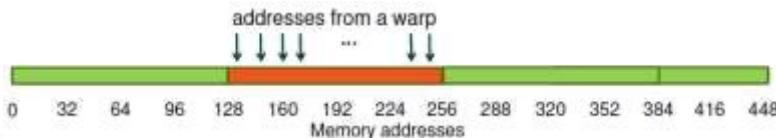


# Load Operations from Global Memory

- All load/store requests in global memory are issued per *warp* (as all other instructions)
  1. each *thread* in a *warp* compute the address to access
  2. *load/store* units select segments where data resides
  3. *load/store* start transfer of needed segments

**Warp requires 32 consecutive 4-byte word aligned to segment (total 128 bytes)**

| Caching Load                                 | Non-caching Load                              |
|----------------------------------------------|-----------------------------------------------|
| all addresses belong to 1 line cache segment | all addresses belong to 4 line cache segments |
| 128 bytes are moved over the bus             | 128 bytes are moved over the bus              |
| bus utilization: <b>100%</b>                 | bus utilization: <b>100%</b>                  |



# Load Operations from Global Memory

Warp requests 32 permuted 4-byte words aligned to segment (total 128 bytes)

## Caching Load

addresses belong to 1 line cache segments

128 bytes are moved over the bus

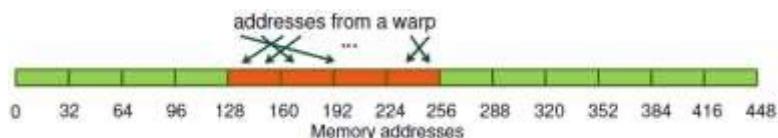
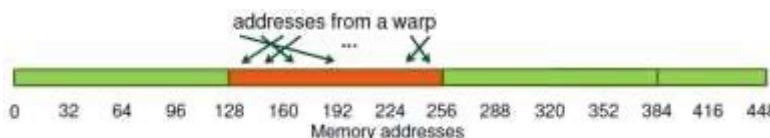
bus utilization: **100%**

## Non-caching Load

addresses belong to 4 line cache segments

128 bytes are moved over the bus

bus utilization: **100%**



Warp requires 32 consecutive 4-bytes words not aligned to segment (total 128 bytes)

## Caching Load

addresses belong to 2 line cache segments

256 bytes are moved over the bus

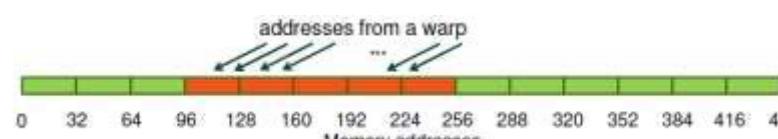
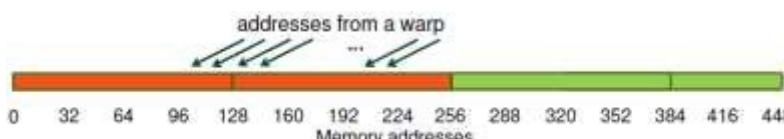
bus utilization: **50%**

## Non-caching Load

addresses belong to 5 line cache segments

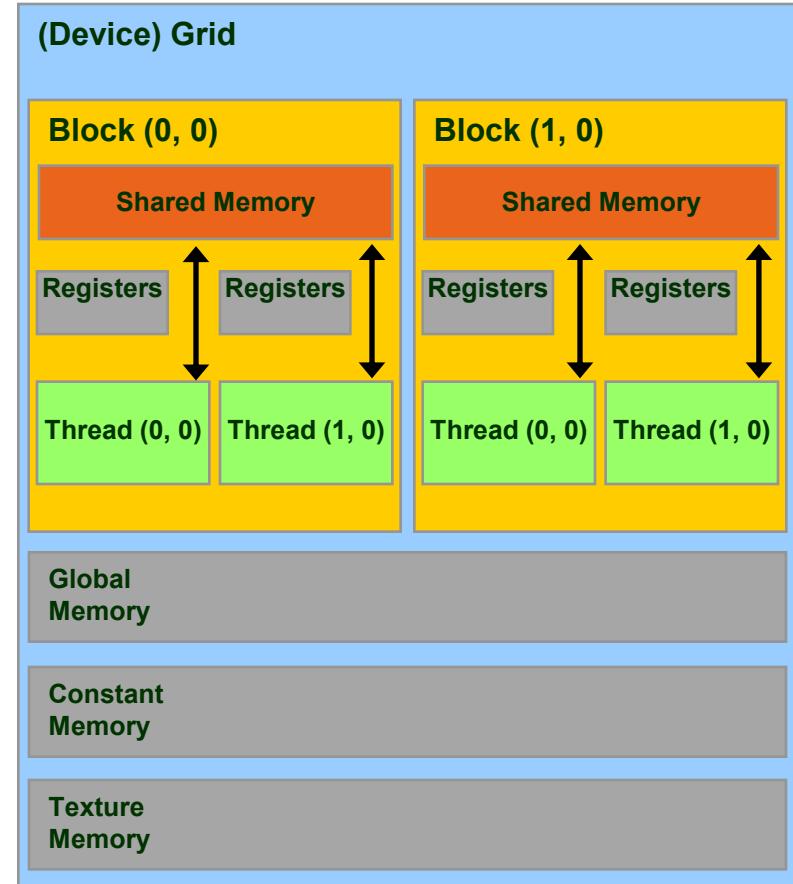
160 bytes are moved over the bus

bus utilization: **80%**



# *Shared Memory*

- The ***Shared Memory*** is a small, but quite fast memory mounted on each SM
  - read/write access for threads of blocks residing on the same SM
  - a cache memory under the direct control of the programmer
  - its status is not maintained among different kernel calls
- Specifications:
  - **Very low latency**: 2 clock cycles
  - Throughput: 32 bit every 2 cycles
  - Dimension : **48 KB [default]**  
(Configurable : 16/32/48 KB)



# Shared Memory Allocation

```
// statically inside the kernel
__global__ myKernelOnGPU (...) {
 ...
 __shared__ type shmem[MEMSZ];
 ...
}

or using dynamic allocation

// dynamically sized
extern __shared__ type *dynshmem;

__global__ myKernelOnGPU (...) {
 ...
 dynshmem[i] = ... ;
 ...
}

void myHostFunction() {
 ...
 myKernelOnGPU<<<gs,bs,MEMSZ>>>();
}
```

```
! statically inside the kernel
attribute(global)
subroutine myKernel(...)
 ...
 type, __shared__: variable_name
 ...
end subroutine

oppure

! dynamically sized
type, __shared__: dynshmem(*)

attribute(global)
subroutine myKernel(...)
 ...
 dynshmem(i) = ...
 ...
end subroutine
```

- variables allocated in shared memory has storage duration of the kernel launch (not persistent!)
- only accessible by threads of the same block

# Thread Block Synchronization

- All threads in the same block can be synchronized using the CUDA runtime API call:

`__syncthreads()` | `call syncthreads()`

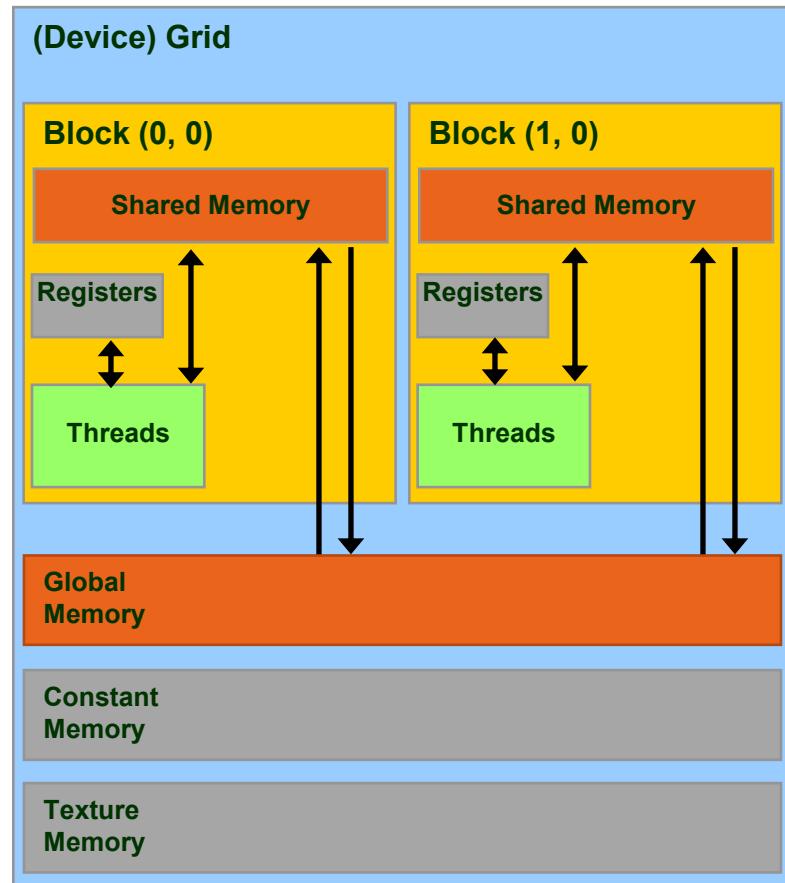
which blocks execution until all other threads reach the same call location

- can be used in conditional too, but only if all thread in the block reach the same synchronization call

*“... otherwise the code execution is likely to hang or produce unintended side effects”*

# Using Shared Memory for Thread Cooperation

- Threads belonging to the same block can cooperate together using the shared memory to share data
  - if a thread is in need of some data which has been already retrieved by another thread in the same block, this data can be shared using the shared memory
- typical Shared Memory usage pattern:
  - declare a buffer residing on shared memory (this buffer is per block)
  - load data into shared memory buffer
  - synchronize threads so to make sure all needed data is present in the buffer
  - perform operation on data
  - synchronize threads so all operations have been performed
  - write back results to global memory



# Kernel Synchronization

```
__global__ void vector_sum(int *in, int *out) {
 __shared__ int temp[BLOCK_SIZE+2*RADIUS];
 int gindex=threadIdx.x+blockIdx.x*blockDim.x; // global index
 int lindex=threadIdx.x+RADIUS; // local index

 // Read input elements into shared memory
 temp[lindex]=in[gindex];
 if (threadIdx.x<RADIUS) { // some extra work
 temp[lindex-RADIUS]=in[gindex-RADIUS];
 temp[lindex+BLOCK_SIZE]=in[gindex+BLOCK_SIZE];
 }
 __syncthreads();
 int offset, result=0;
 for (offset=-RADIUS; offset<=RADIUS; offset++)
 result+=temp[lindex+offset];
 out[gindex]=result;
}
```

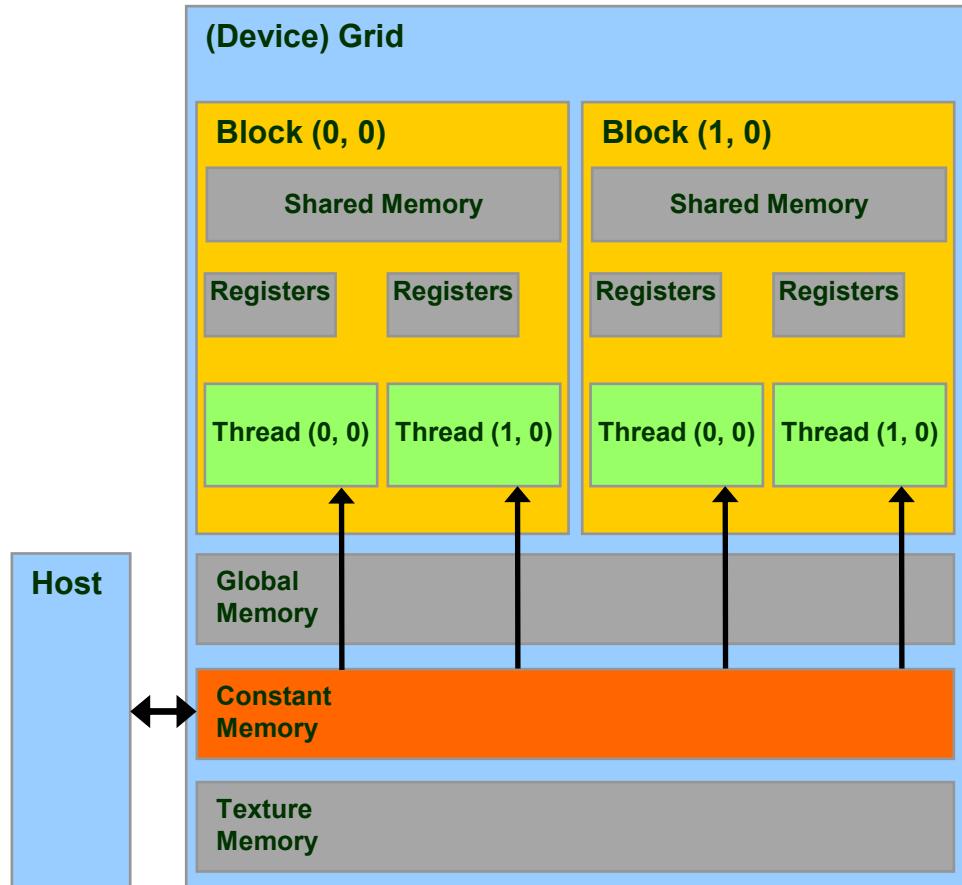
# Constant Memory

- **Constant Memory** is the ideal place to store constant data in **read-only** access from all threads

- constant memory data actually reside in the global memory, but fetched data is moved into a dedicated *constant-cache*
- very effective when all *thread* of a *warp* request the same memory address
- its values are initialized from host code using a special CUDA API

- Specifications:

- Dimension : **64 KB**
- Throughput: 32 bits per warp every 2 clock cycles



# Accessing Constant Memory

Suppose a kernel is launched using 320 warps per SM and all threads requests the same data

- if data is on global memory:
  - all *warp* will request the same segment from global memory
  - the first time segment is copied into L2 cache
  - if other data pass through L2, there are good chances it will be lost
  - there are good chances that data should be requested 320 times
- if data is in constant memory:
  - during first *warp* request, data is copied in *constant-cache*
  - since there is less traffic in *constant-cache*, *there are good chances all other warp will find the data already in cache*, so no more traffic on the BUS



# Constant Memory Allocation

```
__constant__ type variable_name; // static

cudaMemcpyToSymbol(const_mem, &host_src, sizeof(type), cudaMemcpyHostToDevice);

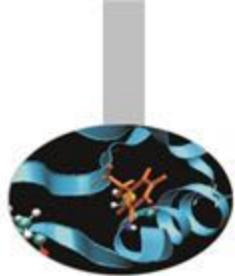
// warning
// cannot be dynamically allocated
```

```
type, constant :: variable_name
```

```
! warning
! cannot be dynamically allocated
```

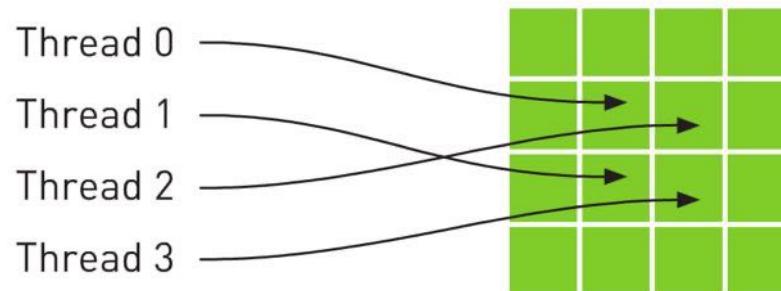
- data will reside in the constant memory address space
- has static storage duration (persists until the application ends)
- readable from all threads of a running kernel

# Texture Memory



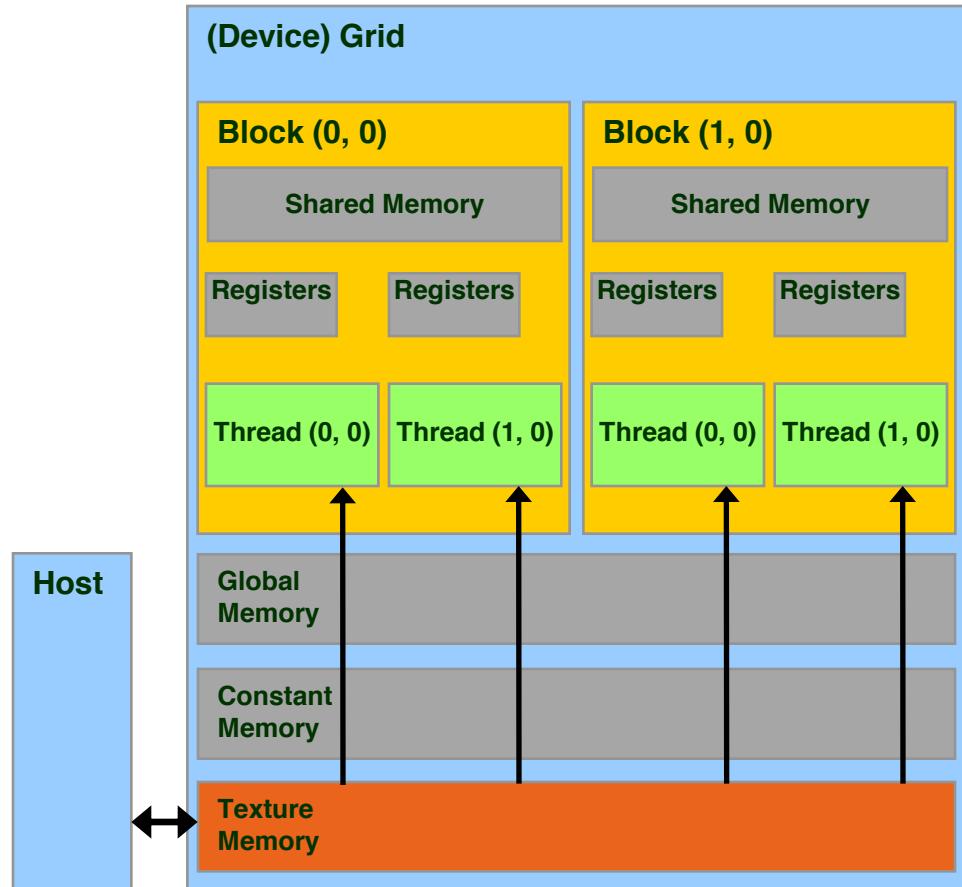
- **Read only**, must be set by the host;
- Load requests are cached (dedicated cache);
- specifically, texture memories and caches are designed for graphics applications where memory access patterns exhibit a great deal of spatial locality;
- Dedicated texture cache hardware provides:
  - Out-of-bounds index handling (clamp or wrap-around)
  - Optional interpolation (on-the-fly interpolation)
  - Optional format conversion
- could bring benefits if the threads within the same block access memory using regular 2D patterns, but you need appropriate binding;

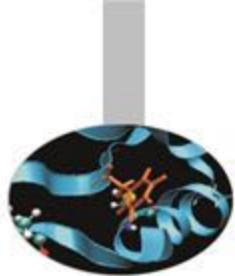
For typical linear patterns,  
global memory (if coalesced)  
is faster.



# Texture Memory

- **Texture Memory** is afterall a remain of basic graphic rendering functionality needs
- as for constant memory, data actually reside in the global memory, fetched across dedicated texture-cache
- data is accessed in **read-only** using special CUDA API function, called **texture fetch**
- Specifications:
  - address resolution is more efficient since it is performed on dedicated hardware
- specialized hardware for:
  - out-of-bound address resolution
  - floating-point interpolation
  - type conversion or bit operations





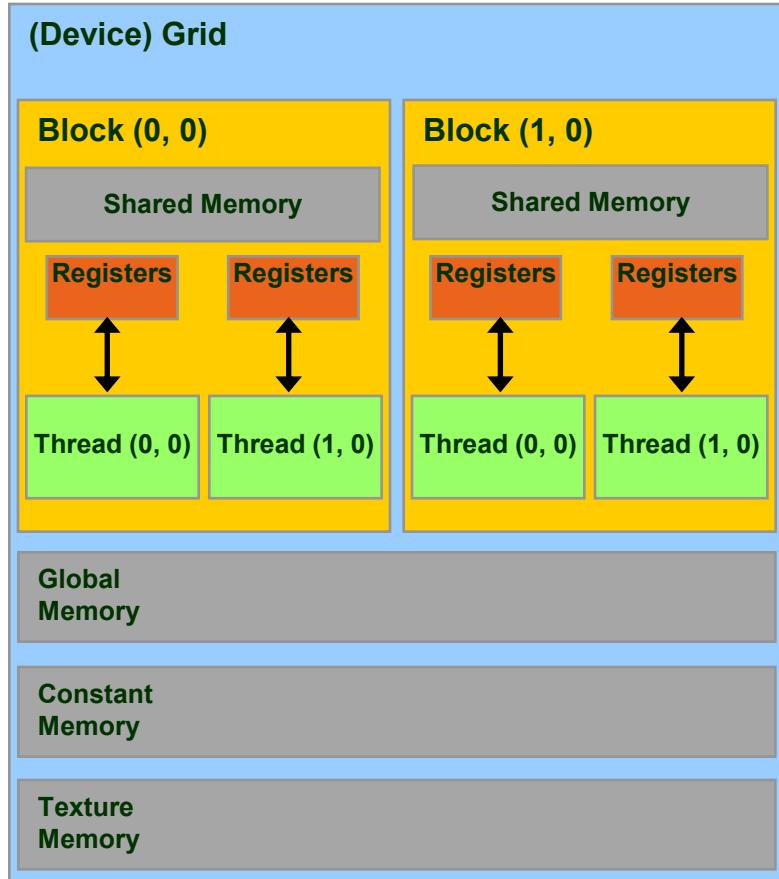
# Registers

- ➊ Just like CPU registers, access has no latency;
- ➋ used for scalar data local to a thread;
- ➌ taken by the compiler from the Streaming Multiprocessor (SM) pool and statically allocated to each thread;
  - ➍ each SM of a Fermi GPU has a 32KB register file, 64KB for a Kepler GPU
- ➎ *register pressure one of the most dangerous occupancy limiting factors.*

# Registers

- **registers** are used to store scalar or small array variables with frequent access by each thread
  - **Kepler, Pascal, Volta** : 255 registers per thread
- **WARNING:**
  - the less registers a kernel needs, the more blocks can be assigned to a SM
  - pay attention to *Register Pressure*: can be a limiting factor for performances
  - the number of register per kernel can be limited during *compile time*:  
**--maxregcount max\_registers**
  - the number of active blocks per kernel can be forced using the CUDA special qualifier  
**\_\_launch\_bounds\_\_**

```
__global__ void __launch_bounds__
(maxThreadsPerBlock, minBlocksPerMultiprocessor)
my_kernel(...) { ... }
```



# Local Memory

- **Local Memory** does not correspond to a real physical memory place
- Automatic variables are often placed in local memory by the compiler:
  - large structures or arrays that would consume too much register space
- If a kernel uses more registers than available (register spilling), the compiler shall move variables into local memory
- Local memory is often mapped to global memory
  - using the same *Caching* hierachies (L1 for read-only variables)
  - facing the same latency and bandwidth limitation of global memory
- In order to obtain information on how much local, constant, shared memory and registers are required for each kernel, you can provide the following compiler options

**--ptxas-options=-v**

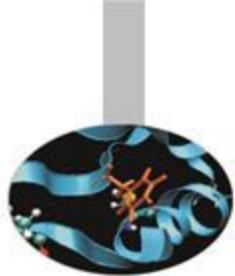
```
$ nvcc -arch=sm_60 --ptxas-options=-v my_kernel.cu
...
ptxas info : Used 34 registers, 60+56 bytes lmem, 44+40 bytes
smem, 20 bytes cmem[1], 12 bytes cmem[14]
...
```

# Local Memory

- **Local Memory** does not correspond to a real physical memory place
- Automatic variables are often placed in local memory by the compiler:
  - large structures or arrays that would consume too much register space
- If a kernel uses more registers than available (register spilling), the compiler shall move variables into local memory
- Local memory is often mapped to global memory
  - using the same *Caching* hierachies (L1 for read-only variables)
  - facing the same latency and bandwidth limitation of global memory
- In order to obtain information on how much local, constant, shared memory and registers are required for each kernel, you can provide the following compiler options

**--ptxas-options=-v**

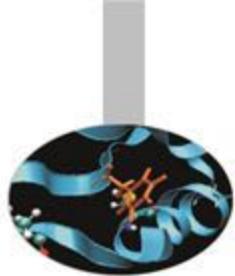
```
$ nvcc -arch=sm_60 --ptxas-options=-v my_kernel.cu
...
ptxas info : Used 34 registers, 60+56 bytes lmem, 44+40 bytes
smem, 20 bytes cmem[1], 12 bytes cmem[14]
...
```



# Occupancy

The board's occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor.

Keeping the hardware busy helps the warp scheduler to hide latencies.

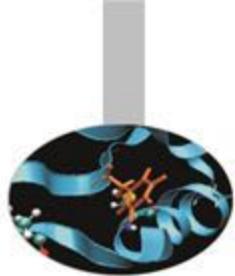


# Occupancy: constraints

Every board's resource can become an occupancy limiting factor:

- shared memory allocated per block,
- registers allocated per thread,
- block size
  - (max threads (warp) per SM/max blocks per SM)

Given an actual kernel configuration, is possible to predict the maximum *theoretical occupancy* allowed.



# Occupancy: block sizing tips

Some experimentation is required.

However there are some heuristic rules:

- ⌚ threads per block should be a **multiple of warp size**;
- ⌚ a minimum of **64 threads per block** should be used;
- ⌚ **128-256 threads per block** is universally known to be a good starting point for further experimentation;
- ⌚ prefer to split **very large** blocks into **smaller blocks**.

# Three steps for a CUDA porting

1. identify data-parallel, computational intensive portions
  1. isolate them into functions (CUDA kernels candidates)
  2. identify involved data to be moved between CPU and GPU
2. translate identified CUDA kernel candidates into real CUDA kernels functions
  1. choose the appropriate thread index map to access data
  2. change code so that each thread acts on its own data
3. modify code in order to manage memory and kernel calls
  1. allocate memory on the device
  2. transfer needed data from host to device memory
  3. insert calls to CUDA kernel with execution configuration syntax
  4. transfer resulting data from device to host memory

# Vector Sum

## 1. identify data-parallel computational intensive portions

```
int main(int argc, char *argv[]) {
 int i;
 const int N = 1000000;
 double u[N], v[N], z[N];

 initVector (u, N, 1.0);
 initVector (v, N, 2.0);
 initVector (z, N, 0.0);

 printVector (u, N);
 printVector (v, N);

 // z = u + v
 for (i=0; i<N; i++)
 z[i] = u[i] + v[i];

 printVector (z, N);

 return 0;
}
```

```
program vectoradd
integer :: i
integer, parameter :: N=1000000
real(kind(0.0d0)),dimension(N):: u, v, z

call initVector (u, N, 1.0)
call initVector (v, N, 2.0)
call initVector (z, N, 0.0)

call printVector (u, N)
call printVector (v, N)

! z = u + v
do i = 1,N
 z(i) = u(i) + v(i)
end do

call printVector (z, N)

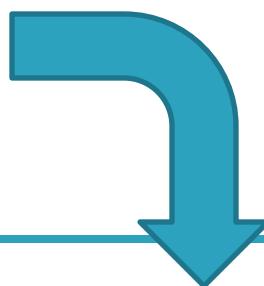
end program
```

# Vector Sum

## 2. translate the identified data-parallel portions into CUDA kernels

- each *thread* execute the same kernel, but acts on different data:
  - turn the loop into a CUDA kernel function
  - map each CUDA *thread* onto a unique index to access data
  - let each *thread* retrieve, compute and store its own data using the unique address
  - prevent out of border access to data if data is not a multiple of thread block size

```
// z = u + v
for (i=0; i<N; i++)
 z[i] = u[i] + v[i];
```



```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
 // index is a unique identifier of each GPU thread
 int index = blockIdx.x * blockDim.x + threadIdx.x ;
 if (index < N)
 z[index] = u[index] + v[index];
}
```

# Vector Sum

2. translate the identified data-parallel portions into CUDA kernels

(0)

(1)

(0)

(1)

(2)

(3)

(4)

(3)

^ (index)

```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
 // index is a unique identifier of each GPU thread
 int index = blockIdx.x * blockDim.x + threadIdx.x ;
 if (index < N)
 z[index] = u[index] + v[index];
}
```

The **\_\_global\_\_** qualifier  
declares this function to be a CUDA kernel

CUDA kernels are special C functions:

- can be called from host only
- must be called using the *execution configuration* syntax
- the return type must be *void*
- they are **asynchronous**: control is returned immediately to the host code
- an explicit synchronization is needed in order to be sure that a CUDA kernel has completed the execution

# CUDA kernels

```
__global__ void add(int *a, int *b, int *c) {

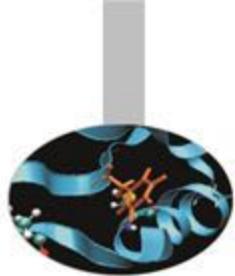
 int index = threadIdx.x + blockIdx.x *
blockDim.x; // global thread id

 if (index < N)
 c[index] = a[index] + b[index];

}
```

- Kernel functions indicated in the code by `__global__` (called by the host) or `__device__` (called by another function on the device).
- Must be void - cannot return values
- Remember that every CUDA thread executes the code in the function. May need to use if statements to make sure unallocated memory is not accessed.

# CUDA Function modifiers



CUDA extends C function declarations with three qualifier keywords.

| Function declaration                                   | Executed on the | Only callable from the |
|--------------------------------------------------------|-----------------|------------------------|
| <code>__device__</code><br>( <i>device functions</i> ) | device          | device                 |
| <code>__global__</code><br>( <i>kernel function</i> )  | device          | host                   |
| <code>__host__</code><br>( <i>host functions</i> )     | host            | host                   |

- **CUDA C API:** `cudaMalloc(void **p, size_t size)`
  - allocates size bytes of GPU global memory
  - p is a valid device memory address (i.e. SEGV if you dereference p on the host)

```
double *u_dev, *v_dev, *z_dev;

cudaMalloc((void **) &u_dev, N * sizeof(double));
cudaMalloc((void **) &v_dev, N * sizeof(double));
cudaMalloc((void **) &z_dev, N * sizeof(double));
```

- in CUDA Fortran the attribute **device** needs to be used while declaring a GPU array. The array can be allocated by using the Fortran statement **allocate**:

```
real(kind(0.0d0)), device, allocatable, dimension(:, :) :: u_dev, v_dev, z_dev

allocate(u_dev(N), v_dev(N), z_dev(N))
```

- CUDA C API:

```
cudaMemcpy(void *dst, void *src, size_t size, direction)
```

- copy size bytes from the src to dst buffer

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

- in CUDA Fortran you can rely on operator overload or use the array syntax to slice subelements of the array

```
u_dev = u ; v_dev = v
```

Insert calls to CUDA kernels using the execution configuration syntax:

```
kernelCUDA<<<numBlocks , numThreads>>>(. . .)
```

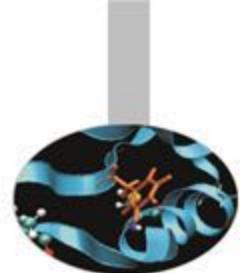
specifying the thread/block hierarchy you want to apply:

- **numBlocks**: specify grid size in terms of thread blocks along each dimension
- **numThreads**: specify the block size in terms of threads along each dimension

```
dim3 numThreads(32);
dim3 numBlocks((N + numThreads - 1) / numThreads.x);
gpuVectAdd<<<numBlocks , numThreads>>>(N, u_dev, v_dev, z_dev);
```

```
type(dim3) :: numBlocks, numThreads
numThreads = dim3(32, 1, 1)
numBlocks = dim3((N + numThreads%x - 1) / numThreads%x, 1, 1)
call gpuVectAdd<<<numBlocks , numThreads>>>(N, u_dev, v_dev, z_dev)
```

# CUDA variable qualifiers



| Variable declaration                                           | memory   | lifetime    | scope  |
|----------------------------------------------------------------|----------|-------------|--------|
| Automatic scalar variables                                     | register | kernel      | thread |
| Automatic array variables<br><code>__device__ __local__</code> | local    | kernel      | thread |
| <code>__device__ __shared__</code>                             | shared   | kernel      | block  |
| <code>__device__</code>                                        | global   | application | grid   |
| <code>__device__ __constant__</code>                           | constant | application | grid   |

- 💡 Global variables are often used to pass information from one kernel to another.
- 💡 Constant variables are often used for providing input values to kernel functions.

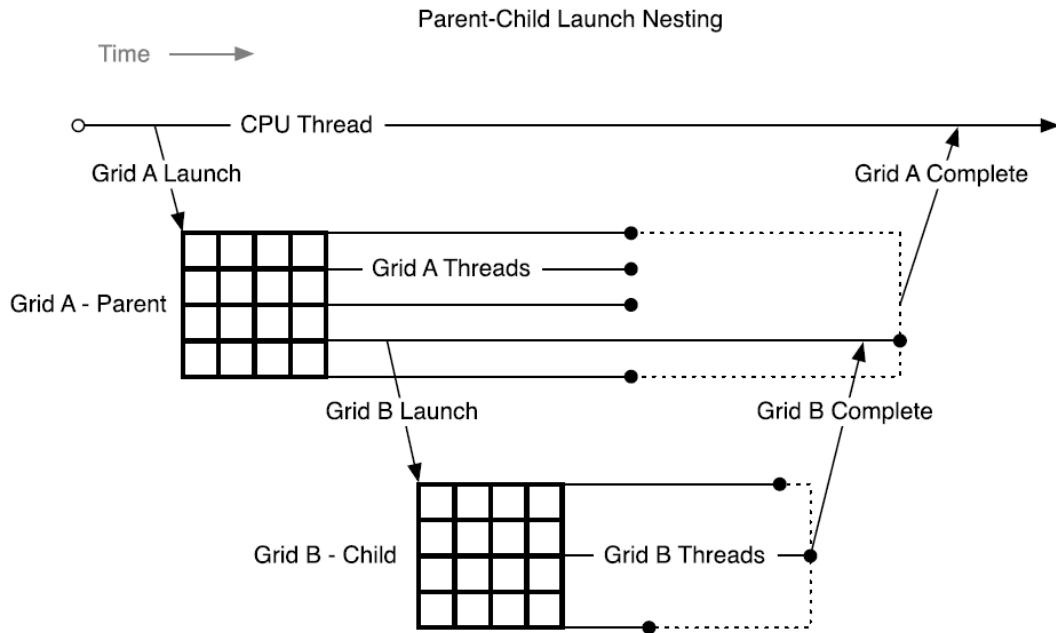


# Kepler: dynamic parallelism

- One of the biggest CUDA limitations is the need to fit a single grid configuration for the whole kernel.

If you need to reshape the grid, you have to resync back to host and split your code.

- Kepler K20 (in addition to CUDA 5.x) introduced *Dynamic Parallelism*
- It enables a global kernel to be called from within another kernel
- The child grid can be *dynamically sized and optionally synchronized*



```
__global__ ChildKernel(void* data) {
 //Operate on data
}

__global__ ParentKernel(void *data) {
 ChildKernel<<<16, 1>>>(data);
}

// In Host Code:
ParentKernel<<<256, 64>>>(data);
```

- Open Compute Language (OpenCL): “The Open Standard for Heterogeneous Parallel Programming”
  - Open cross-platform framework for programming modern multicore and heterogeneous systems
- Supports wide range of applications and architectures, including GPUs
  - Supported on NVIDIA Tesla + AMD FireStream
- See <http://www.khronos.org/opencl/>

- NVIDIA support both CUDA and OpenCL as APIs to the hardware.
  - But put much more effort into CUDA
  - CUDA more mature, well documented and performs better
- OpenCL and C for CUDA conceptually very similar
  - Very similar abstractions, basic functionality etc
  - Different names e.g. “Thread” CUDA -> “Work Item” (OpenCL)
  - Porting between the two should in principle be straightforward
- OpenCL is a lower level API than C for CUDA
  - More work for programmer
- OpenCL obviously portable to other systems
  - But in reality work will still need to be done for efficiency on different architecture
- OpenCL may well catch up with CUDA given time

# OpenACC Friendly Disclaimer

OpenACC  
Directives

Easily Accelerate  
Applications

OpenACC does not make GPU programming easy. (...)

GPU programming and parallel programming is not easy. It cannot be made easy. However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing. OpenACC is designed to fill that role.

(Michael Wolfe, The Portland Group)

# OpenACC History

- OpenACC is a high-level specification with compiler directives for expressing parallelism for accelerators.
  - Portable to a wide range of accelerators.
  - One specification for Multiple Vendors and Multiple Devices
- OpenACC specification was released in November 2011.
  - Original members: CAPS, Cray, NVIDIA, Portland Group
- OpenACC 2.0 was released in June 2013
  - More functionality
  - Improve portability
- OpenACC 2.5 in November 2015
- OpenACC 2.6 in November 2017
- OpenACC had more than 10 member organizations

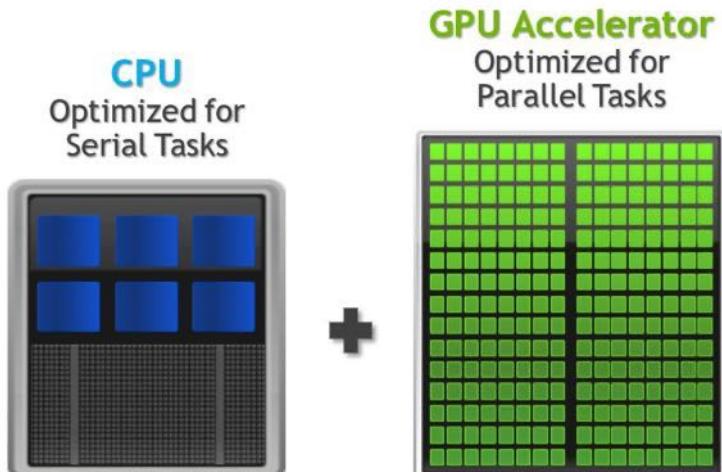
# OpenACC Info & Vendors

- <http://www.openacc.org>
- Novelty in OpenACC 2.0 are significant
  - OpenACC 1.0 maybe not very mature...
- Some changes are inspired by the development of CUDA programming model
  - but the standard is not limited to NVIDIA GPUs: one of its pros is the **interoperability** between platforms
- Standard implementation
  - CRAY provides full OpenACC 2.0 support in CCE 8.2
  - PGI support to OpenACC 2.5 is almost complete (starting from version 15.1)
    - Support for OpenACC 2.0 starting from 14.1
  - **GNU implementation effort ongoing** (there is a partial implementation in the 5.1 release and a dedicated branch for 7.1 realease)
- We will focus on PGI compiler
  - 30 days trial license useful for testing
- PGI:
  - all-in-one compiler, easy usage
  - sometimes the compiler tries to help you...
  - but also a constraint on the compiler to use

# OpenACC – Simple, Powerful, Portable

```
main()
{
 <serial code>

 #pragma acc kernels
 //automatically runs on GPU
 {
 <parallel code>
 }
}
```



## 1. Simple:

- Simple compiler directives
- Directives are the easy path to accelerate compute intensive applications
- Compiler parallelizes code

## 2. Open:

- OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

## 3. Portable:

- Works on many-core GPUs and multi-core CPUs

## 4. Powerful:

- GPU Directives allow complete access to the massive parallel power of a GPU

# OpenMP 4.0/4.5 alternative

- OpenMP 4.0/4.5 supports heterogeneous systems (accelerators/devices)
- What's new in OpenMP 4.x for support accelerator model
  - **Target regions**
    - Structured and unstructured target data regions
      - `omp target [clause[,] clause],...`
      - `omp declare target`
    - **Asynchronous** execution (`nowait`) and **data dependency** (`depend`)
  - Manage device data environment
    - **Data mapping** APIs
      - `map ([map-type:] list)`
    - **Data regions**
      - `omp target data [clause[,] clause], ...`
      - `omp target enter/exit data [clause[,] clause], ...`
  - **Parallelism & Workshare** for devices
    - `omp teams [clause[,] clause],...`
    - `omp distribute [clause[,] clause],...`
  - **SIMD** parallelism

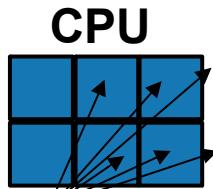
# Familiar to OpenMP Programmers

OpenMP

```
main() {
 double pi = 0.0; long i;

#pragma omp parallel for reduction(+:pi)
for (i=0; i<N; i++)
{
 double t = (double) ((i+0.05)/N);
 pi += 4.0/(1.0+t*t);
}

printf("pi = %f\n", pi/N);
}
```

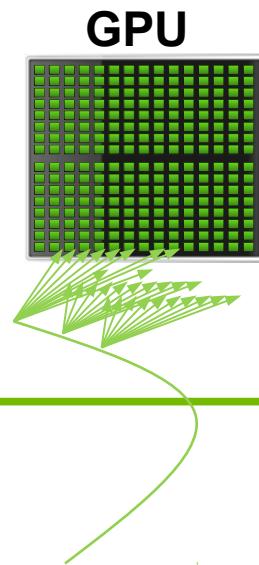


OpenACC

```
main() {
 double pi = 0.0; long i;

#pragma acc parallel loop reduction(+:pi)
for (i=0; i<N; i++)
{
 double t = (double) ((i+0.05)/N);
 pi += 4.0/(1.0+t*t);
}

printf("pi = %f\n", pi/N);
}
```



# MPI

A collage of several tutorials

Mirko Cestari  
m.cestari@cineca.it  
High Performance Computing department

# Parallel programming with MPI

Introduction and Point-to-Point Communications



# Distributed Memory Programming with MPI

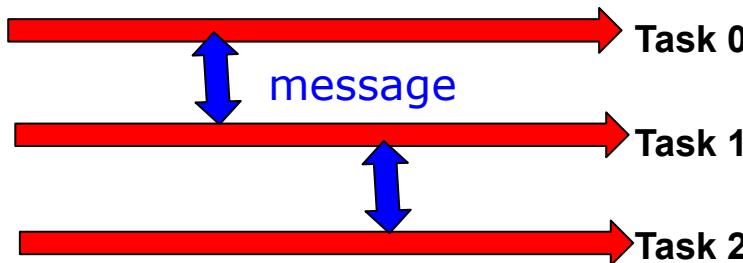
Moreno Marzolla

Dip. di Informatica—Scienza e Ingegneria (DISI)  
Università di Bologna

[moreno.marzolla@unibo.it](mailto:moreno.marzolla@unibo.it)

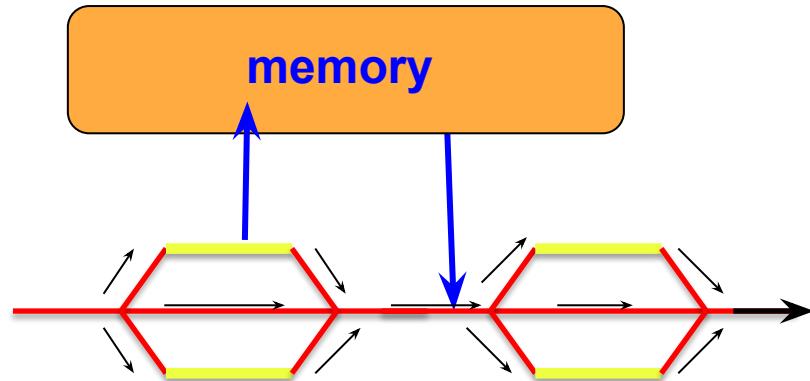
# Message passing and shared memory parallelism

## *message passing*



Multiple tasks exchange data via explicit messages

## *shared memory*



Program splits into threads which share data via variables in shared memory

# Message Passing

- Unlike the shared memory model, resources are local;
- Each process operates in its own environment (logical address space) and communication occurs via the exchange of messages;
- Messages can be instructions, data or synchronisation signals;
- The message passing scheme can also be implemented on shared memory architectures;
- Delays are much longer than those due to shared variables in the same memory space;

# Advantages and Drawbacks

- **Advantages**
  - Communications hardware and software are important components of HPC system and often very highly optimised;
  - Portable and scalable;
  - Long history (many applications already written for it);
- **Drawbacks**
  - Explicit nature of message-passing is error-prone and discourages frequent communications;
  - Most serial programs need to be completely re-written;
  - High memory overheads.

# A brief history of MPI

- Before the 1990's, programmers weren't as lucky as us. Many libraries could facilitate building parallel applications, but there was not a standard accepted way of doing it.
- Since most libraries at this time used the same message passing model with only minor feature differences among them, the authors of the libraries and others came together at the Supercomputing 1992 conference to define a standard interface for performing message passing - the Message Passing Interface.
- This standard interface would allow programmers to write parallel applications that were portable to all major parallel architectures.

# A brief history of MPI

- By 1994, a complete interface and standard was defined (MPI-1).  
Keep in mind that MPI is *only* a definition for an interface, as OpenMP.
- The first implementation has been made available in 1995.
- MPI Forum <https://www.mpi-forum.org/>
- The standards <https://www.mpi-forum.org/docs/>
- Another official reference  
<https://www.mcs.anl.gov/research/projects/mpi/>
- A collection of tutorials <https://mpitutorial.com/tutorials/>



# The most important concept in message passing is...

...to minimize message passing as much as possible!

To maximise performance, the program should spend as little time as possible communicating data or waiting for other processes.



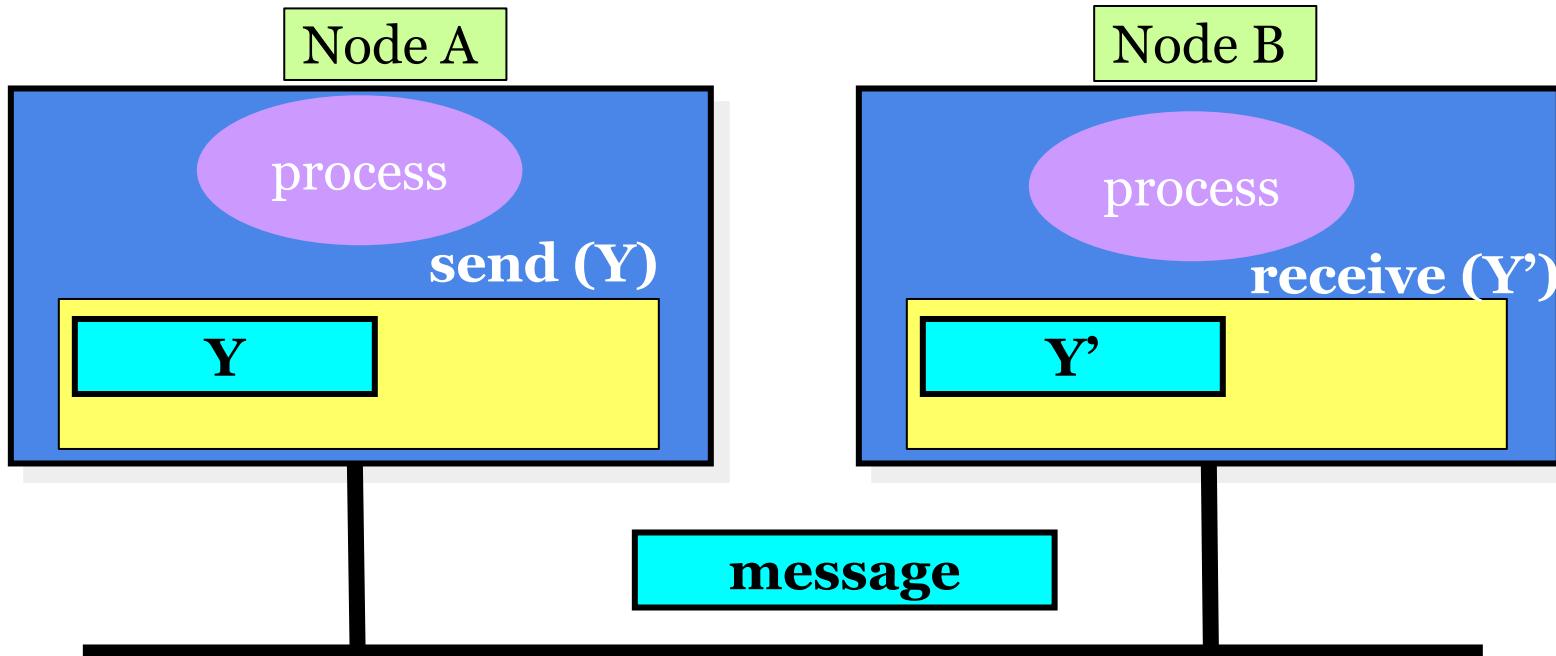
# Data transfer and Synchronization

The sender process cooperates with the destination process

The communication system must allow the following three operations:

- **send(*message*)**
- **receive(*message*)**
- **synchronization**

# MPI Programming Model





# Goals of the MPI standard

MPI's prime goals are:

- To allow efficient implementation
- To provide source-code portability

MPI also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

MPI2 further extends the library power (parallel I/O, Remote Memory Access, Multi Threads)

MPI3 aims to support exascale by including non-blocking collectives, improved RMA and neighbourhood collectives.



# Basic Features of MPI Programs

An MPI program consists of multiple instances of a serial program that communicate by library calls.

Calls may be roughly divided into four classes:

1. Calls used to **initialize, manage, and terminate** communications
2. Calls used to **communicate between pairs** of processors. (point to point communication)
3. Calls used to **communicate among groups** of processors. (collective communication)
4. Calls to create **data types**.



# A note about MPI Implementations

- The MPI standard defines the functionalities and the API, i.e. what the C or FORTRAN calls should look like.
- The MPI standard **does not define how the calls should be performed at the system level** (algorithms, buffers, etc) or how the environment is set up (env variables, mpirun or mpiexec, libraries, etc). This is left to the **implementation**.
- There are various implementations (IntelMPI, OpenMPI, MPICH, HPMPI, etc) which have different performances, features and standards compliance.
- On some clusters (e.g. Galileo, Marconi) you may choose which MPI to use, on other systems you have only the vendor-supplied version (IBM MPI for FERMI).



# A First Program: Hello World!

## Fortran

```
PROGRAM hello

INCLUDE 'mpif.h'

INTEGER err

CALL MPI_INIT(err)

PRINT *, "hello world!"

CALL MPI_FINALIZE(err)

END
```

## C

```
#include <stdio.h>

#include <mpi.h>

void main (int argc, char * argv[])

{

 int err;

 err = MPI_Init(&argc, &argv);

 printf("Hello world!\n");

 err = MPI_Finalize();

}
```

# Header files

All Subprogram that contains calls to MPI subroutine must include the MPI header file

C:

```
#include <mpi.h>
```

Fortran:

```
include 'mpif.h'
```

Fortran 90:

```
USE MPI
```

Fortran 08 (MPI-3):

```
USE MPI_F08
```

The header file contains definitions of MPI constants, MPI types and functions

## FORTRAN note:

The FORTRAN include and module forms are *not equivalent*: the module can also do type checking. Some compilers gave problems with the module but it is now highly recommended to use the module, particularly for FORTRAN 2008 (most rigorous type-checking)



# MPI function format

C:

```
int error = MPI_Xxxxxx(parameter, . . .);
MPI_Xxxxxx(parameter, . . .);
```

FORTRAN:

```
CALL MPI_XXXXXX(parameter, IERROR)
INTEGER IERROR
```



# Initializing MPI

C:

```
int MPI_Init(int *argc, char ***argv)
```

FORTRAN:

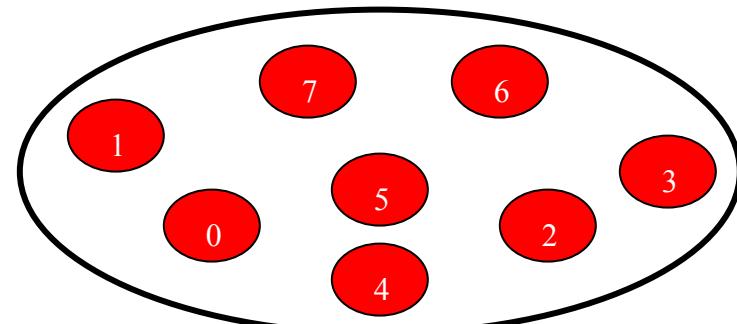
```
INTEGER IERROR
MPI_INIT(IERROR)
```

Must be first MPI call: initializes the message passing routines

# MPI Communicator

- In MPI it is possible to divide the total number of processes into groups, called *communicators*.
- The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.
- The communicator that includes all processes is called **MPI\_COMM\_WORLD**
- **MPI\_COMM\_WORLD** is the default communicator (automatically defined)

All MPI communication subroutines have a communicator argument.  
The Programmer can define many communicators at the same time



**MPI\_COMM\_WORLD**



# Communicator Size

How many processes are associated with a communicator?

C:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

FORTRAN:

```
INTEGER COMM, SIZE, IERR
```

```
OUTPUT: SIZE
```

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```



# Process Rank

How can you identify different processes?

What is the ID of a processor in a group?

C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

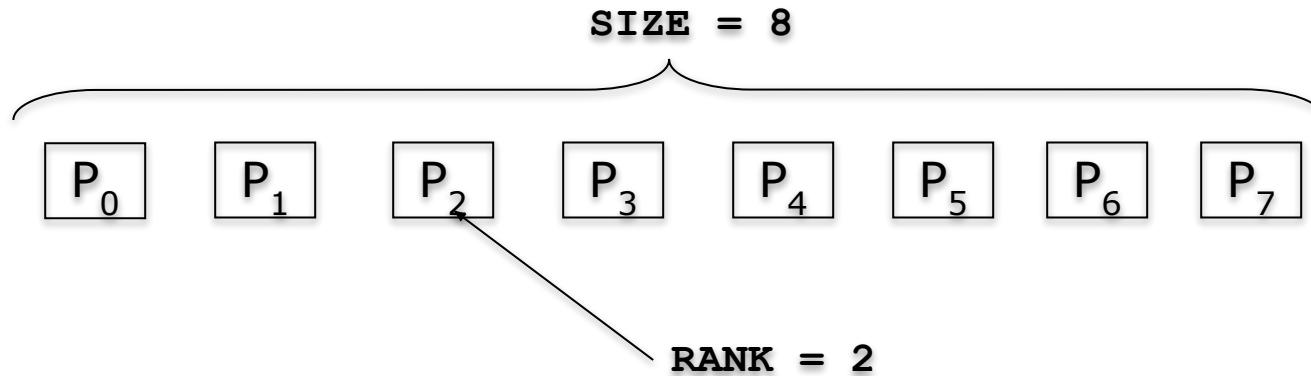
Fortran:

```
CALL MPI_COMM_RANK(COMM, RANK, IERR)
INTEGER COMM, RANK, IERR
OUTPUT: RANK
```

- *rank* is an integer that identifies the Process inside the communicator *comm*
- `MPI_COMM_RANK` is used to find the rank (the name or identifier) of the Process running the code

# Communicator Size and Process Rank / 1

How many processes are contained within a communicator?



**size** is the number of processes associated to the communicator

**rank** is the index of the process within a group associated to a communicator (**rank** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication



# Exiting MPI

Finalizing MPI environment

C:

```
int MPI_Finalize()
```

Fortran:

```
INTEGER IERR
```

```
CALL MPI_FINALIZE(IERR)
```

This two subprograms should be called by all processes, and no other MPI calls are allowed before `mpi_init` and after `mpi_finalize`. Before and after these calls all instances execute the same code (as in serial execution)



# MPI\_ABORT

- Usage

```
int MPI_Abort(MPI_Comm comm, int errorcode);
```

- Description

Terminates all MPI processes associated with the communicator comm; in most systems (all to date), terminates *all* processes.



# A Template for C MPI programs

```
#include <stdio.h>
#include <mpi.h>

void main (int argc, char * argv[])
{
 int err, nproc, myid;

 err = MPI_Init(&argc, &argv) ;
 err = MPI_Comm_size(MPI_COMM_WORLD, &nproc) ;
 err = MPI_Comm_rank(MPI_COMM_WORLD, &myid) ;

 /* *** INSERT YOUR PARALLEL CODE HERE *** */

 err = MPI_Finalize() ;
}
```

# Hello, world!

```
/* mpi-hello.c */

#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
 int rank, size, len;
 char hostname[MPI_MAX_PROCESSOR_NAME];
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank); No MPI call before this line
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Get_processor_name(hostname, &len);
 printf("Greetings from process %d of %d running on %s\n",
 rank, size, hostname);
 MPI_Finalize(); No MPI call after this line
 return 0;
}
```



# Compiling and Running MPI programs

- Implementation is system dependent but it is usual to use the “wrapped” version of the compiler to include the MPI headers and link in the MPI libraries. Wrapped compilers tend to be called `mpif90`, `mpicc`, `mpic++`, etc.
- On HPC systems MPI programs are run via the batch system with appropriate settings. For debugging sometimes it is possible to open interactive sessions (e.g. SLURM on GALILEO).
- a program such as **mpirun** or **mpiexec** is then used to launch multiple instances of the program on the assigned nodes. For MARCONI and GALILEO you can also use the **srun** command.

# Usage

## COMPILE

- GCC-based MPI   `mpicc hello.c -o hello`
- INTEL-based MPI `mpiicc hello.c -o hello`

## RUN

Here you decide how many processes will execute the computation

- `mpirun / mpiexec -np 100 ./hello`
- The path is important!
- IN this case you execute all the 100 processes on hpcocapie01

# Usage

<https://software.intel.com/en-us/mpi-developer-reference-linux-introduction>  
<https://software.intel.com/en-us/mpi-developer-reference-linux-global-options>

```
mpiexec -nolocal -hostfile list.txt -perhost 1 -np 11 /bin/hostname
```

```
[dadagostino@hpcocapie01 mpi]$ cat list.txt
hpcocapie01
hpcocapie02
hpcocapie03
hpcocapie04
hpcocapie05
hpcocapie06
hpcocapie07
hpcocapie08
hpcocapie09
hpcocapie10
hpcocapie11
```

You can choose the file name

2 on hpcocapie02

Try

```
mpiexec -hostfile list.txt -np 11 /bin/hostname
```

```
mpiexec -hostfile list.txt -nolocal -np 11 /bin/hostname
```

```
mpiexec -hostfile list.txt -nolocal -perhost 1 -np 15 /bin/hostname
```

hpcocapie02.ge.infn.it  
hpcocapie02.ge.infn.it  
hpcocapie05.ge.infn.it  
hpcocapie07.ge.infn.it  
hpcocapie08.ge.infn.it  
hpcocapie03.ge.infn.it  
hpcocapie04.ge.infn.it  
hpcocapie06.ge.infn.it  
hpcocapie09.ge.infn.it  
hpcocapie11.ge.infn.it  
hpcocapie10.ge.infn.it

# Example

```
mpiexec -hostfile list.txt -nolocal -perhost 1 -np 15 ./hello2
```

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char * argv[])
{
 int err, nproc, myid;
 err = MPI_Init(&argc, &argv);
 err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
 printf("I'm proc %d of total %d\n",myid,nproc);
 err = MPI_Finalize();
}
```

# Using bsub - examples

<https://web.ge.infn.it/calcolo/joomla/2-uncategorised/106-farm-hpc-ocapie#h4-sottomissione-di-job>

(ITA, manual at the end of the page). You'll receive results on the ge.infn.it mail address

bsub -q ocapie -n 30 mpiexec -np 30 /bin/hostname

All on a single node

bsub -q ocapie -R span[ptile=1] -n 30 mpiexec -np 30 /bin/hostname  
nodes

Never executed, you should have 30

bjobs 155735

| JOBID  | USER    | STAT | QUEUE  | FROM_HOST   | EXEC_HOST | JOB_NAME   | SUBMIT_TIME |
|--------|---------|------|--------|-------------|-----------|------------|-------------|
| 155735 | dadagos | PEND | ocapie | hpcocapie01 |           | */hostname | Nov 8 17:03 |

bsub -q ocapie -R span[ptile=5] -n 30 mpiexec -np 30 /bin/hostname

Job was executed on host(s) <5\*hpcocapie07.ge.infn.it>, <5\*hpcocapie01.ge.infn.it>, <5\*hpcocapie02.ge.infn.it>  
<5\*hpcocapie05.ge.infn.it>, <5\*hpcocapie08.ge.infn.it>, <5\*hpcocapie09.ge.infn.it>

bsub -q ocapie -n 300 mpiexec -np 300 /bin/hostname

Job was executed on host(s) <256\*hpcocapie01.ge.infn.it>, <44\*hpcocapie07.ge.infn.it>

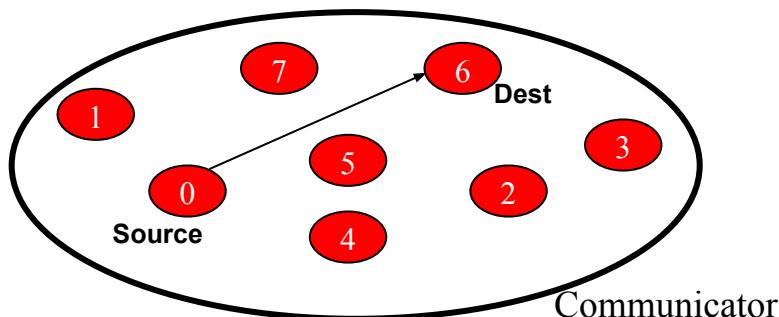
# SPMD (Single Program Multiple Data)

- The **same** program is executed by  $P$  processes
- Each process may choose a different execution path depending on its ID (*rank*)

```
...
MPI_Init(...);
foo(); /* executed by all processes */
if (my_id == 0) {
 do_something(); /* executed by process 0 only */
} else {
 do_something_else(); /* executed by all other processes */
}
MPI_Finalize();
...
```

# Point-to-Point Communication

- It is the basic communication method provided by MPI library. Communication between 2 processes
- It is conceptually simple: source process A sends a message to destination process B, B receive the message from A.
- Communication take places within a **communicator**
- Source and Destination are identified by their rank in the communicator





# Point-to-Point communication quick example

The construction

```
if rank equals i
 send information

else if rank equals j
 receive information
```

is very common in MPI programs. Often one rank (usually rank 0) is selected for particular tasks which can be or should be done by one task only such as reading or writing files, giving messages to the user or for managing the overall logic of the program (e.g. master-slave).

# A Simple MPI Program

```
/* mpi-point-to-point.c */
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
 int rank, buf;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 /* process 0 sends and process 1 receives */
 if (rank == 0) {
 buf = 123456;
 MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
 } else if (rank == 1) {
 MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
 printf("Received %d\n", buf);
 }

 MPI_Finalize();
 return 0;
}
```

# MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI\_ANY\_TAG** as the tag in a receive

# The Message

- **Buffers** of data are exchanged. They are series of **count** elements of a particular MPI **data type**
- In this way MPI knows how many bytes (length) has to send/receive.
- This allows MPI programs to run in **heterogeneous** environments
- C types are different from Fortran types.

Messages are identified by their envelopes. A message could be exchanged only if the sender and receiver specify the correct envelope

## Message Structure

| envelope |             |              |     | body   |       |          |
|----------|-------------|--------------|-----|--------|-------|----------|
| source   | destination | communicator | tag | buffer | count | datatype |
|          |             |              |     |        |       |          |

# Data Types

- MPI Data types
  - Basic types (portability)
  - Derived types (MPI\_Type\_xxx functions)
- Derived type can be built up from basic types
- User-defined data types allows MPI to use non-contiguous buffers of data
- MPI defines '**handles**' to allow programmers to refer to data types
  - declare the **right type** of the handles as defined in the API of language (C/Fortran)
  - MPI data type and language data type have to be **compatible** (MPI\_INTEGER and INTEGER)



# C - MPI Intrinsic Datatypes

| MPI Data type                   | C Data type                                                                                                                                                                             |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>MPI_CHAR</code>           | <code>signed char</code>                                                                                                                                                                |
| <code>MPI_SHORT</code>          | <code>signed short int</code>                                                                                                                                                           |
| <code>MPI_INT</code>            | <code>signed int</code>                                                                                                                                                                 |
| <code>MPI_LONG</code>           | <code>signed long int</code>                                                                                                                                                            |
| <code>MPI_UNSIGNED_CHAR</code>  | <code>unsigned char</code>                                                                                                                                                              |
| <code>MPI_UNSIGNED_SHORT</code> | <code>unsigned short int</code>                                                                                                                                                         |
| <code>MPI_UNSIGNED</code>       | <code>unsigned int</code>                                                                                                                                                               |
| <code>MPI_UNSIGNED_LONG</code>  | <code>unsigned long int</code>                                                                                                                                                          |
| <code>MPI_FLOAT</code>          | <code>float</code>                                                                                                                                                                      |
| <code>MPI_DOUBLE</code>         | <code>double</code>                                                                                                                                                                     |
| <code>MPI_LONG_DOUBLE</code>    | <code>long double</code>                                                                                                                                                                |
| <code>MPI_BYTE</code>           |                                                                                                                                                                                         |
| <code>MPI_PACKED</code>         | <a href="https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node32.htm">https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node32.htm</a> |



# For a communication to succeed ...

1. Sender must specify a valid destination rank.
2. Receiver must specify a valid source rank.
3. The communicator must be the same.
4. Tags must match.
5. Buffers must be large enough.



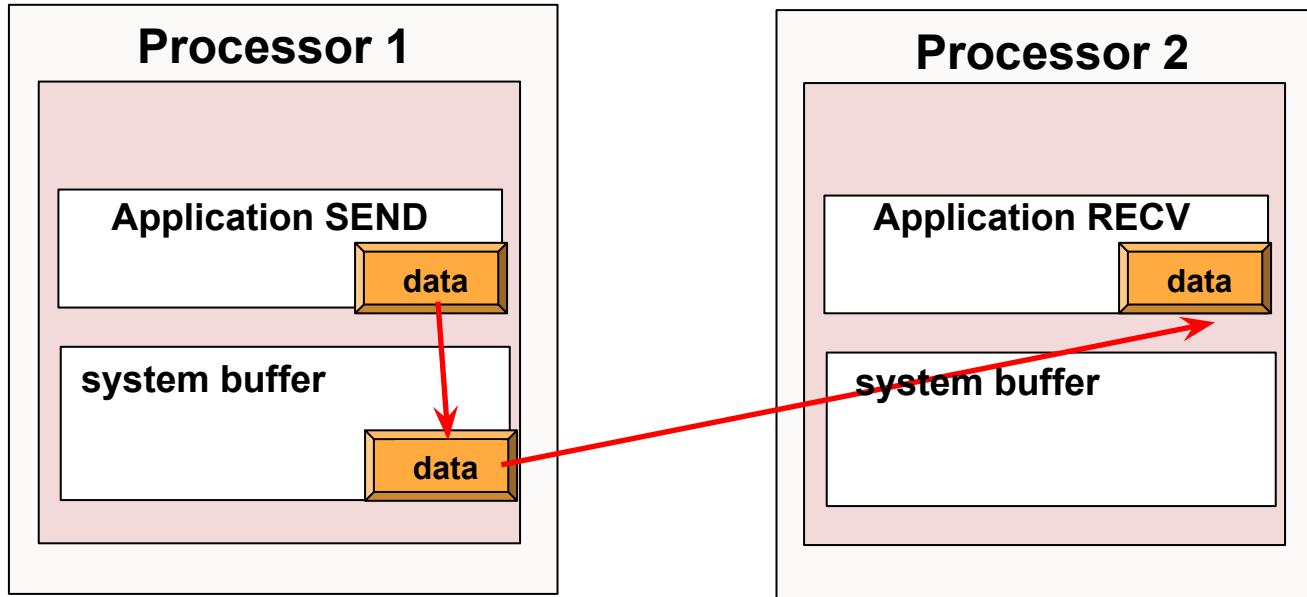
# Completion

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. The MPI implementation is able to deal with the exchanging data when the two tasks are out of sync.
- **Completion** of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can be used after completion

# Blocking communications

- Most of the MPI point-to-point routines can be used in either **blocking** or **non-blocking** mode.
- Blocking:
  - A blocking **send** returns **after it is safe to modify the sent buffer**. Safe does not imply that the data was actually received - it may be stored in a system buffer.
  - A blocking **send** **can be synchronous**.
  - A blocking **send** **can be asynchronous** if a system **buffer** is used to hold the data for eventual delivery.
  - A blocking **receive** only "returns" after the data **has arrived and is ready for use by the program**.

# Blocking Communications





# Standard Send and Receive

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
 int dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype type,
 int source, int tag, MPI_Comm comm, MPI_Status *status);
```



# Standard Send and Receive

Basic blocking point-to-point communication routine in MPI.

Fortran:

```
MPI_SEND(buf, count, type, dest, tag, comm, ierr)
MPI_RECV(buf, count, type, source, tag, comm, status, ierr)
```

Message body

Message envelope

|               |                                                                                                                                                |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>buf</b>    | send/receive buffer (memory located in sender/receiver memory).                                                                                |
| <b>count</b>  | (INTEGER) number of contiguous elements of <b>buf</b> to be sent                                                                               |
| <b>type</b>   | (INTEGER) MPI type of <b>buf</b> ( <i>implicitly number of bytes</i> )                                                                         |
| <b>dest</b>   | (INTEGER) rank of the destination process                                                                                                      |
| <b>tag</b>    | (INTEGER) number identifying the message                                                                                                       |
| <b>comm</b>   | (INTEGER) communicator of the sender and receiver                                                                                              |
| <b>status</b> | (INTEGER) array of size <b>MPI_STATUS_SIZE</b> containing<br>communication status information (Orig Rank, Tag, Number of elements<br>received) |
| <b>ierr</b>   | (INTEGER) error code (if <b>ierr=0</b> no error occurs)                                                                                        |



# Send and Receive - C

```
#include <stdio.h>
#include <mpi.h>

void main (int argc, char * argv[])
{
 int err, nproc, myid;
 MPI_Status status;
 float a[2];

 err = MPI_Init(&argc, &argv);
 err = MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

 if(myid == 0) {
 a[0] = 3.0, a[1] = 5.0;
 MPI_Send(a, 2, MPI_FLOAT, 1, 10, MPI_COMM_WORLD);
 } else if(myid == 1) {
 MPI_Recv(a, 2, MPI_FLOAT, 0, 10, MPI_COMM_WORLD, &status);
 printf("%d: a[0]=%f a[1]=%f\n", myid, a[0], a[1]);
 }

 err = MPI_Finalize();
}
```

# **MPI\_Status**

- **MPI\_Status** is a C structure with the following fields:
  - `int MPI_SOURCE;`
  - `int MPI_TAG;`
  - `int MPI_ERROR;`
- Therefore, a process can check the actual source and tag of a message received with `MPI_ANY_TAG` or `MPI_ANY_SOURCE`

# **MPI\_Get\_count()**

```
MPI_Get_count(&status, datatype, &count);
```

- **MPI\_Recv** may complete even if less than *count* elements have been received
  - Provided that the matching **MPI\_Send** actually sent fewer elements
- **MPI\_Get\_count** can be used to know how many elements of type *datatype* have actually been received
- See [mpi-get-count.c](#)

# Example

```
#include <stdio.h> #include <stdlib.h>
#include <time.h> #include <mpi.h>
#define BUFLEN 16

int main(int argc, char *argv[]) {
 int rank, buf[BUFLEN] = {0};
 int count, i;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 /* Process 0 sends, Process 1 receives */
 if (rank == 0) {
 srand(time(NULL));
 /* Fills the buffer with a random number of integers */
 count = 1 + rand()%BUFLEN;
 for (i=0; i<count; i++) { buf[i] = i; }
 MPI_Send(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
 printf("Sent %d integers\n", count);
 }
```

```
else if (rank == 1) {
 MPI_Recv(buf, BUFLEN, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);
 MPI_Get_count(&status, MPI_INT, &count);
 printf("Received %d integers: ", count);
 for (i=0; i<count; i++) { printf("%d ", buf[i]); }
 printf("\n");
}
MPI_Finalize();
return 0;
}
```

[[dadagostino@hpcocapie01 mpi]\$ mpiexec -np 2 ./mpi-get-count  
Sent 12 integers  
Received 12 integers: 0 1 2 3 4 5 6 7 8 9 10 11  
[[dadagostino@hpcocapie01 mpi]\$ mpiexec -np 2 ./mpi-get-count  
Sent 16 integers  
Received 16 integers: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  
[[dadagostino@hpcocapie01 mpi]\$ mpiexec -np 2 ./mpi-get-count  
Sent 15 integers  
Received 15 integers: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
[[dadagostino@hpcocapie01 mpi]\$ mpiexec -np 2 ./mpi-get-count  
Sent 15 integers  
Received 15 integers: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
[[dadagostino@hpcocapie01 mpi]\$ mpiexec -np 2 ./mpi-get-count  
Sent 15 integers  
Received 15 integers: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14



# Non Blocking communications

- Non-blocking **send** and **receive** routines will **return almost immediately**. They do not wait for any communication events to complete
- Non-blocking operations simply "request" the MPI library to perform the operation **when it is possible**. The user can not predict when that will happen.
- It is unsafe to modify the application buffer until you know for a fact that the requested non-blocking operation was actually performed by the library. There are "**wait**" routines used to do this.
- Non-blocking communications are primarily used to **overlap computation with communication**.
- Another important use is to **overlap communication with communication**, i.e. avoiding serialization issues (for example, when a loop of communications is involved)



# Non-Blocking Send and Receive

C:

```
int MPI_Isend(void *buf, int count, MPI_Datatype type,
 int dest, int tag, MPI_Comm comm, MPI_Request *req);
```

```
int MPI_Irecv (void *buf, int count, MPI_Datatype type,
 int source, int tag, MPI_Comm comm, MPI_Request *req);
```

# Non-Blocking Send and Receive

FORTRAN:

```
MPI_ISEND(buf, count, type, dest, tag, comm, req, ierr)
MPI_IRECV(buf, count, type, source, tag, comm, req, ierr)
```

**buf** send/receive buffer (memory located in sender/receiver memory).  
**count** (INTEGER) number of contiguous elements of **buf** to be sent  
**type** (INTEGER) MPI type of **buf**  
**dest** (INTEGER) rank of the destination process  
**tag** (INTEGER) number identifying the message  
**comm** (INTEGER) communicator of the sender and receiver  
**req** (INTEGER) output, identifier of the communications handle  
**ierr** (INTEGER) output, error code (if **ierr=0** no error occurs)



# Waiting for Completion

**FORTRAN:**

```
MPI_WAIT(req, status, ierr)
MPI_WAITALL (count, array_of_requests, array_of_statuses, ierr)
```

A call to this subroutine causes the code to wait until the communication referred to by req is complete.

**req** (INTEGER): input/output, identifier associated to a communications event (initiated by **MPI\_ISEND** or **MPI\_RECV**).

**Status** (INTEGER): array of size **MPI\_STATUS\_SIZE**, if **req** was associated to a call to **MPI\_RECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

**ierr** (INTEGER): output, error code (if **ierr=0** no error occurs).

**C:**

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
Int MPI_Waitall (count,&array_of_requests,&array_of_statuses) ;
```



# Testing Completion

FORTRAN:

```
MPI_TEST(req, flag, status, ierr)
MPI_TESTALL (count,array_of_requests,flag,array_of_statuses,ierr)
```

A call to this subroutine sets **flag** to **.true.** if the communication pointed by **req** is complete, sets **flag** to **.false.** otherwise.

**Req** (INTEGER) input/output, identifier associated to a communications event (initiated by **MPI\_ISEND** or **MPI\_RECV**).

**Flag** (LOGICAL) output, **.true.** if communication **req** has completed **.false.** otherwise

**Status** (INTEGER)array of size **MPI\_STATUS\_SIZE**, if **req** was associated to a call to **MPI\_RECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

**ierr** (INTEGER) output, error code (if **ierr=0** no error occurs).

C:

```
int MPI_Test (&request,&flag,&status);
Int MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses);
```

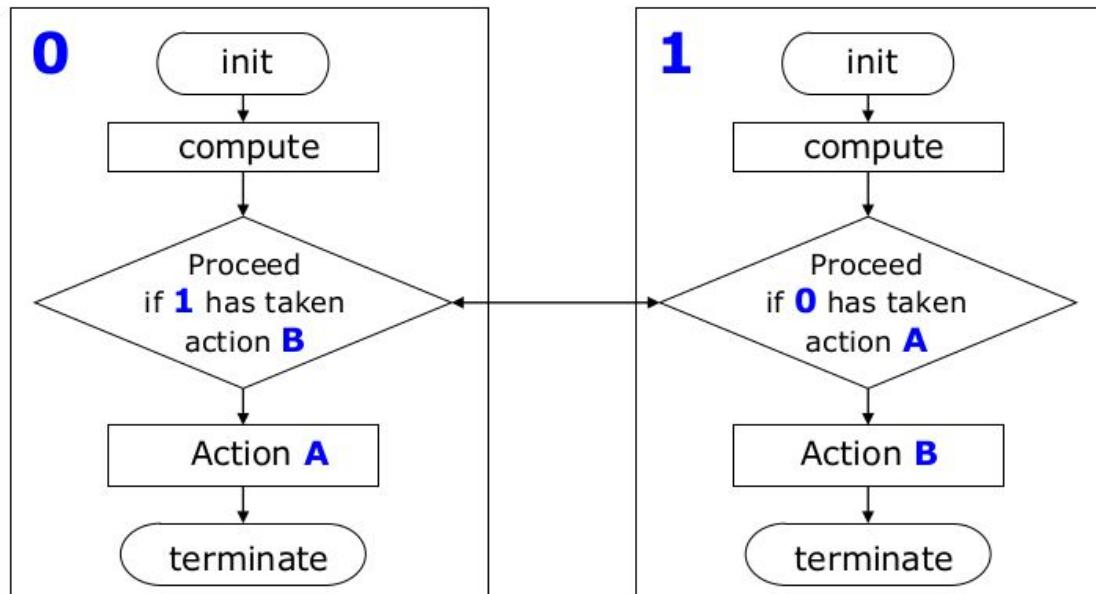


# Wildcards

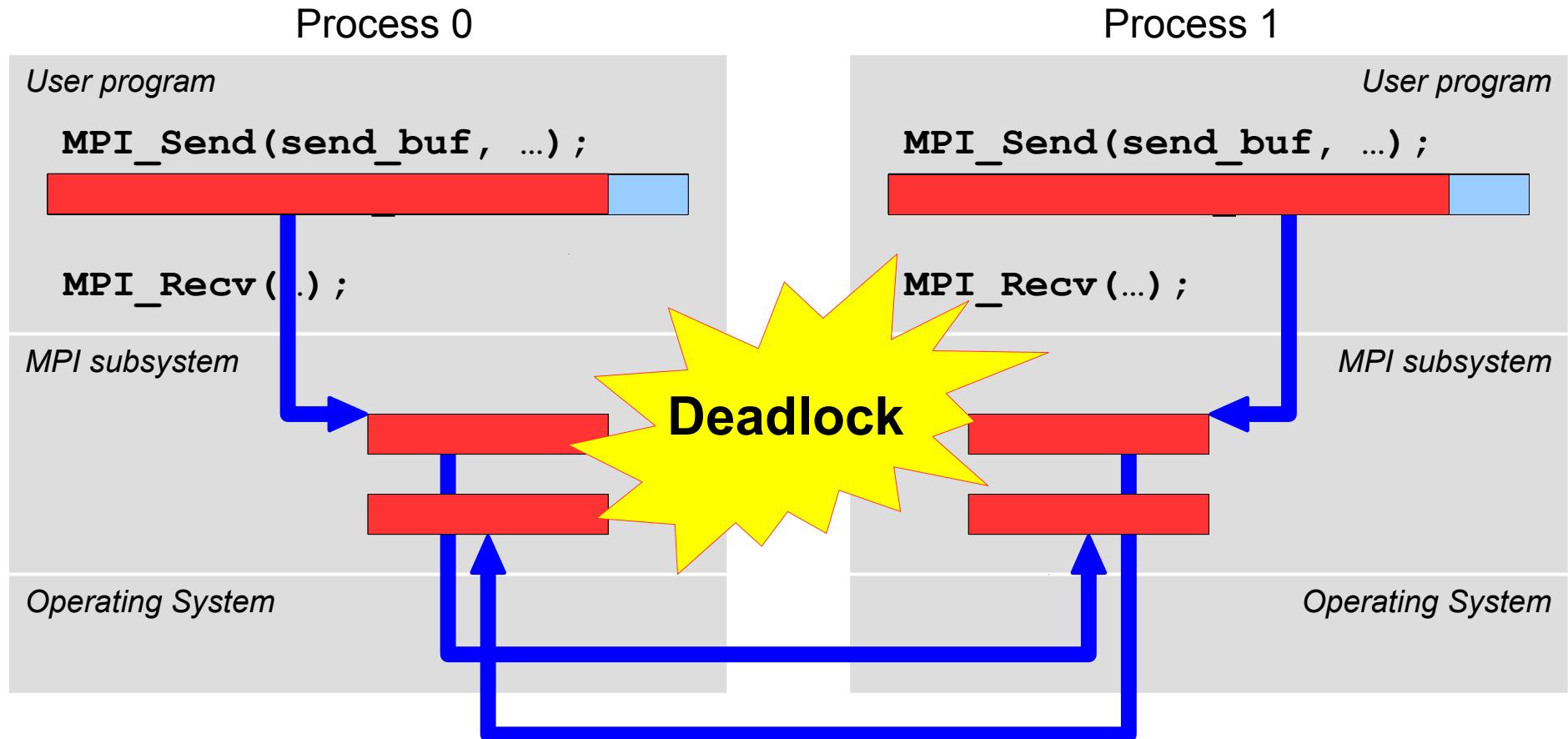
- Both in FORTRAN and C **MPI\_RECV** accepts wildcards:
- To receive from any source: **MPI\_ANY\_SOURCE**
- To receive with any tag: **MPI\_ANY\_TAG**
- Actual source and tag are returned in the receiver's status parameter

# DEADLOCK

A Deadlock occurs when 2 (or more) processes are blocked and each one is waiting for the other to make progress.



# Blocking communication and deadlocks





# Simple DEADLOCK

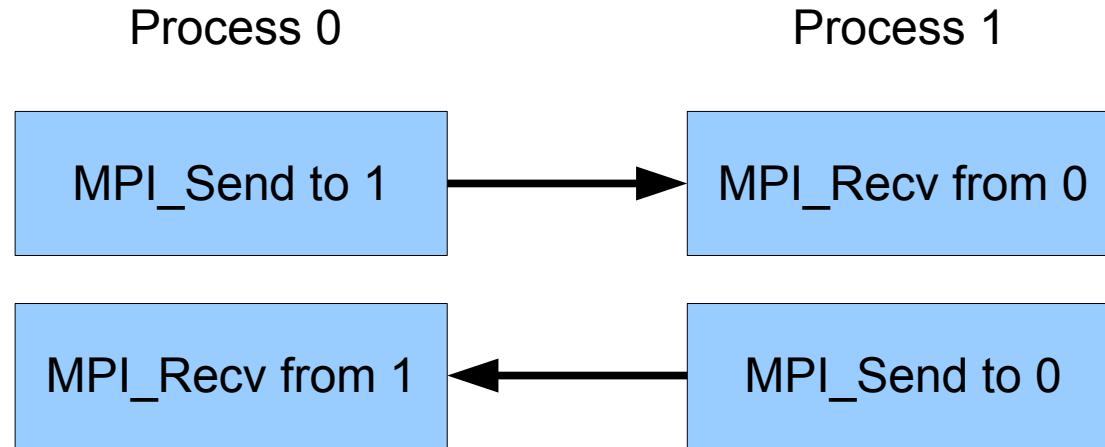
```
PROGRAM deadlock
 USE mpi
 implicit none
 INTEGER ierr, myid, nproc
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2), B(2)

 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

 IF(myid .EQ. 0) THEN
 a(1) = 2.0
 a(2) = 4.0
 CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
 CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
 ELSE IF(myid .EQ. 1) THEN
 a(1) = 3.0
 a(2) = 5.0
 CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
 CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
 END IF
 WRITE(6,*) myid, ': b(1)=' , b(1), ' b(2)=' , b(2)
 CALL MPI_FINALIZE(ierr)
END PROGRAM
```

# Blocking communication and deadlocks

- To avoid the deadlock it is necessary to reorder the operations so that send/receive pairs match...



- ...or use **non-blocking** communication primitives  
... or smart implementations as the Intel one.



# Avoiding DEADLOCK

```
PROGRAM avoid_lock
 USE mpi
 Implicit none
 INTEGER ierr, myid, nproc
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2), B(2)

 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

 IF(myid .EQ. 0) THEN
 a(1) = 2.0
 a(2) = 4.0
 CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
 CALL MPI_SEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, ierr)
 ELSE IF(myid .EQ. 1) THEN
 a(1) = 3.0
 a(2) = 5.0
 CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
 CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
 END IF
 WRITE(6,*) myid, ': b(1)=' , b(1), ' b(2)=' , b(2)
 CALL MPI_FINALIZE(ierr)
END PROGRAM
```



# Avoiding DEADLOCK (2)

```
PROGRAM deadlock
 USE mpi
 implicit none
 INTEGER ierr, myid, nproc, req
 INTEGER status(MPI_STATUS_SIZE)
 REAL A(2), B(2)

 CALL MPI_INIT(ierr)
 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

 IF(myid .EQ. 0) THEN
 a(1) = 2.0
 a(2) = 4.0
 CALL MPI_ISEND(a, 2, MPI_REAL, 1, 10, MPI_COMM_WORLD, req, ierr)
 CALL MPI_RECV(b, 2, MPI_REAL, 1, 11, MPI_COMM_WORLD, status, ierr)
 CALL MPI_WAIT(req, status, ierr)
 ELSE IF(myid .EQ. 1) THEN
 a(1) = 3.0
 a(2) = 5.0
 CALL MPI_SEND(a, 2, MPI_REAL, 0, 11, MPI_COMM_WORLD, ierr)
 CALL MPI_RECV(b, 2, MPI_REAL, 0, 10, MPI_COMM_WORLD, status, ierr)
 END IF
 WRITE(6,*) myid, ': b(1)=' , b(1), ' b(2)=' , b(2)
 CALL MPI_FINALIZE(ierr)
END PROGRAM
```

# SendRecv

- Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.
- The easiest way to send and receive data without worrying about deadlocks

FORTRAN:

```
CALL MPI_SENDRECV(sndbuf, snd_size, snd_type, destid, tag,
 recvbuf, recv_size, recv_type, sourceid, tag, comm, status,
 ierr)
```

**Sender side**

**Receiver side**

A red bracket above the code spans from the first argument 'sndbuf' to the last argument 'status', indicating the portion of the code responsible for the sender's side of the communication. Another red bracket below the code spans from the first argument 'sndbuf' to the last argument 'ierr', indicating the portion of the code responsible for the receiver's side of the communication.

# SendRecv example

```
#include <mpi.h>
#include <stdio.h>

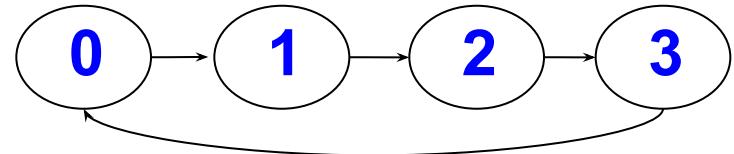
int main(int argc, char *argv[])
{
 int myid, numprocs, left, right;
 int buffer[1], buffer2[1];
 MPI_Status status;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD, &myid);

 right = (myid + 1) % numprocs;
 left = myid - 1;
 if (left < 0)
 left = numprocs - 1;
 buffer[0]=myid;
 MPI_Sendrecv(buffer, 1, MPI_INT, right, 123, buffer2, 1, MPI_INT, left, 123,
 MPI_COMM_WORLD, &status);

 printf("I am rank %d and I have received %d\n",myid,buffer2[0]);
 MPI_Finalize();

 return 0;
}
```



Useful for cyclic communication patterns



# SEND and RECV variants

| Mode             | Completion Condition                                                                              | Blocking subroutine | Non-blocking subroutine |
|------------------|---------------------------------------------------------------------------------------------------|---------------------|-------------------------|
| Standard send    | <b>Message sent (receive state unknown)</b>                                                       | <b>MPI_SEND</b>     | <b>MPI_ISEND</b>        |
| receive          | <b>Completes when a matching message has arrived</b>                                              | <b>MPI_RECV</b>     | <b>MPI_IRecv</b>        |
| Synchronous send | <b>Only completes after a matching recv() is posted and the receive operation is started.</b>     | <b>MPI_SSEND</b>    | <b>MPI_ISSEND</b>       |
| Buffered send    | <b>Always completes, irrespective of receiver</b><br><b>Guarantees the message being buffered</b> | <b>MPI_BSEND</b>    | <b>MPI_IBSEND</b>       |
| Ready send       | <b>Always completes, irrespective of whether the receive has completed</b>                        | <b>MPI_RSEND</b>    | <b>MPI_IRSEND</b>       |

# Final Comments

- MPI is a standard for message-passing and has numerous implementations (OpenMPI, IntelMPI, MPICH, etc)
- MPI uses send and receive calls to manage communications between two processes (point-to-point)
- The calls can be blocking or non-blocking.
- Non-blocking calls can be used to overlap communication with computation (or communication) but wait routines are needed for synchronization.
- Deadlock is a common error and is due to incorrect order of send/receive

Nicola Spallanzani  
n.spallanzani@cineca.it  
High Performance Computing department

# Parallel programming with MPI

Collective Communications





# Collective communications

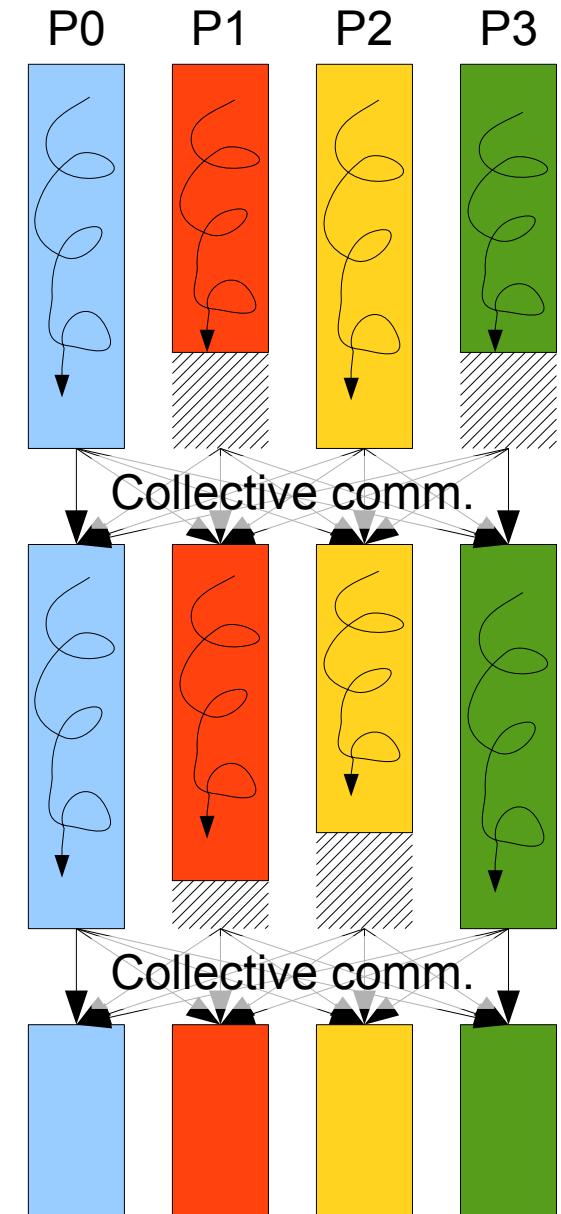
Collective communications is a method of communication which involves all processes in a communicator:

- All processes (in a communicator) call the collective function
- Collective communications will not interfere with point-to-point
- All collective communications are blocking (in MPI 2.0)
- No tags are required
- Receive buffers must match in size (number of bytes)

It's a safe communication mode

# Collective communications

- Many applications use the *bulk synchronous* pattern:
  - Repeat:
    - Local computation
    - Communicate to update global view on all processes
- Collective communications are executed by all processes in the group to compute and share some global result



# Collective communications

- Collective communications are assumed to be more efficient than point-to-point operations achieving the same result
- Understanding when collective communications are to be used is an **essential skill** of a MPI programmer



# Collective communications

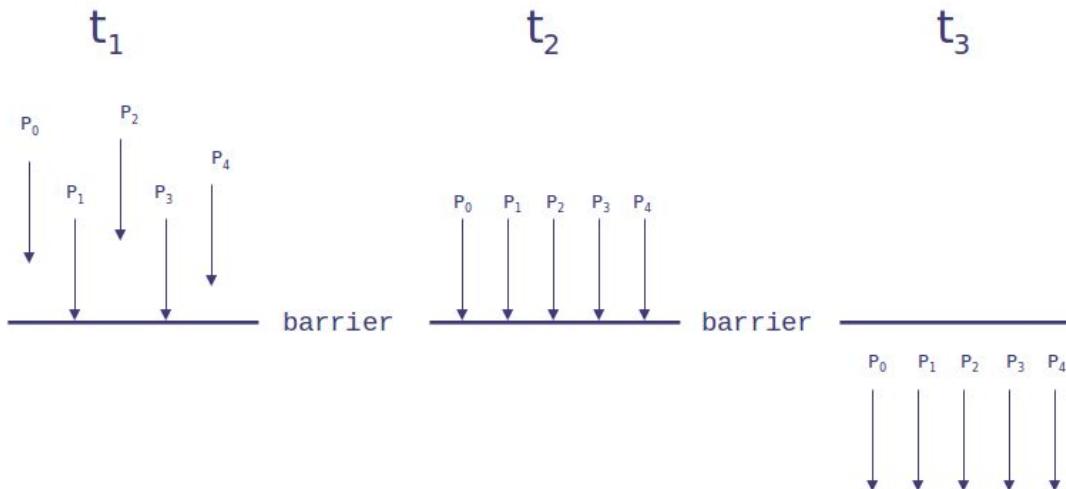
Communications involving a group of processes. They are called by all the ranks involved in a communicator (or a group) and are of three types:

- Synchronization (e.g. Barrier)
- Data Movement (e.g. Broadcast or Gather/scatter)
- Global Computation (e.g. reductions)

# MPI Barrier

It stops all processes within a communicator until they are synchronized

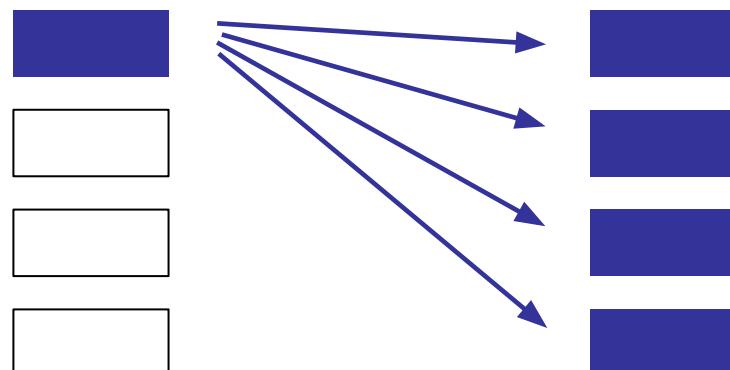
```
int MPI_Barrier(MPI_Comm comm);
```



# MPI Broadcast

```
int MPI_Bcast (void *buf, int count, MPI_Datatype datatype,
 int root, MPI_Comm comm);
```

Note that all processes must specify the same root and same comm.



# **MPI\_Bcast()**

Broadcasts a message to all other processes of a group

```
count = 3;
src = 1; /* broadcast originates from process 1 */
MPI_Bcast(buf, count, MPI_INT, src, MPI_COMM_WORLD);
```

Proc 0

Proc 1

Proc 2

Proc 3



buf[] (before)



buf[] (after)

# Example

```
PROGRAM broad_cast
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
INTEGER status(MPI_STATUS_SIZE)
REAL A(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
IF(myid .EQ. root) THEN
 a(1) = 2.0
 a(2) = 4.0
END IF
CALL MPI_BCAST(a, 2, MPI_REAL, root, MPI_COMM_WORLD, ierr)
WRITE(6,*) myid, ': a(1)=' , a(1), 'a(2)=' , a(2)
CALL MPI_FINALIZE(ierr)
END PROGRAM broad_cast
```

# MPI Gather

Each process, root included, sends the content of its send buffer to the root process. The root process receives the messages and stores them in the rank order.

```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
 int root, MPI_Comm comm);
```



# **MPI\_Gather()**

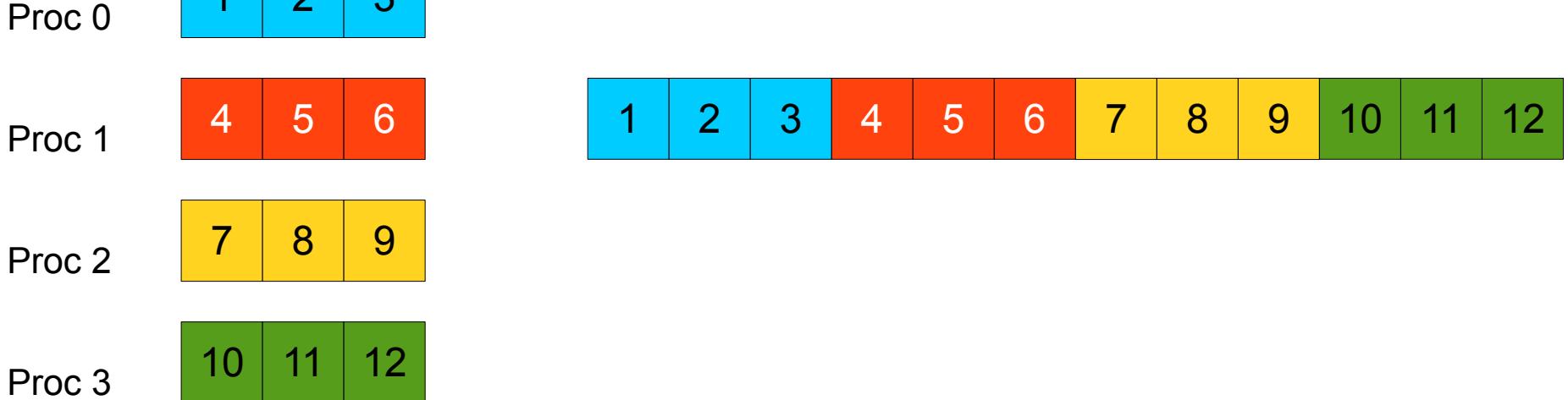
Gathers together data from other processes

```
sendcnt = 3; /* how many items are sent by each process */
recvcnt = 3; /* how many items are received from each process */
dst = 1; /* message will be gathered at process 1 */
MPI_Gather(sendbuf, sendcnt, MPI_INT,
 recvbuf, recvcnt, MPI_INT, dst, MPI_COMM_WORLD);
```

sendbuf[] (before)



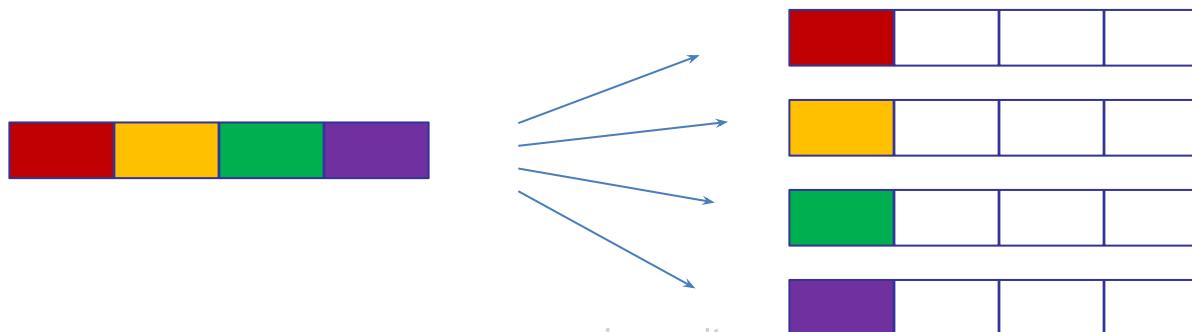
recvbuf[] (after)



# MPI Scatter

The root sends a message. The message is split into n equal segments, the i-th segment is sent to the i-th process in the group and each process receives this message.

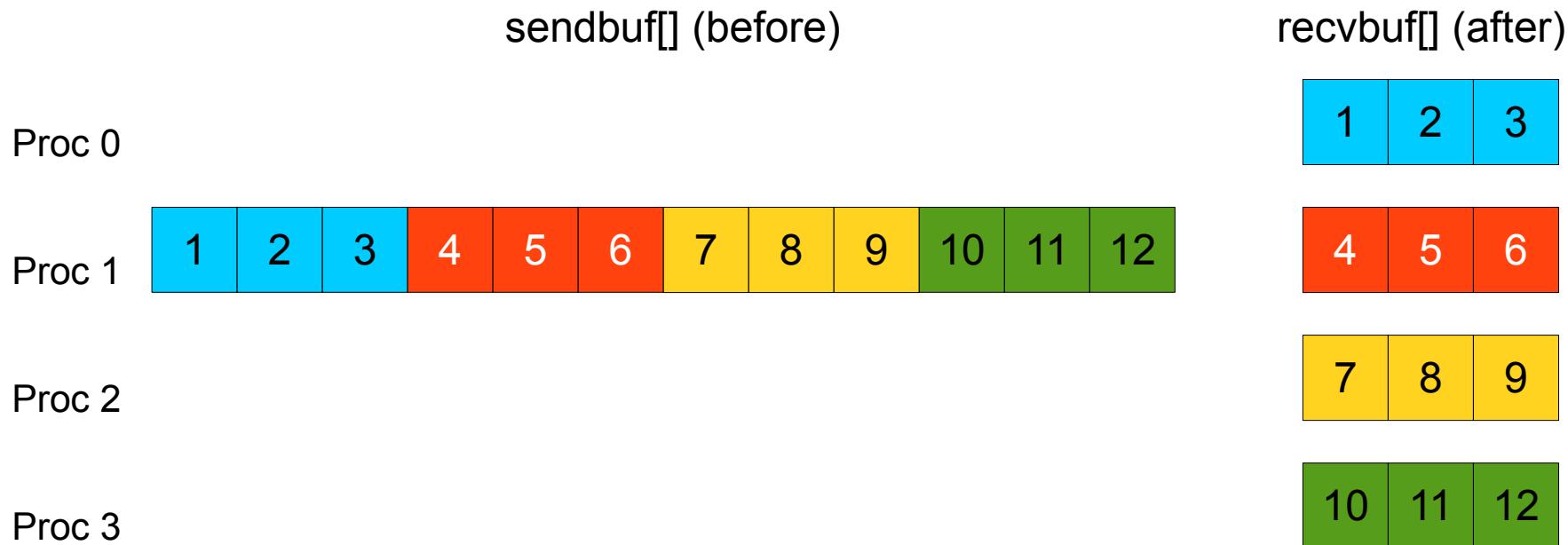
```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
 int root, MPI_Comm comm);
```



# **MPI\_Scatter()**

Distribute data to other processes in a group

```
sendcnt = 3; /* how many items are sent to each process */
recvcnt = 3; /* how many items are received by each process */
src = 1; /* process 1 contains the message to be scattered */
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
 recvbuf, recvcnt, MPI_INT, src, MPI_COMM_WORLD);
```



# **MPI\_Scatter()**

- The **MPI\_Scatter()** operation produce the same result as if the root executes a series of

```
MPI_Send(sendbuf + i * sendcount * extent(sendtype),
sendcount, sendtype, i, ...)
```

and all other processes execute

```
MPI_Recv(recvbuf, recvcount, recvtype, i, ...)
```

# MPI Allgather

There are possible combinations of collective functions.

For example, MPI Allgather is a combination of a gather + a broadcast:

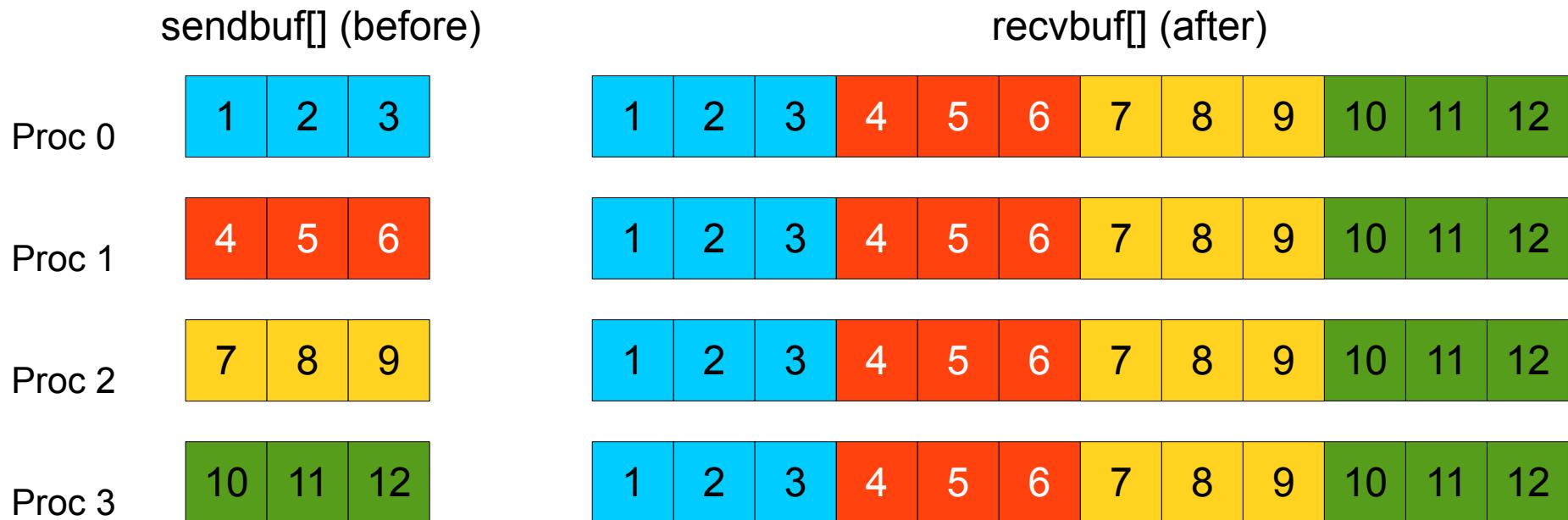
```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 MPI_Comm comm);
```



# **MPI\_Allgather()**

Gathers data from other processes and distribute to all

```
sendcnt = 3;
recvcnt = 3;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
 recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);
```

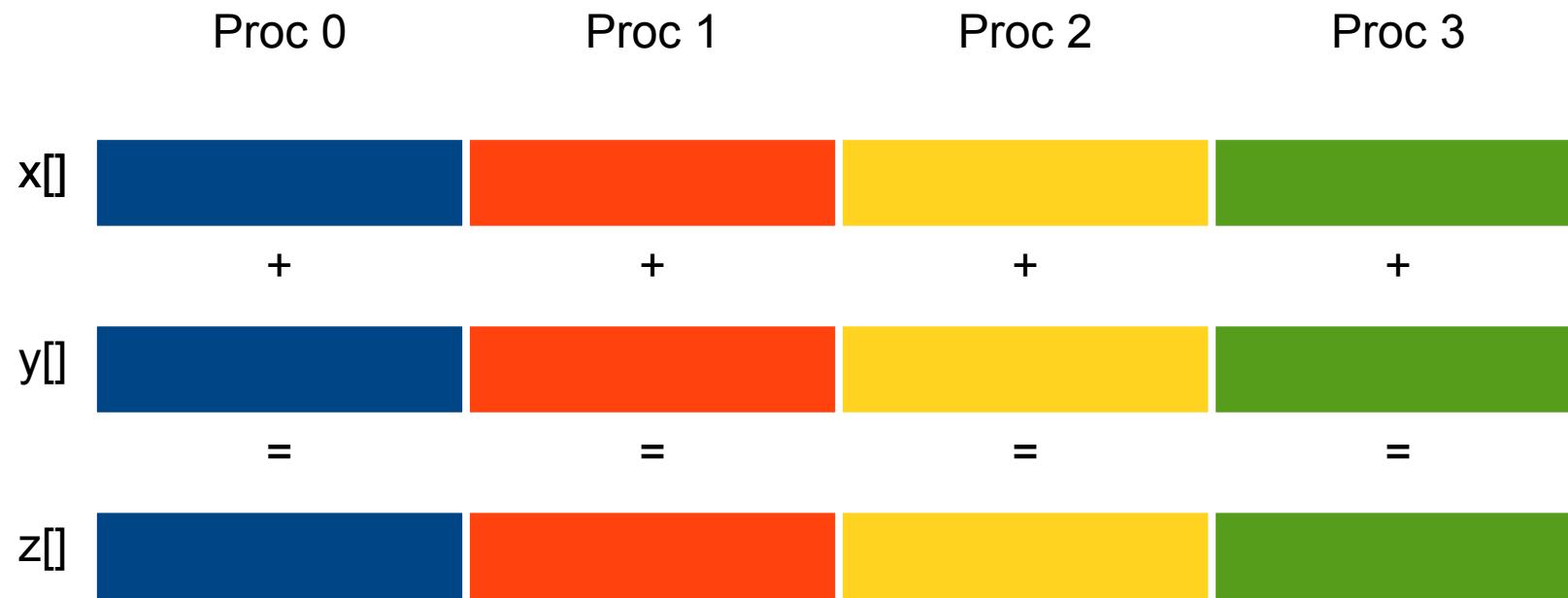


# Example: Parallel Vector Sum

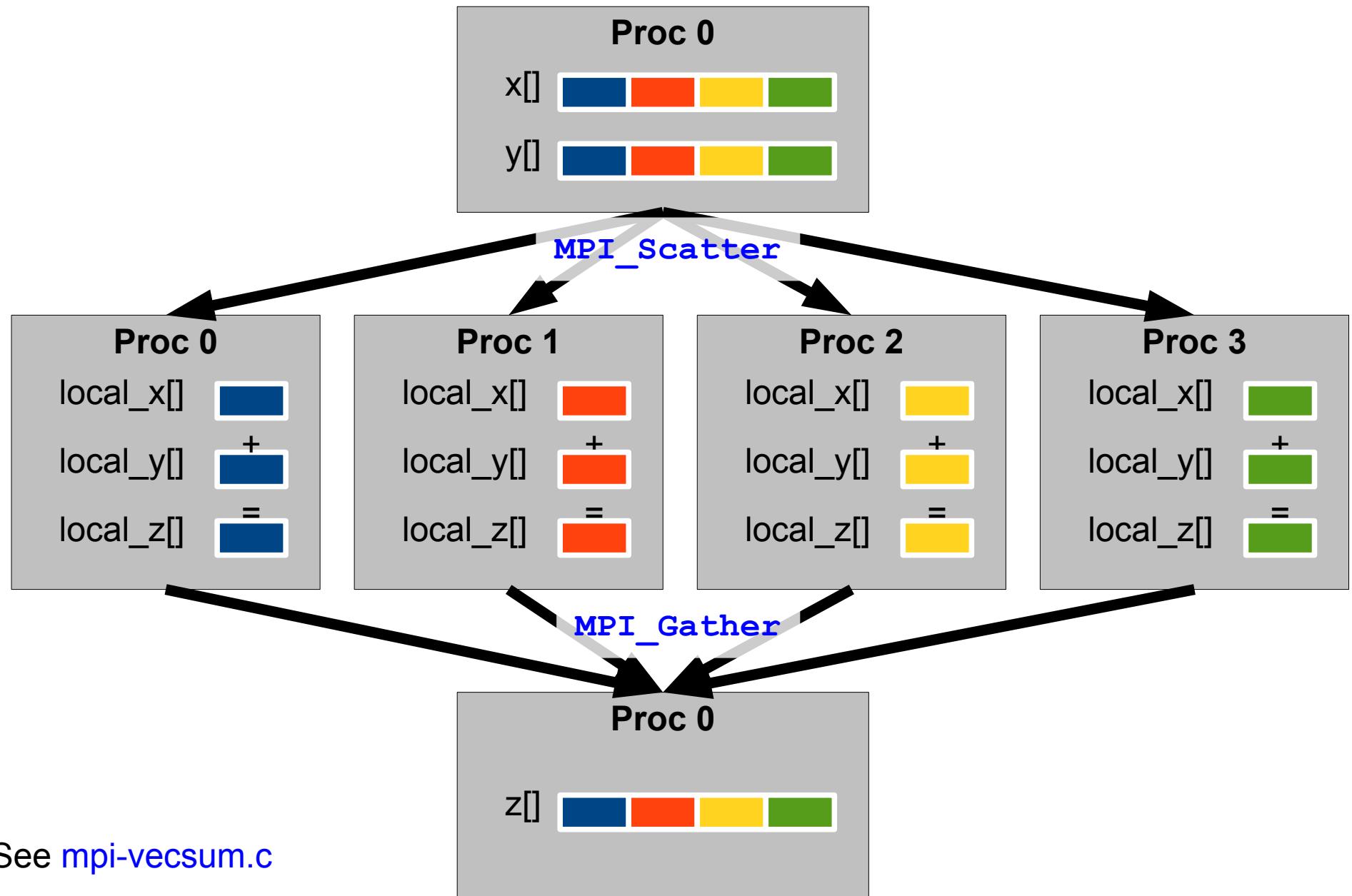
$$\begin{aligned}x + y &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\&= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\&= (z_0, z_1, \dots, z_{n-1}) \\&= z\end{aligned}$$

```
void sum(double* x, double* y, double* z, int n)
{
 int i;
 for (i=0; i<n; i++) {
 z[i] = x[i] + y[i];
 }
}
```

# Parallel Vector Sum



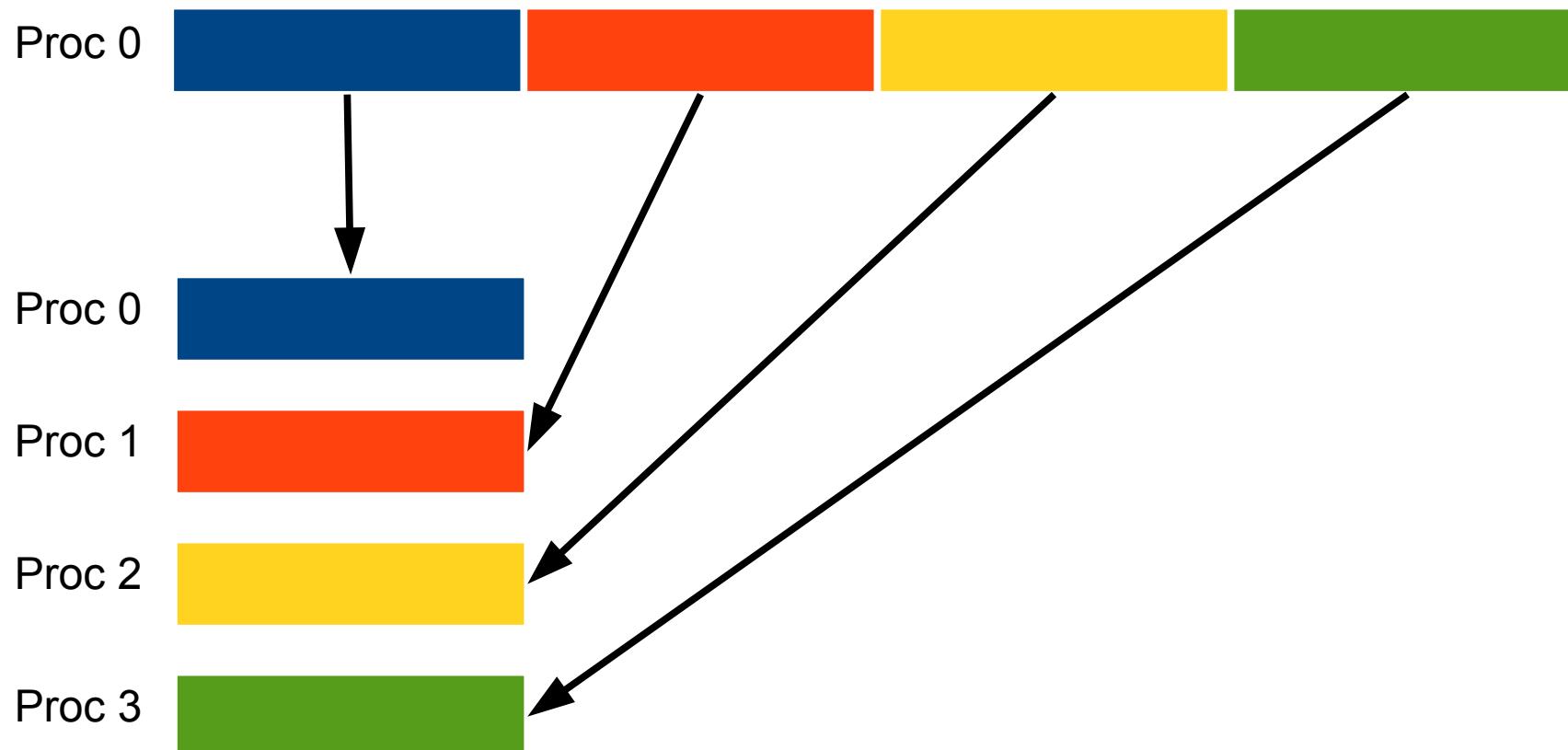
# Parallel Vector Sum



See [mpi-vecsum.c](#)

# **MPI\_Scatter()**

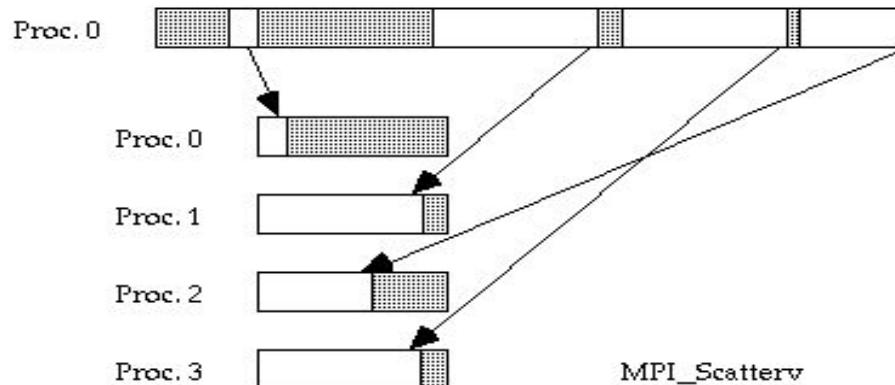
- Contiguous data
- Uniform message size



# MPI\_Scatterv, MPI\_Gatherv

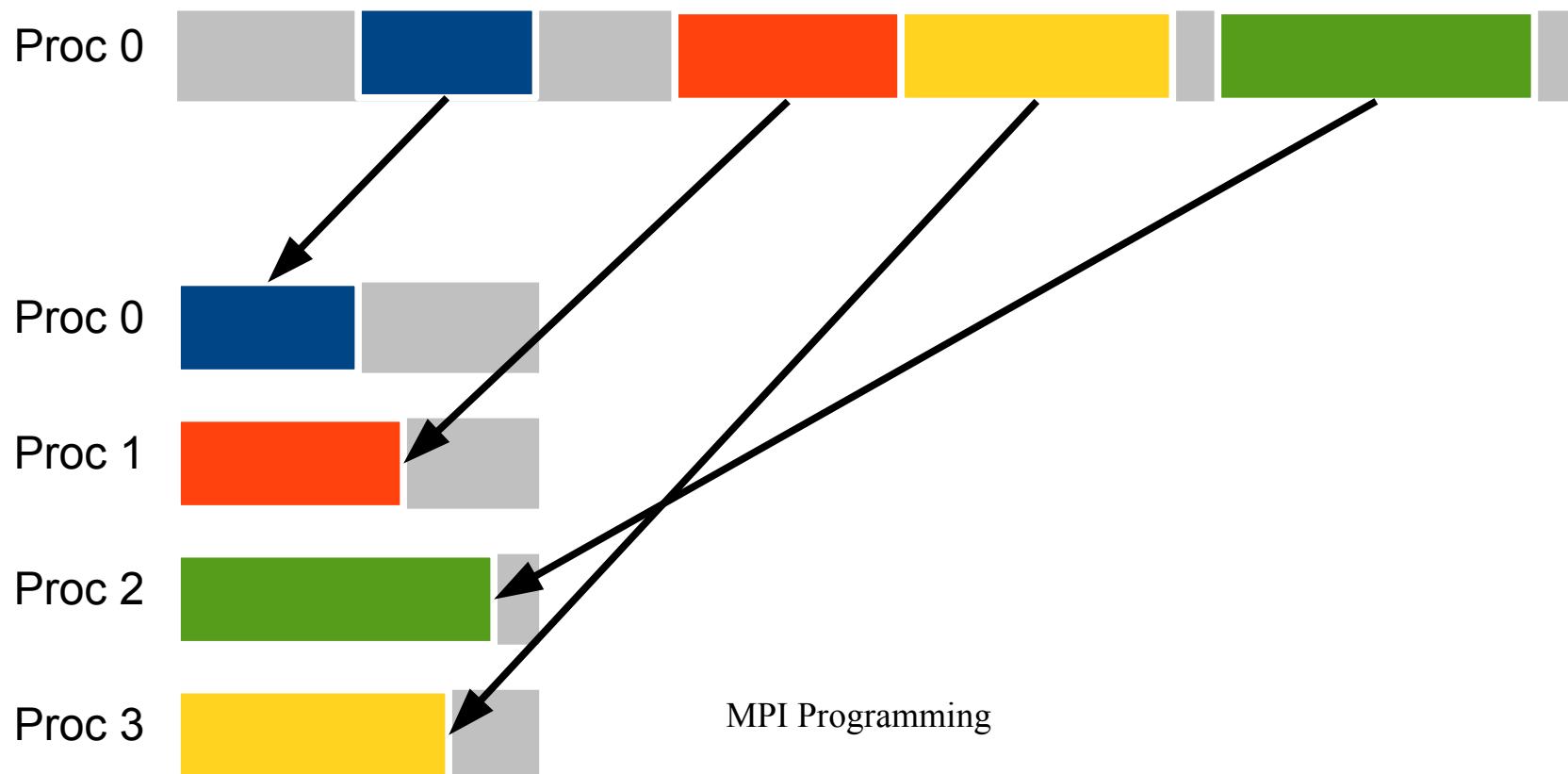
For many collective functions there are extended functionalities.

For example it's possible to define the length of arrays to be scattered or gathered with `MPI_Scatterv` and `MPI_Gatherv`.

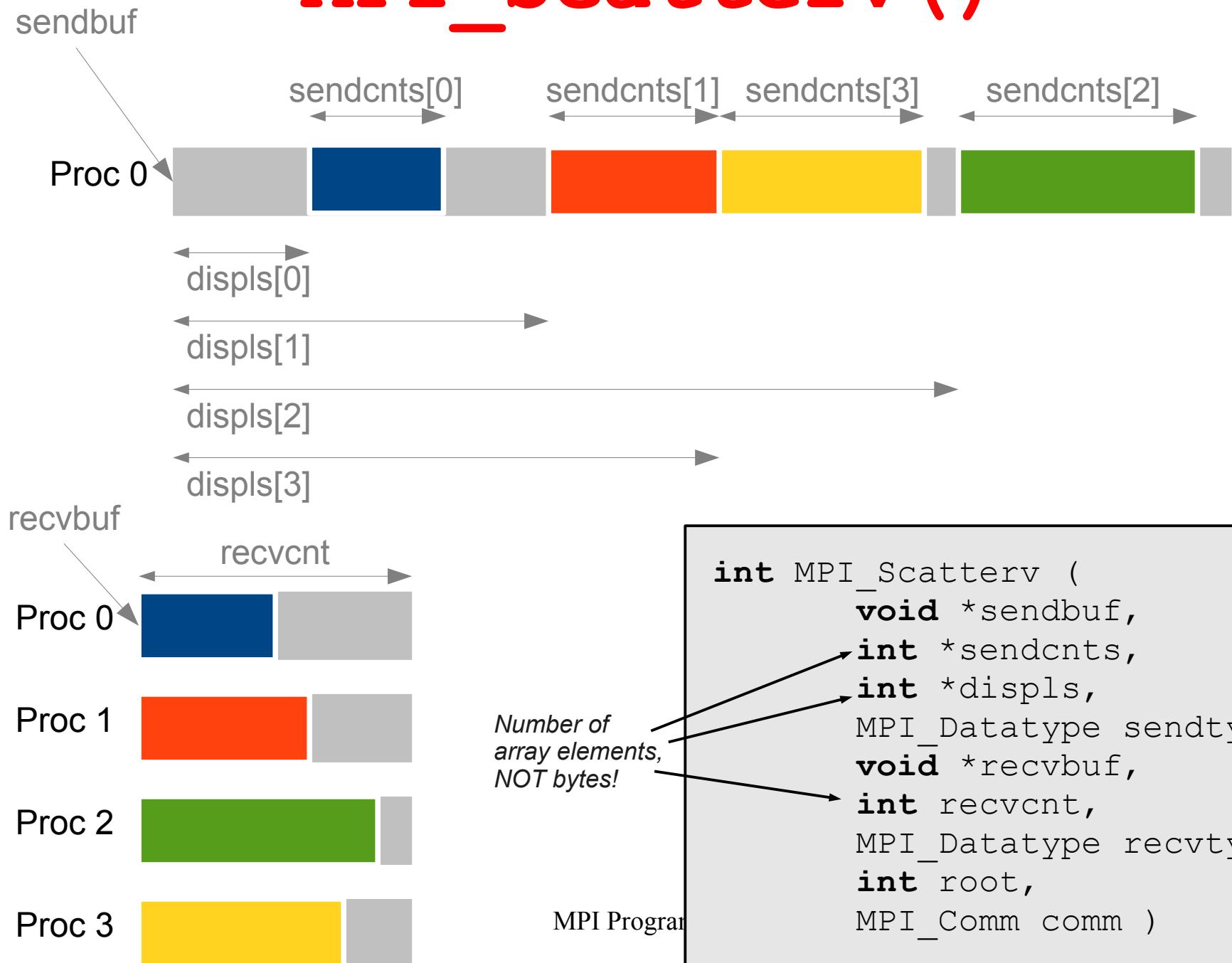


# MPI\_Scatterv() / MPI\_Gatherv()

- Gaps are allowed between messages in source data
- Irregular message sizes are allowed
- Data can be distributed to processes in any order



# **MPI\_Scatterv()**





# MPI Scaterv

C:

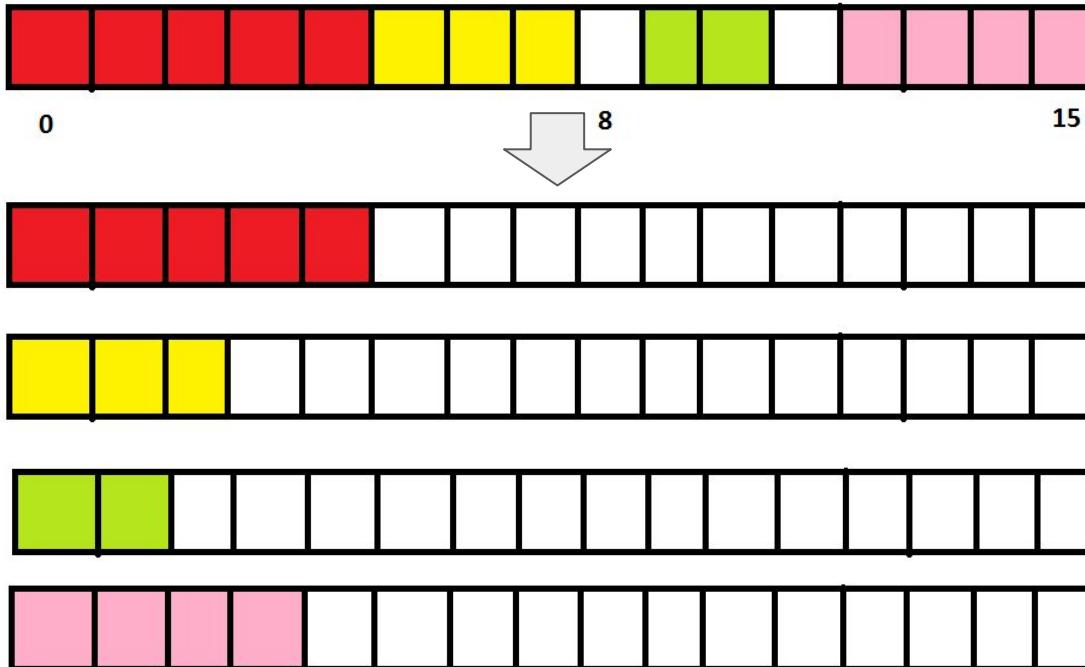
```
int MPI_Scaterv(const void *sendbuf, const int sendcounts[],
 const int displs[], MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

Fortran:

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,
 RECVBUF, REVCOUNT, RCVTYPE,
 ROOT, COMM, IERROR)
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE
INTEGER REVCOUNT, RCVTYPE, ROOT, COMM, IERROR
```

# MPI Scaterv

sendcounts=(5,3,2,4) displs=(0,5,9,12)



# Example

```
int sendbuf[] = {10, 11, 12, 13, 14, 15, 16}; /* at master */
int displs[] = {3, 0, 1}; /* assume P=3 MPI processes */
int sendcnts[] = {3, 1, 4};
int recvbuf[5];
...
MPI_Scatterv(sendbuf, sendcnts, displs, MPI_INT, recvbuf, 5,
MPI_INT, 0, MPI_COMM_WORLD);
```





# MPI Gatherv

C:

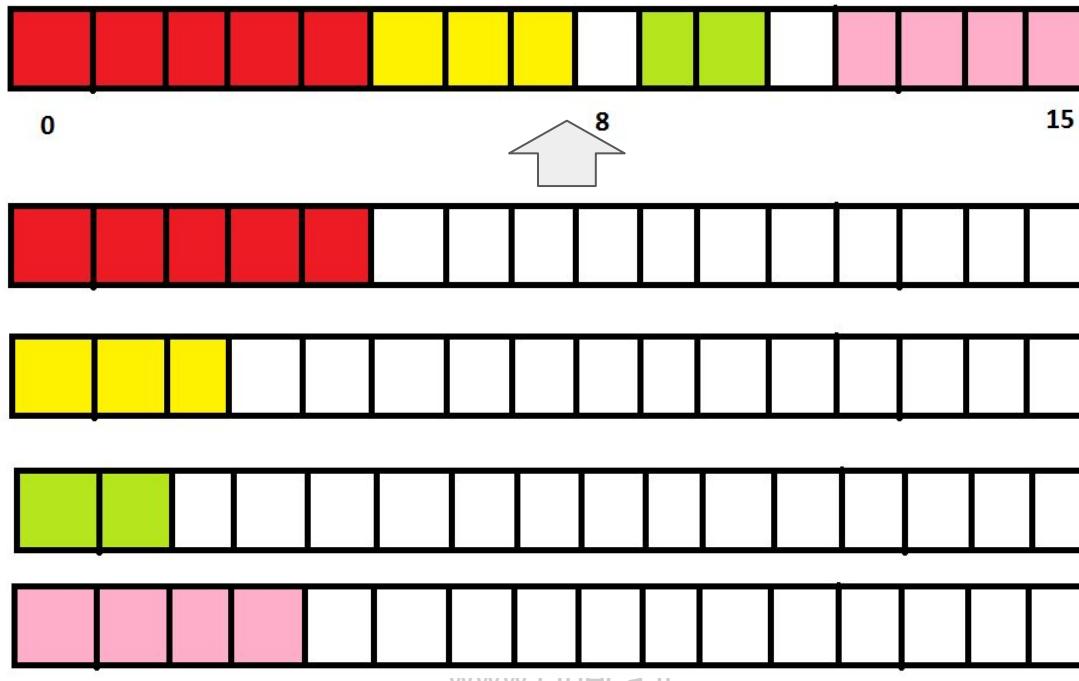
```
int MPI_Gatherv(const void *sendbuf, int sendcount,
 MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
 const int displs[], MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

Fortran:

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE,
 RECVBUF, REVCOUNTS, DISPLS, RECVTYPE,
 ROOT, COMM, IERROR)
INTEGER SENDCOUNT, SENDTYPE, REVCOUNTS(*), DISPLS(*)
INTEGER RECVTYPE, ROOT, COMM, IERROR
```

# MPI Gatherv

recvcounts=(5,3,2,4) displs=(0,5,9,12)



# MPI All to All

This function makes a redistribution of the content of each process in a way that each process know the buffer of all others. It is a way to implement the matrix data transposition.

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 MPI_Comm comm)
```

|    |    |    |    |
|----|----|----|----|
| a1 | a2 | a3 | a4 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| b1 | b2 | b3 | b4 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| c1 | c2 | c3 | c4 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| d1 | d2 | d3 | d4 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| a1 | b1 | c1 | d1 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| a2 | b2 | c2 | d2 |
|----|----|----|----|

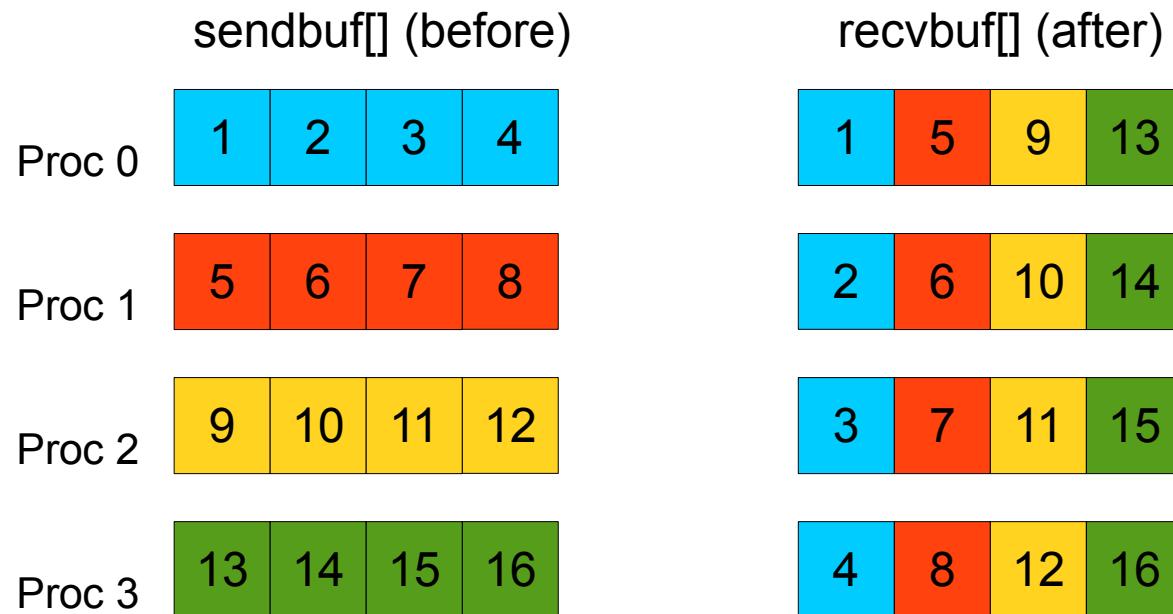
|    |    |    |    |
|----|----|----|----|
| a3 | b3 | c3 | d3 |
|----|----|----|----|

|    |    |    |    |
|----|----|----|----|
| a4 | b4 | c4 | d4 |
|----|----|----|----|

# **MPI\_Alltoall()**

Each process performs a scatter operation

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
 recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);
```



# Reduction

Reduction operations permits us to

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root process (MPI\_Reduce) or
- Store the result on all processes (MPI\_Allreduce)

C:

```
int MPI_Reduce(const void *sendbuf,
 void *recvbuf, int count,
 MPI_Datatype datatype,
 MPI_Op op, int root,
 MPI_Comm comm);
```

## Predefined reduction operations

| MPI op     | Function             |
|------------|----------------------|
| MPI_MAX    | Maximum              |
| MPI_MIN    | Minimum              |
| MPI_SUM    | Sum                  |
| MPI_PROD   | Product              |
| MPI_LAND   | Logical AND          |
| MPI_BAND   | Bitwise AND          |
| MPI_LOR    | Logical OR           |
| MPI_BOR    | Bitwise OR           |
| MPI_LXOR   | Logical exclusive OR |
| MPI_BXOR   | Bitwise exclusive OR |
| MPI_MAXLOC | Maximum and location |
| MPI_MINLOC | Minimum and location |

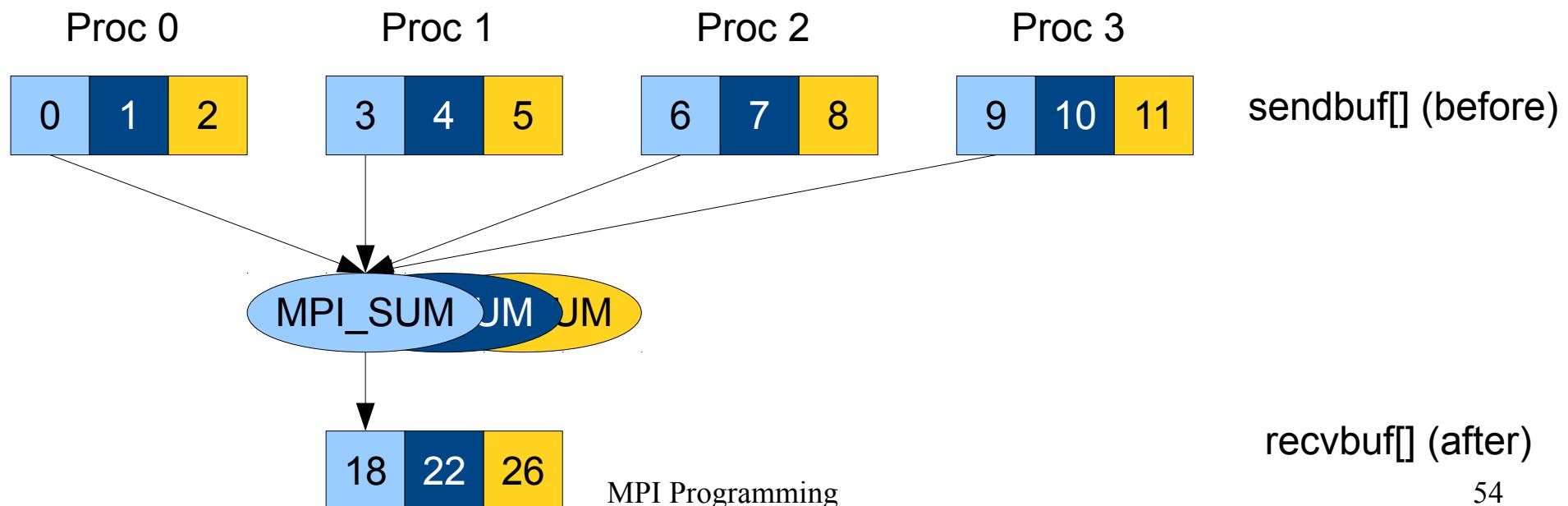
# Example

```
PROGRAM reduce
INCLUDE 'mpif.h'
INTEGER ierr, myid, nproc, root
REAL A(2), res(2)
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
root = 0
A(1) = 2.0 * myid
A(2) = 4.0 * myid
CALL MPI_REDUCE(A, res, 2, MPI_REAL, MPI_SUM, root, MPI_COMM_WORLD, ierr)
IF(myid .EQ. 0) THEN
 WRITE(6,*) myid, ': res(1)=' , res(1), 'res(2)=' , res(2)
END IF
CALL MPI_FINALIZE(ierr)
END
```

# **MPI\_Reduce()**

- If count > 1, **recvbuf[i]** is the reduction of all elements **sendbuf[i]** at the various processes

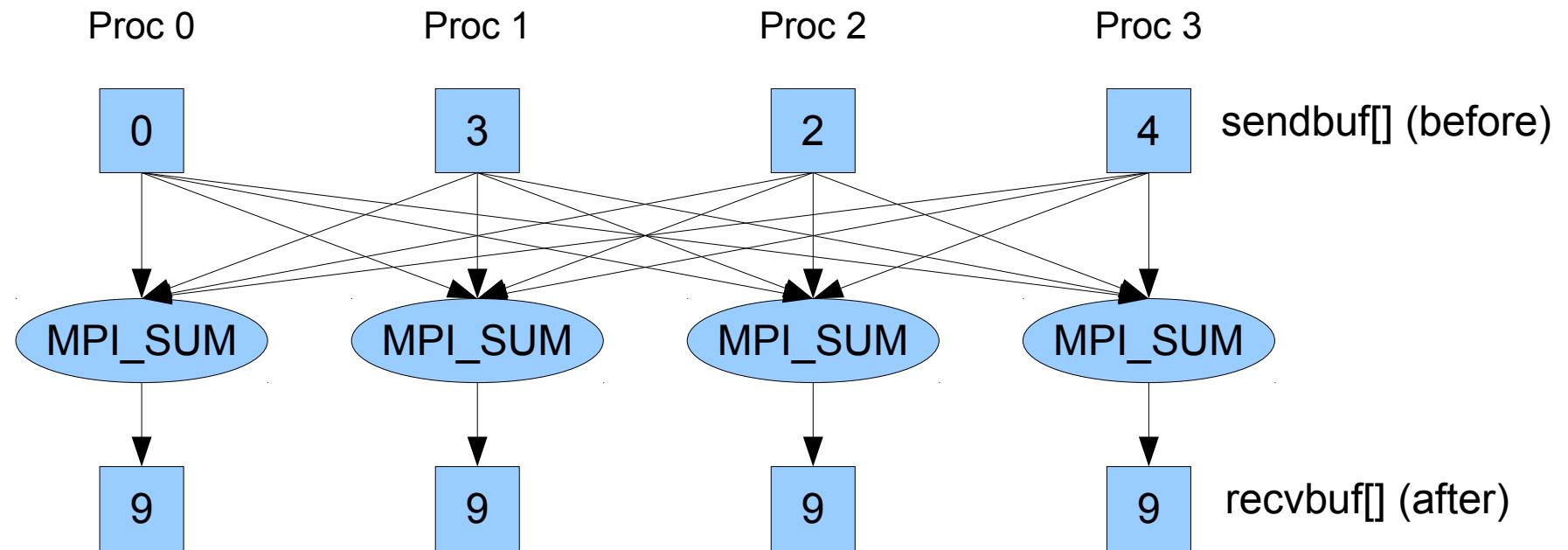
```
count = 3;
dst = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dst, MPI_COMM_WORLD);
```



# **MPI\_Allreduce()**

Performs a reduction and place result in all processes

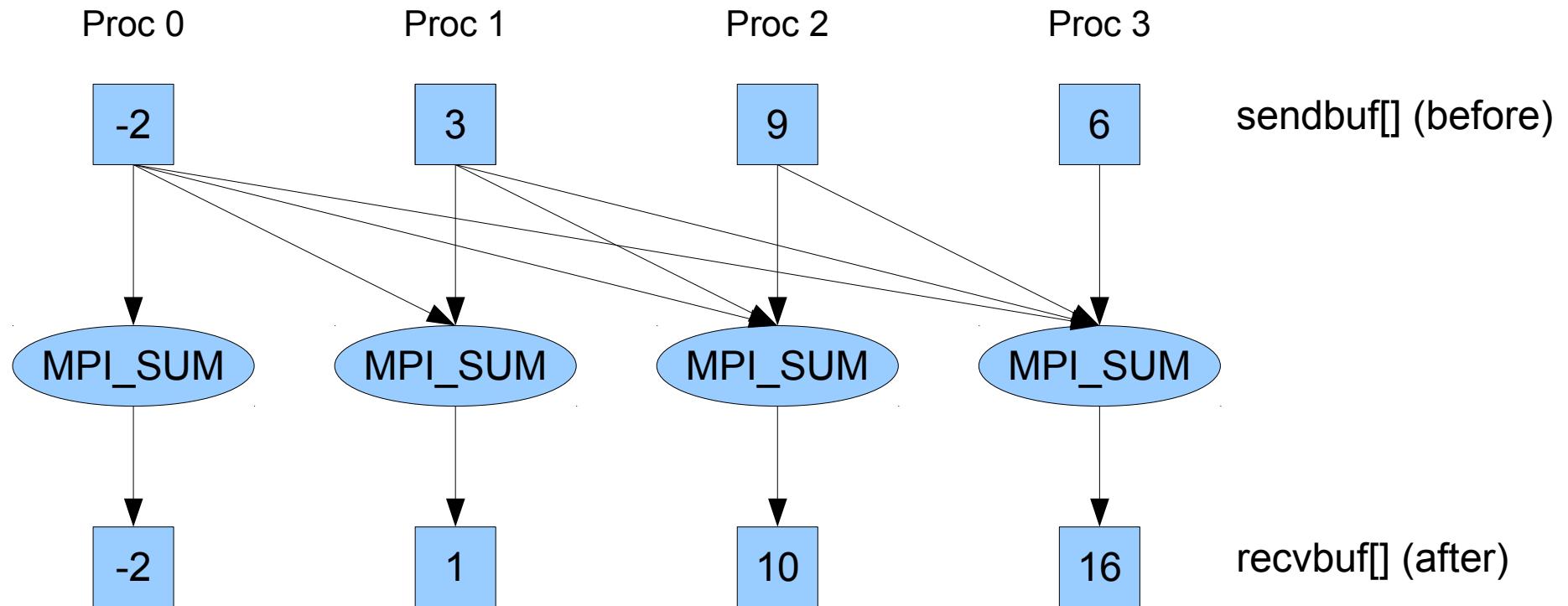
```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



# MPI\_Scan()

## Compute the inclusive scan

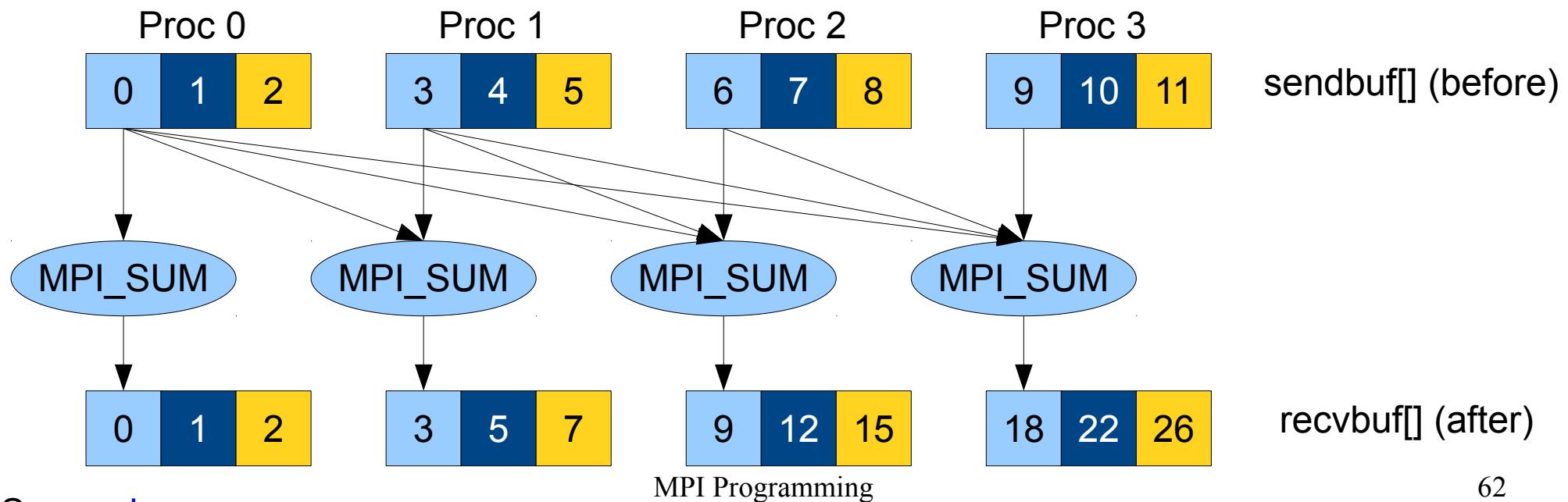
```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



# MPI\_Scan()

- If count > 1, **recvbuf[i]** at proc. *j* is the scan of all elements **sendbuf[i]** at the first *j* processes (incl.)

```
count = 3;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



See [mpi-scan.c](#)

# Collective Communication Routines / 1

## **MPI\_Barrier(comm)**

- Synchronization operation. Creates a barrier synchronization in a group. Each process, when reaching the MPI\_Barrier call, blocks until all processes in the group reach the same MPI\_Barrier call. Then all processes can continue.

## **MPI\_Bcast(&buffer, count, datatype, root, comm)**

- Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

## **MPI\_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)**

- Distributes distinct messages from a single source process to each process in the group

## **MPI\_Gather(&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)**

- Gathers distinct messages from each process in the group to a single destination process. This routine is the reverse operation of MPI\_Scatter

## **MPI\_Allgather(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)**

- Concatenation of data to all processes in a group. Each process in the group, in effect, performs a one-to-all broadcasting operation within the group

## **MPI\_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)**

- Applies a reduction operation on all processes in the group and places the result in one process

# Collective Communication Routines / 2

## `MPI_Reduce_scatter(&sendbuf, &recvbuf, recvcount, datatype, op, comm)`

- Collective computation operation + data movement. First does an element-wise reduction on a vector across all processes in the group. Next, the result vector is split into disjoint segments and distributed across the processes. This is equivalent to an `MPI_Reduce` followed by an `MPI_Scatter` operation.

## `MPI_Scan(&sendbuf, &recvbuf, count, datatype, op, comm)`

- Performs a scan with respect to a reduction operation across a process group.

## `MPI_Alltoall(&sendbuf, sendcount, sendtype, &recvbuf, recvcnt, recvtype, comm)`

- Data movement operation. Each process in a group performs a scatter operation, sending a distinct message to all the processes in the group in order by index.



# Performance issues

- Much hidden communication takes place with collective communication.
- Hardware vendors work hard to provide optimized collective calls but performances will vary according to implementation.
- Because of forced synchronization, collective communications may not always be the best solution.

Some studies show that around 80% transfer time is in collectives.

# A Brief Word on MPI-2 and MPI-3

Intentionally, the MPI-1 specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2 in 1998. Key areas of new functionality in MPI-2:

- **Dynamic Processes** - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
- **One-Sided Communications** - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
- **Extended Collective Operations** - allows for the application of collective operations to inter-communicators
- **External Interfaces** - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.
- **Additional Language Bindings** - describes C++ bindings and discusses Fortran-90 issues.
- **Parallel I/O** - describes MPI support for parallel I/O.

# A Brief Word on MPI-2 and MPI-3

The MPI-3 standard was adopted in 2012, and contains significant extensions to MPI-1 and MPI-2 functionality including:

- **Nonblocking Collective Operations** - permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
- **New One-sided Communication Operations** .
- **Neighborhood Collectives** - extends the distributed graph and Cartesian process topologies with additional communication power.
- **MPIT Tool Interface** - allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools).
- **Matched Probe** - fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.
- Presently efforts are devoted to a new standard, MPI-4  
<https://www mpi-forum.org/mpi-40/>

Some advanced topics

Alessandro Marani

a.marani@cineca.it

High Performance Computing department

# Parallel programming with MPI

Derived Datatypes



# Derived Data Types

What are they?

- Data types built from the basic MPI datatypes. Formally, the MPI Standard defines a general datatype as an object that specifies two things:
  - a sequence of basic datatypes
  - a sequence of integer (byte) displacements
- An easy way to represent such an object is as a sequence of pairs of basic datatypes and displacements. MPI calls this sequence a typemap.

```
typemap = { (type 0, displ 0), ... (type n-1, displ n-1) }
```

- But for most situations you do not need to worry about the typemap.
- Example: `typemap (mystruct) = { (char,0),(int,8),(double,16) }`



# Derived Data Types

Why use them?

- Sometimes more convenient and efficient. For example, you may need to send messages that contain
  - non-contiguous data of a single type (e.g. a sub-block of a matrix)
  - contiguous data of mixed types (e.g., an integer count, followed by a sequence of real numbers)
  - non-contiguous data of mixed types.

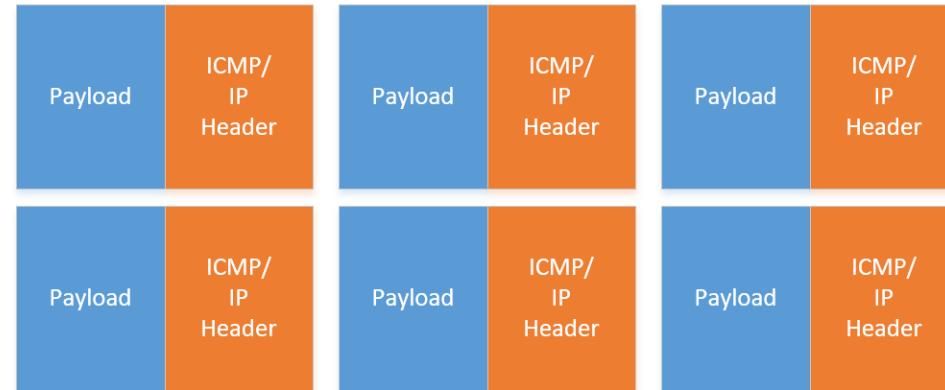
As well as improving program readability and portability they may improve performance.

# Jumbo frames

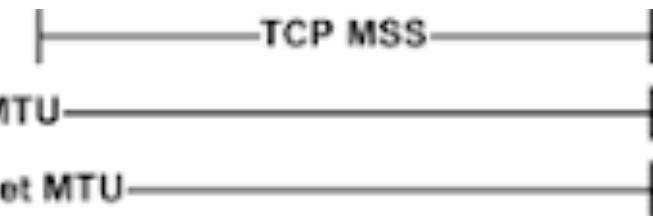
Sending a message costs  $T(n) = \alpha + \beta n$

A Ethernet frame has a transmission unit (MTU) of 1526 byte, with a payload of up to 1472 byte. A jumbo frame, if supported, up to 9000, i.e. it can contain up to 6 messages.

If you merge several payload you can limit the overhead due to  $\alpha$ , the latency, from  $6 * \alpha$  to  $\alpha$ .  
Easy to achieve for sending homogeneous arrays, not for heterogeneous dat

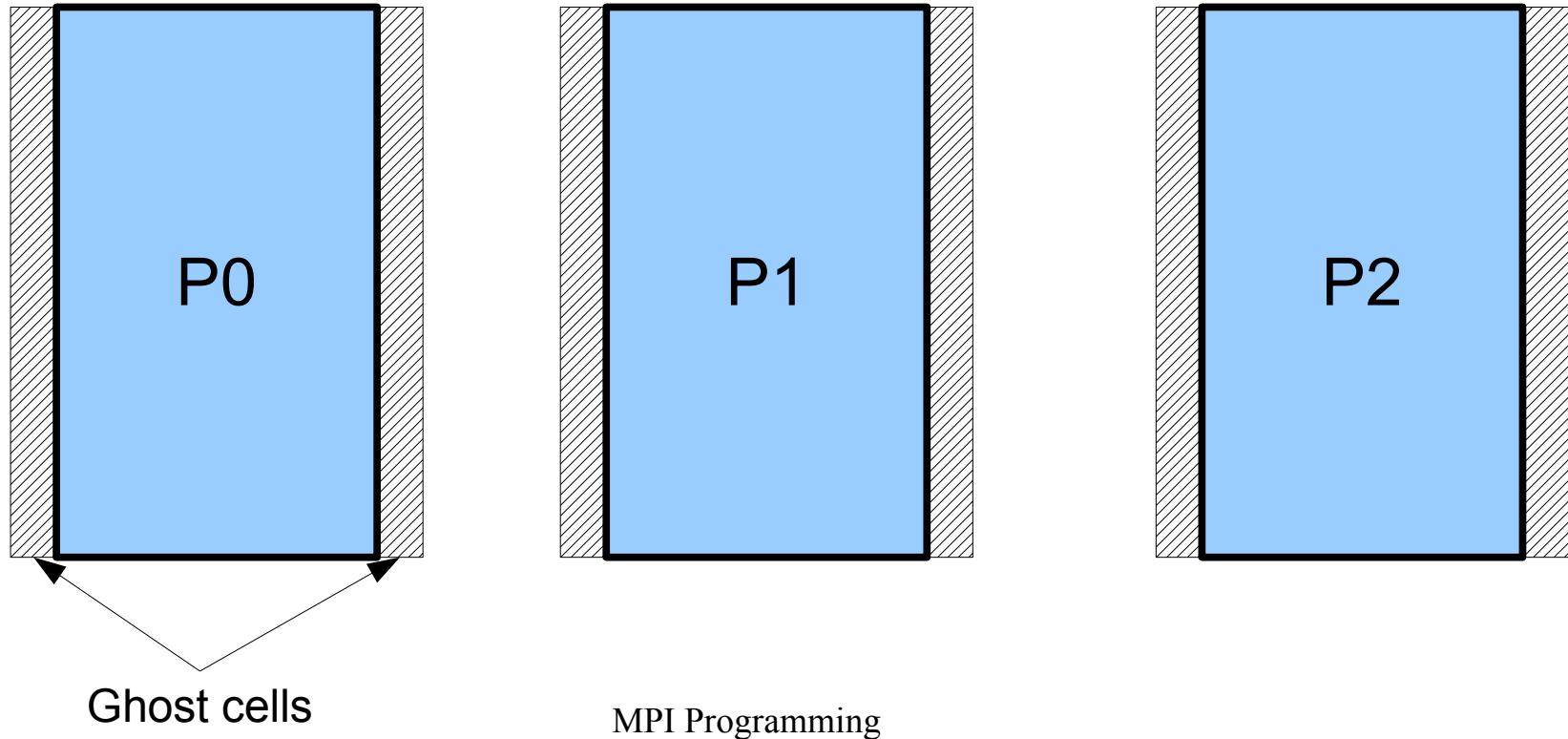


Or



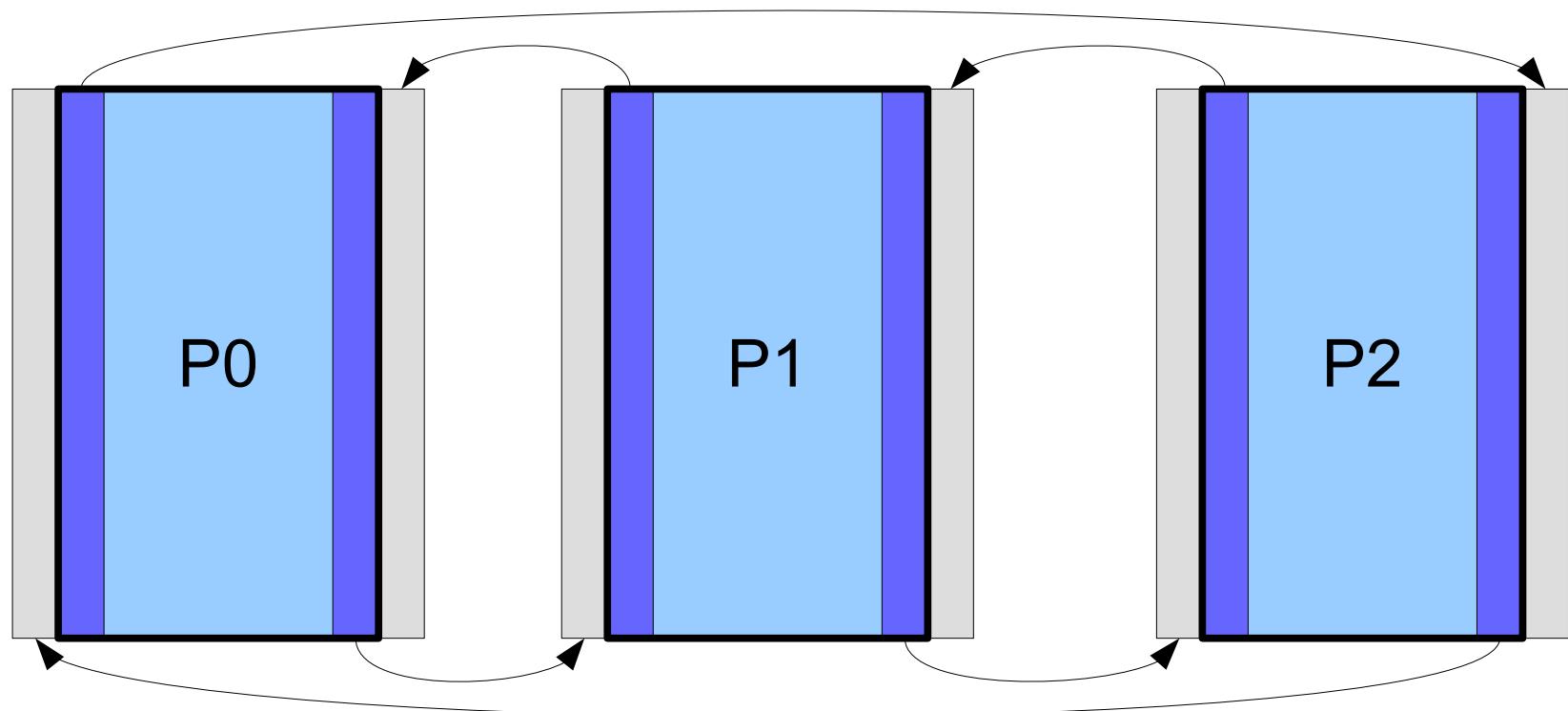
# Example

- Let us consider a two-dimensional domain
- (\*, Block) decomposition
  - with ghost cells along the vertical edges only



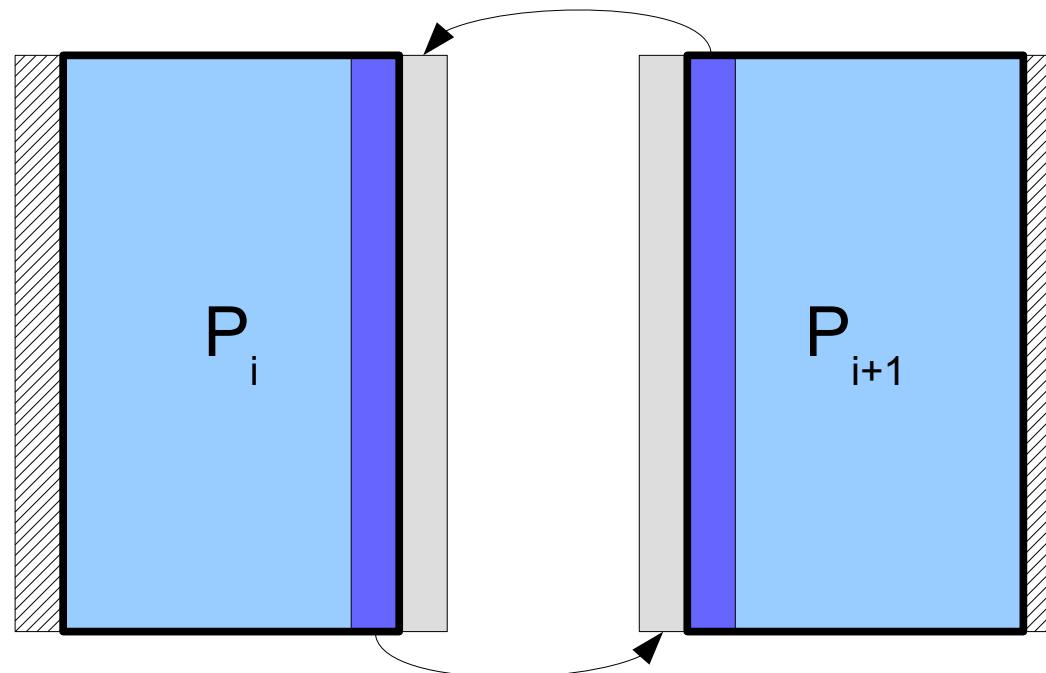
# Example

- At each step, nodes must exchange their outer columns with neighbors



# Example

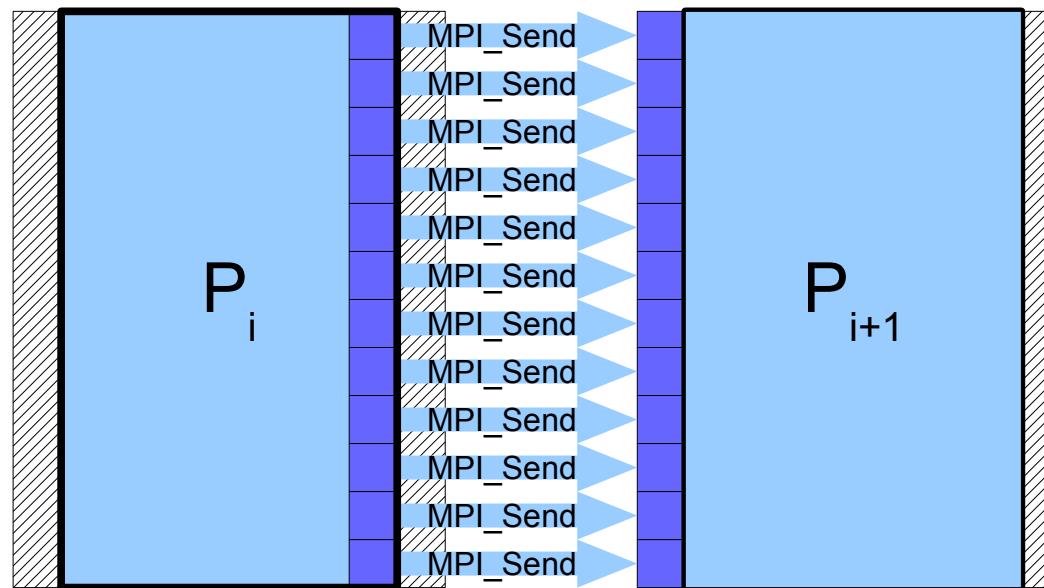
- In the C language, matrices are stored row-wise
  - Elements of the same column are not contiguous in memory





# Example

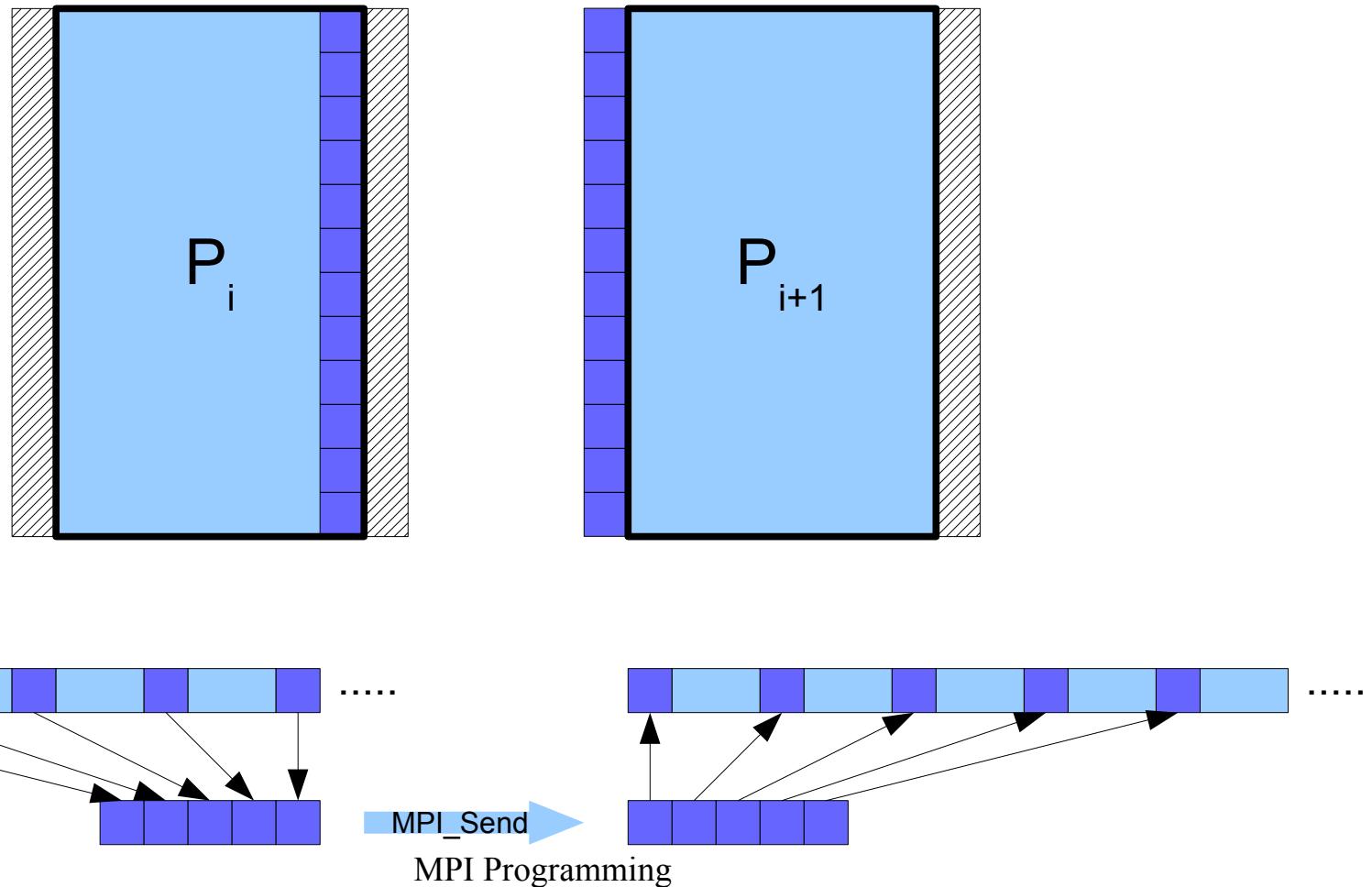
- **The BAD** solution: send each element with `MPI_Send` (or `MPI_Isend`)





# Example

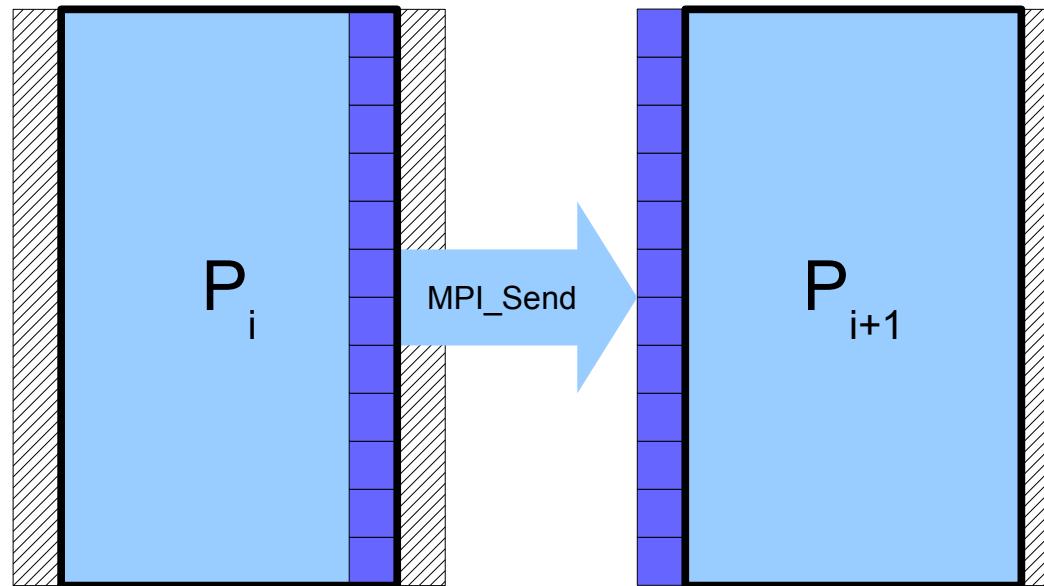
- **The UGLY** solution: copy the column into a temporary buffer; `MPI_Send()` the buffer; fill the destination column





# Example

- **The GOOD** solution: define a new datatype for the column, and MPI\_Send the column directly





# How to use

1. Construct the datatype using a template or *constructor*.
2. Allocate the datatype.
3. Use the datatype.
4. Deallocate the datatype.

You must construct and allocate a datatype before using it. You are not required to use it or deallocate it, but it is recommended (there may be a limit).

# Datatype constructors

- **MPI\_Type\_contiguous**
  - Simplest constructor. Makes count copies of an existing datatype
- **MPI\_Type\_vector, MPI\_Type\_hvector**
  - Like contiguous, but allows for regular gaps (stride) in the displacements. For MPI\_Type\_hvector the stride is specified in bytes.
- **MPI\_Type\_indexed, MPI\_Type\_hindexed**
  - An array of displacements of the input data type is provided as the map for the new data type. MPI\_Type\_hindexed is identical to MPI\_Type\_indexed except that offsets are specified in byte
- **MPI\_Type\_struct**
  - The most general of all derived datatypes. The new data type is formed according to completely defined map of the component data types



# Allocating/deallocating and using datatypes

- C
  - `int MPI_Type_commit (MPI_datatype *datatype)`
  - `int MPI_Type_free (MPI_datatype *datatype)`
- FORTRAN
  - `INTEGER DATATYPE, MPIERROR`
  - `MPI_TYPE_COMMIT(DATATYPE, MPIERROR)`
  - `MPI_TYPE_FREE(DATATYPE, MPIERROR)`
- C Example
  - `MPI_Type_vector(count, blocklength, stride, oldtype, &newtype);`
  - `MPI_Type_commit (&newtype);`
  - `MPI_Send(buffer, 1, newtype, dest, tag, comm);`

# MPI\_TYPE\_CONTIGUOUS

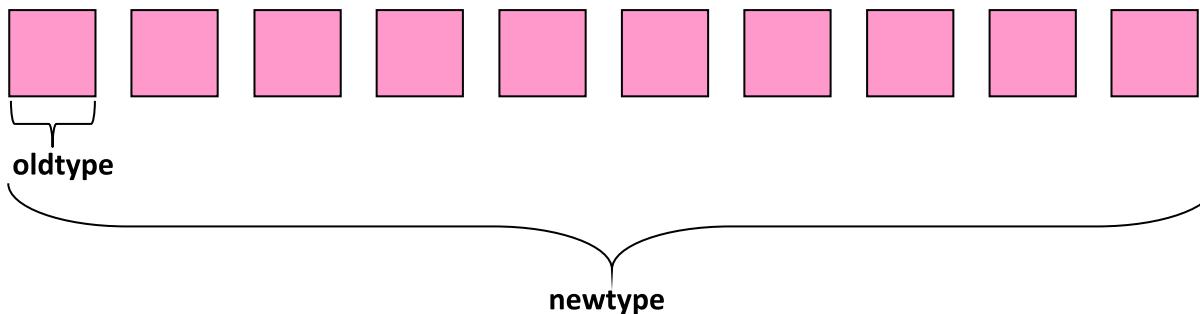
**MPI\_TYPE\_CONTIGUOUS (count, oldtype, newtype)**

IN count: replication count (non-negative integer)

IN oldtype: old datatype (handle)

OUT newtype: new datatype (handle)

- MPI\_TYPE\_CONTIGUOUS constructs a typemap consisting of the **replication** of a **datatype** into contiguous locations.
- newtype is the datatype obtained by concatenating count copies of oldtype.



# MPI\_TYPE\_VECTOR

**MPI\_TYPE\_VECTOR (count, blocklength, stride, oldtype, newtype)**

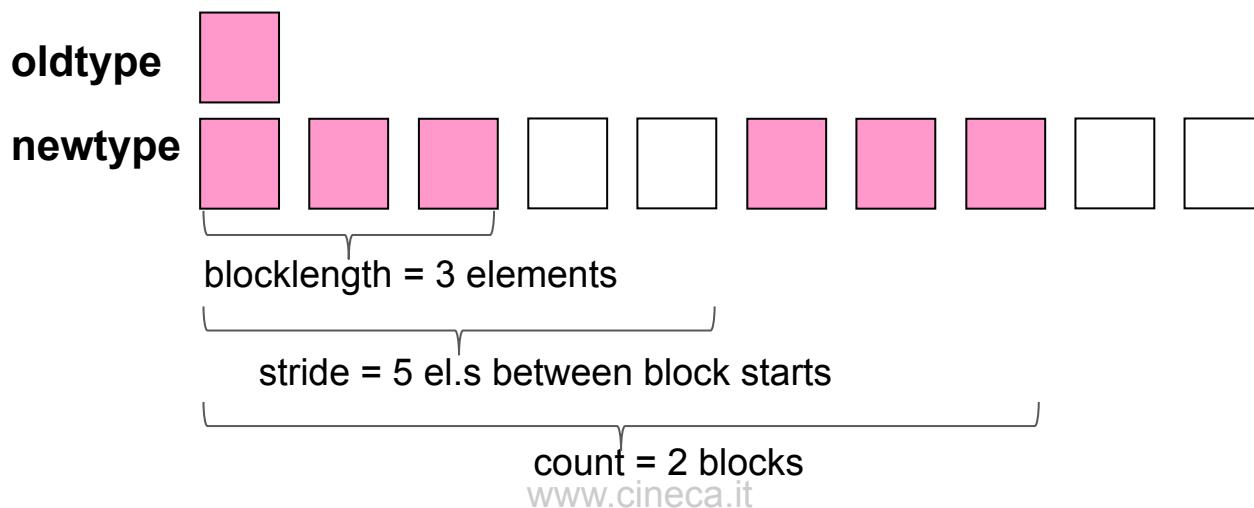
IN count: Number of blocks (non-negative integer)

IN blocklen: Number of elements in each block (non-negative integer)

IN stride: Number of elements (NOT bytes) between start of each block (integer)

IN oldtype: Old datatype (handle)

OUT newtype: New datatype (handle)



# Example 1 – A rowtype

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

|      |      |      |      |
|------|------|------|------|
| 1.0  | 2.0  | 3.0  | 4.0  |
| 5.0  | 6.0  | 7.0  | 8.0  |
| 9.0  | 10.0 | 11.0 | 12.0 |
| 13.0 | 14.0 | 15.0 | 16.0 |

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

|     |      |      |      |
|-----|------|------|------|
| 9.0 | 10.0 | 11.0 | 12.0 |
|-----|------|------|------|

1 element of  
rowtype

## Example 2 - columntype

```
count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
 &columntype);
```

|      |      |      |      |
|------|------|------|------|
| 1.0  | 2.0  | 3.0  | 4.0  |
| 5.0  | 6.0  | 7.0  | 8.0  |
| 9.0  | 10.0 | 11.0 | 12.0 |
| 13.0 | 14.0 | 15.0 | 16.0 |

a[4][4]

```
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```

|     |     |      |      |
|-----|-----|------|------|
| 2.0 | 6.0 | 10.0 | 14.0 |
|-----|-----|------|------|

1 element of  
columntype

# Quiz

```
int count = 4, blocklen = 2, stride = 4;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```

|      |      |      |      |
|------|------|------|------|
| 1.0  | 2.0  | 3.0  | 4.0  |
| 5.0  | 6.0  | 7.0  | 8.0  |
| 9.0  | 10.0 | 11.0 | 12.0 |
| 13.0 | 14.0 | 15.0 | 16.0 |

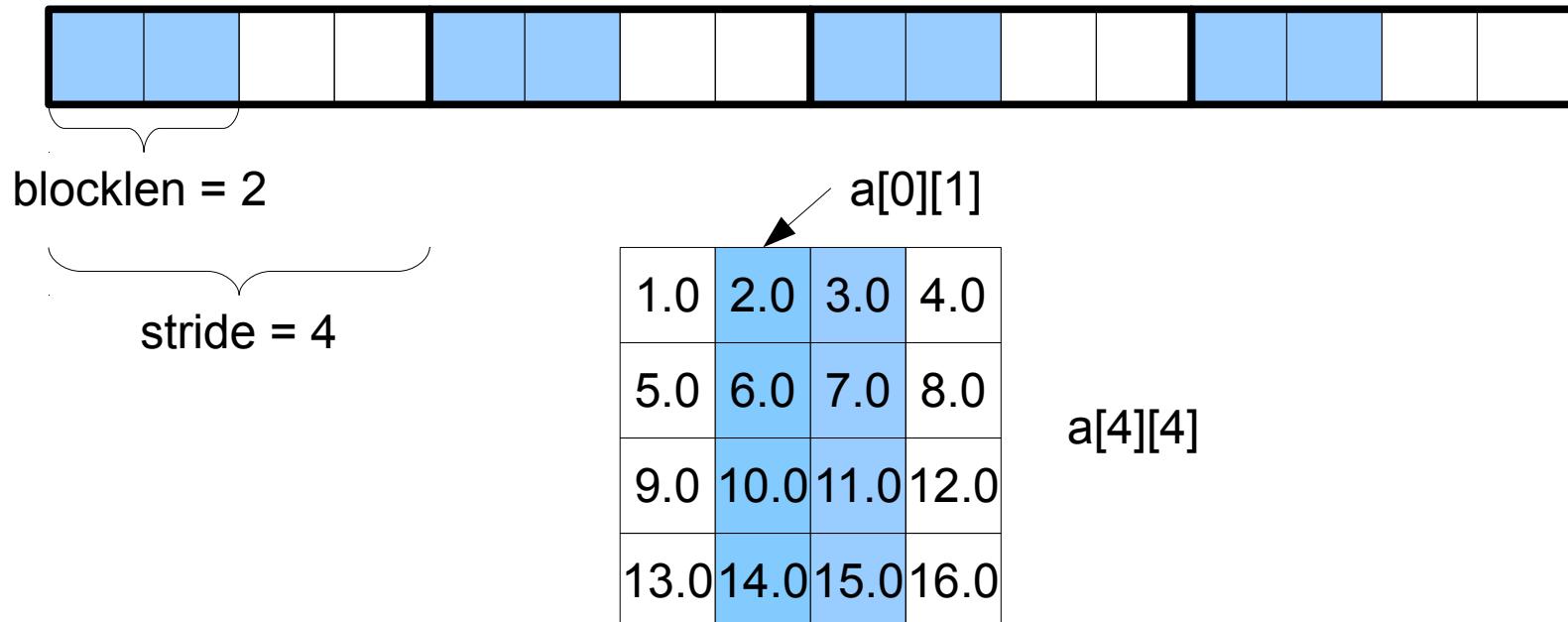
a[4][4]

```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

**Which data are being transmitted?**

# Quiz

```
int count = 4, blocklen = 2, stride = 4;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```



```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

|     |     |     |     |      |      |      |      |
|-----|-----|-----|-----|------|------|------|------|
| 2.0 | 3.0 | 6.0 | 7.0 | 10.0 | 11.0 | 14.0 | 15.0 |
|-----|-----|-----|-----|------|------|------|------|

# Quiz

```
int count = 3, blocklen = 1, stride = 5;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```

|      |      |      |      |
|------|------|------|------|
| 1.0  | 2.0  | 3.0  | 4.0  |
| 5.0  | 6.0  | 7.0  | 8.0  |
| 9.0  | 10.0 | 11.0 | 12.0 |
| 13.0 | 14.0 | 15.0 | 16.0 |

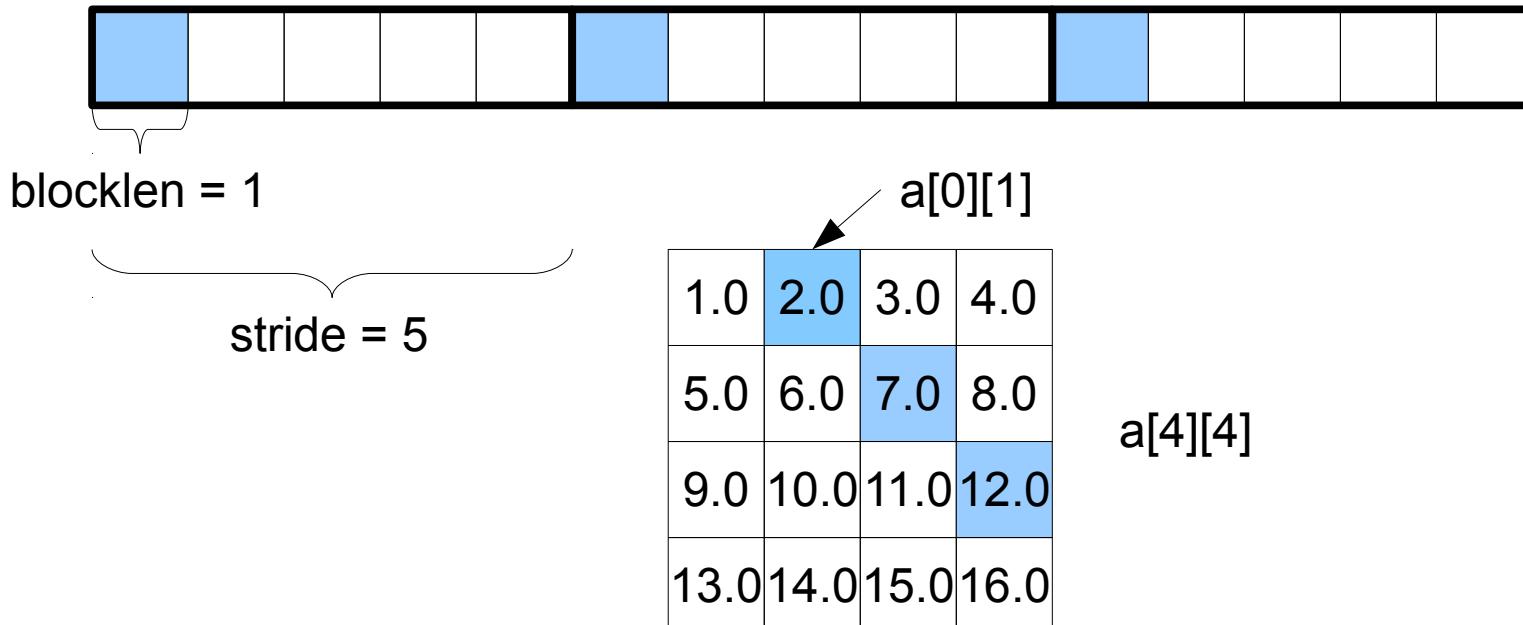
a[4][4]

```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

**Which data are being transmitted?**

# Quiz

```
int count = 3, blocklen = 1, stride = 5;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```



```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

|     |     |      |
|-----|-----|------|
| 2.0 | 7.0 | 12.0 |
|-----|-----|------|

# Quiz

```
int count = ???, blocklen = ???, stride = ???;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```

Fill the ??? with the parameters required  
to get the behavior below

|      |      |      |      |
|------|------|------|------|
| 1.0  | 2.0  | 3.0  | 4.0  |
| 5.0  | 6.0  | 7.0  | 8.0  |
| 9.0  | 10.0 | 11.0 | 12.0 |
| 13.0 | 14.0 | 15.0 | 16.0 |

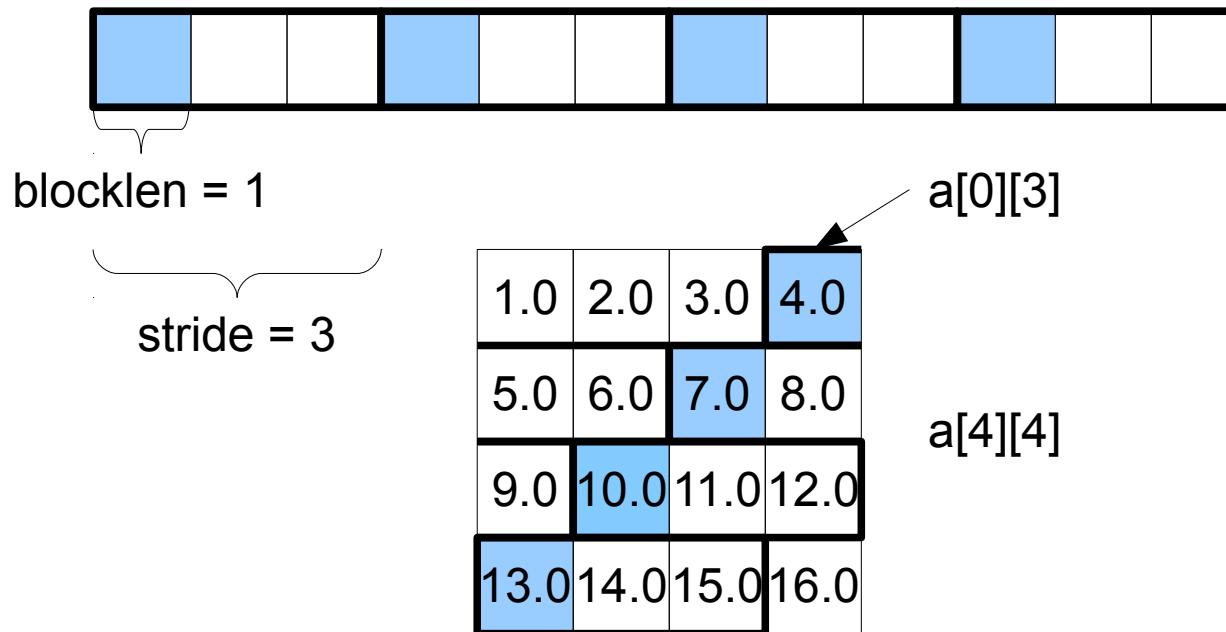
a[4][4]

```
MPI_Send(?????????, 1, newtype, dest, tag, MPI_COMM_WORLD);
```

|     |     |      |      |
|-----|-----|------|------|
| 4.0 | 7.0 | 10.0 | 13.0 |
|-----|-----|------|------|

# Quiz

```
int count = 4; blocklen = 1, stride = 3;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```



```
MPI_Send(&a[0][3], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

|     |     |      |      |
|-----|-----|------|------|
| 4.0 | 7.0 | 10.0 | 13.0 |
|-----|-----|------|------|



# Other tools

## MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS

- Routines which return the number of "copies" of type datatype and the number of basic elements (often used after a MPI\_RECV).

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype, int *count)
```

## MPI\_TYPE\_GET\_EXTENT (Advanced)

- Returns the lower bound and extent of a datatype (i.e. upper bound + padding to align the datatype). Useful for creating new datatypes with MPI\_TYPE\_CREATE\_RESIZED, for example.

# MPI TYPE INDEXED

**`MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)`**

IN `count`: number of blocks - also number of entries in `array_of_blocklengths` and  
`array_of_displacements` (non-negative integer)

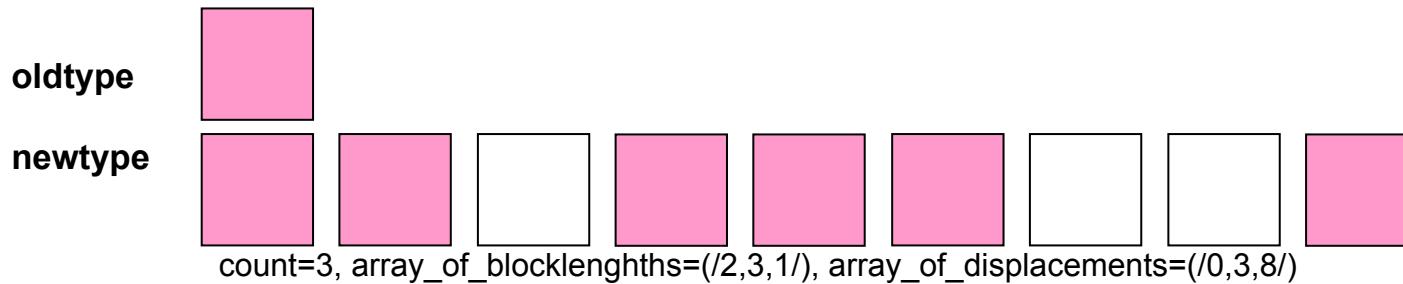
IN `array_of_blocklengths`: number of elements per block (array of non-negative  
integers)

IN `array_of_displacements`: displacement for each block, in multiples of `oldtype`  
extent (array of integer)

IN `oldtype`: old datatype (handle)

OUT `newtype`: new datatype (handle)

- Creates a new type from blocks comprising identical elements
- The size and displacements of the blocks can vary



# MPI TYPE INDEXED

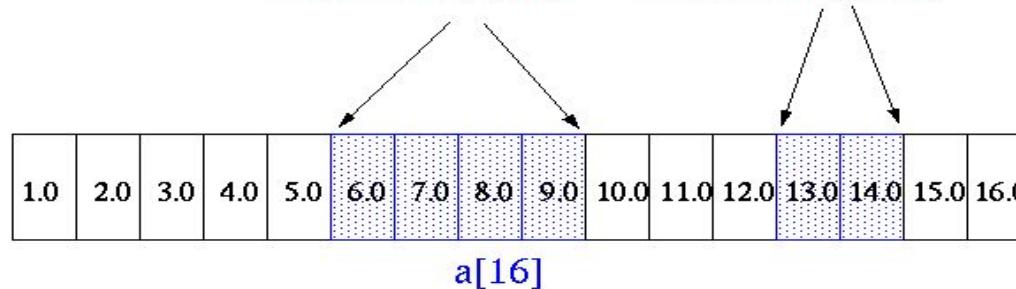
count = 2;

blocklengths[0] = 4;

displacements[0] = 5;

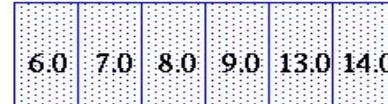
blocklengths[1] = 2;

displacements[1] = 12;



```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



1 element of  
indextype

# Combining custom datatypes

- The `oldtype` parameter of functions

`MPI_Type_contiguous()`, `MPI_Type_vector()` and `MPI_Type_indexed()` can be another user-defined datatype

```
int count, blocklen, stride;
MPI_Datatype vec, vecvec;

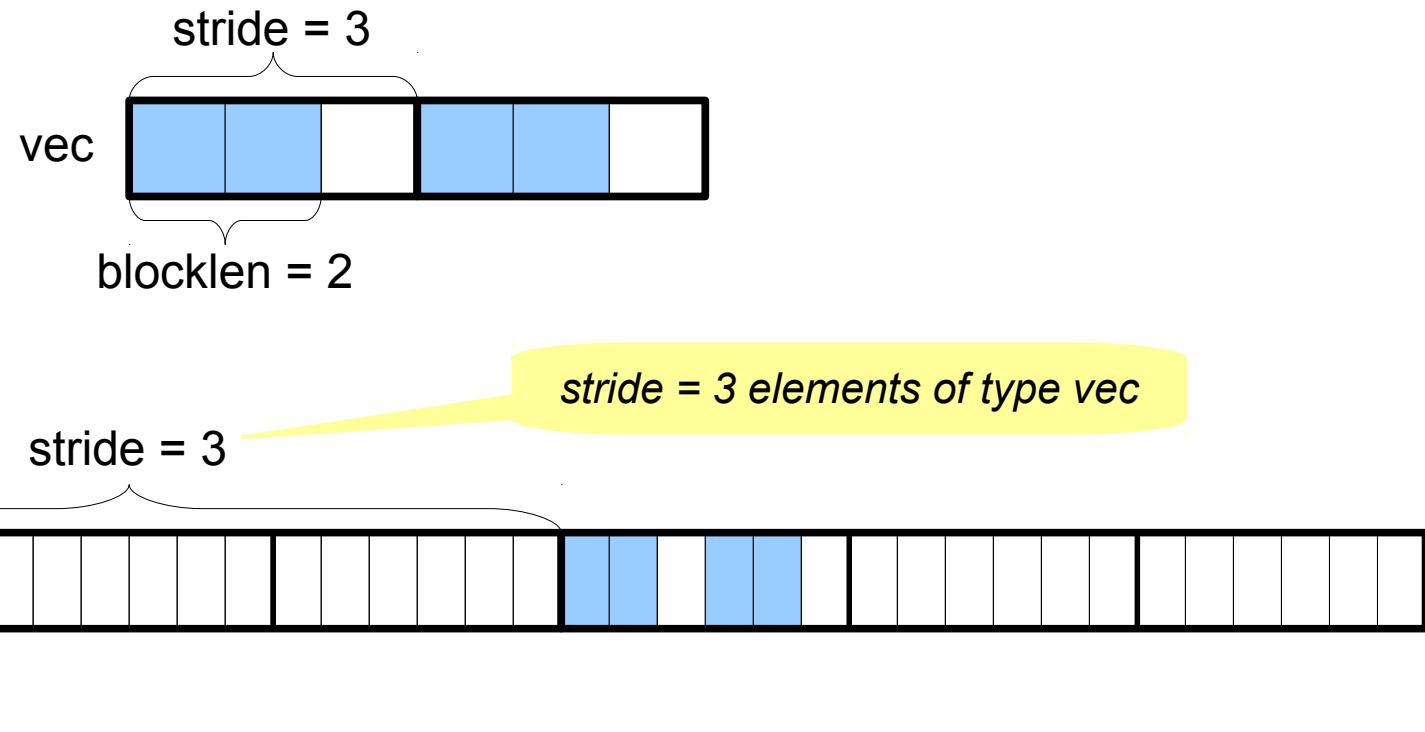
count = 2; blocklen = 2; stride = 3;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &vec);
MPI_Type_commit(&vec);
count = 2; blocklen = 1; stride = 3
MPI_Type_vector(count, blocklen, stride, vec, &vecvec);
MPI_Type_commit(&vecvec);
```

```

int count, blocklen, stride;
MPI_Datatype vec, vecvec;

count = 2; blocklen = 2; stride = 3;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &vec);
MPI_Type_commit(&vec);
count = 2; blocklen = 1; stride = 3
MPI_Type_vector(count, blocklen, stride, vec, &vecvec);
MPI_Type_commit(&vecvec);

```





# MPI TYPE SUBARRAY

```
MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes, array_of_starts,
 order, oldtype, newtype)
```

IN ndims: number of array dimensions (positive integer)

IN array\_of\_sizes: number of elements of type oldtype in each  
dimension of the full array (array of positive integers)

IN array\_of\_subsizes: number of elements of type oldtype in each  
dimension of the subarray (array of positive integers)

IN array\_of\_starts: starting coordinates of the subarray in each  
dimension (array of non-negative integers)

IN order: array storage order flag (state: MPI\_ORDER\_C or MPI\_ORDER\_FORTRAN)

IN oldtype: array element datatype (handle)

OUT newtype: new datatype (handle)

The subarray type constructor creates an MPI datatype describing an n dimensional subarray of an n-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array.

# MPI TYPE SUBARRAY - EXAMPLE (C)

```
MPI_TYPE_CREATE_SUBARRAY (ndims, array_of_sizes, array_of_subsizes,
array_of_starts, order, oldtype, newtype)
```

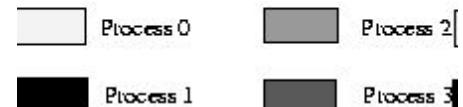
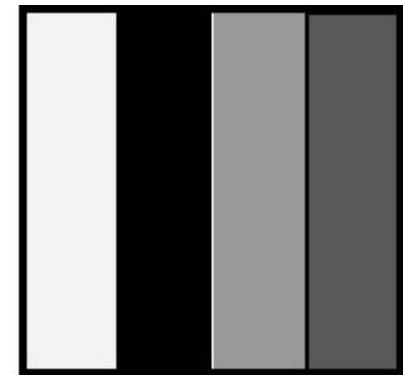
```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &filetype);

MPI_Type_commit(&filetype);
```



# SIZE AND EXTENT

The MPI datatype for structures – MPI\_TYPE\_CREATE\_STRUCT – requires dealing with memory addresses and further concepts:

**Typemap:** pairs of basic types and displacements

**Extent:** The extent of a datatype is the span from the lower to the upper bound (including “holes”)

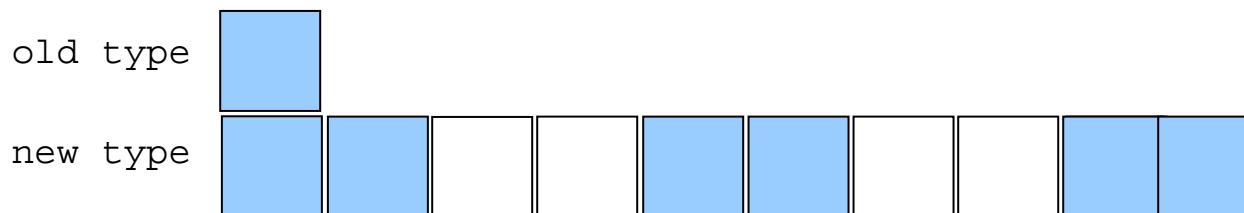
**Size:** The size of a datatype is the net number of bytes to be transferred (without “holes”)



derived datatype

# SIZE AND EXTENT

- **Basic datatypes:**
  - size = extent = number of bytes used by the compiler
- **Derived datatypes:**
  - extent include holes but...
  - beware of the type vector: final holes are a figment of our imagination



- size = 6 x size of “old type”
- extent = 10 x extent of “old type”

# QUERY SIZE AND EXTENT OF DATATYPE

- Returns the total number of bytes of the entry datatype

**MPI\_TYPE\_SIZE (datatype, size)**

IN datatype: datatype (handle)

OUT size: datatype size (integer)

- Returns the lower bound and the extent of the entry datatype

**MPI\_TYPE\_EXTENT (datatype, extent)**

IN datatype: datatype to get information on (handle)

OUT extent: extent of datatype (integer)



# MPI TYPE STRUCT

```
MPI_TYPE_CREATE_STRUCT (count, array_of_blocklengths, array_of_displacements,
 array_of_olddtypes, newtype)

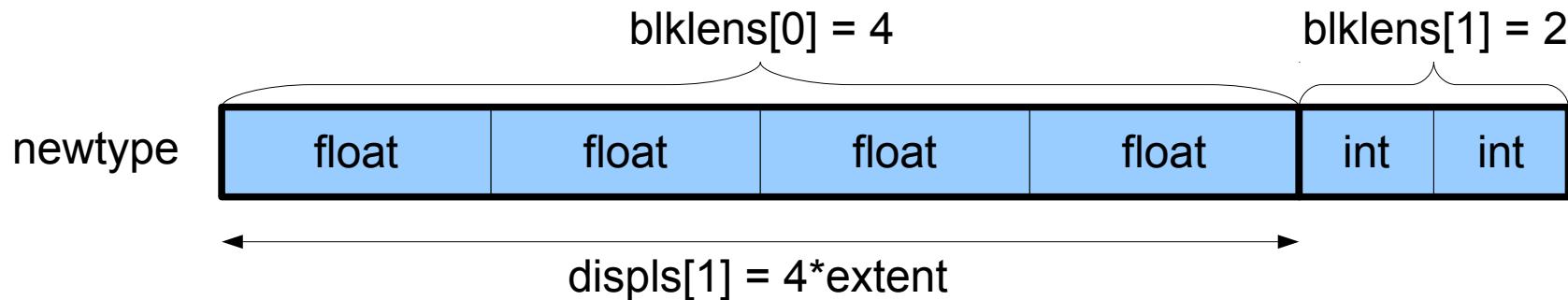
IN count: number of blocks (non-negative integer) -- also number of entries the
 following arrays
IN array_of_blocklengths: number of elements in each block
 (array of non-negative integer)
IN array_of_displacements: byte displacement of each block (array of integer)
IN array_of_olddtypes: type of elements in each block
 (array of handles to datatype objects)
OUT newtype: new datatype (handle)
```

- This subroutine returns a new datatype that represents count blocks. Each block is defined by an entry in array\_of\_blocklengths, array\_of\_displacements and array\_of\_types.
  - Displacements are expressed in bytes (since the type can change!)
  - To gather a mix of different datatypes scattered at many locations in space into one datatype that can be used for the communication.

# MPI\_Type\_struct()

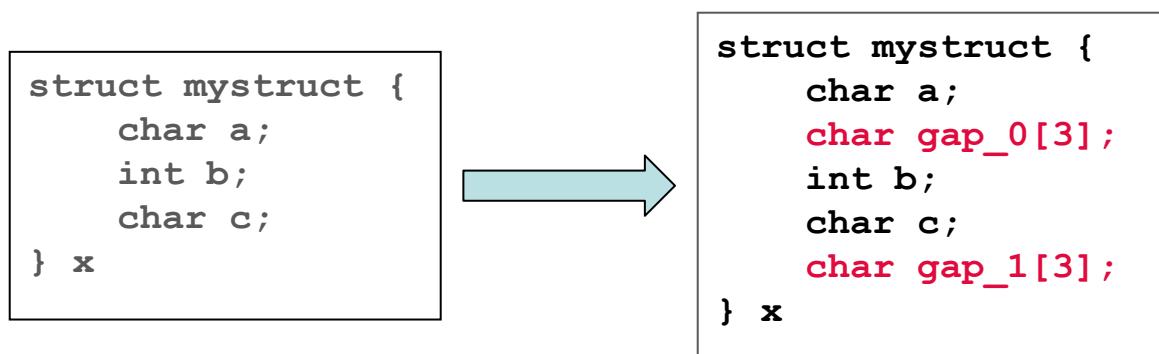
```
typedef struct {
 float x, y, z, v;
 int n, t;
} particle_t;

int count = 2; int blklen[] = {4, 2};
MPI_Aint displs[2], lb, extent;
MPI_Datatype oldtypes[2] = {MPI_FLOAT, MPI_INT}, newtype;
MPI_Type_get_extent(MPI_FLOAT, &lb, &extent);
displs[0] = 0; displs[1] = 4*extent;
MPI_Type_struct(count, blklen, displs, oldtypes, &newtype);
MPI_Type_commit(&newtype);
```



# USING EXTENT (NOT SAFE)

C struct may be automatically padded by the compiler, e.g.



Using extents to handle structs is not safe! Get the addresses



# MPI\_GET\_ADDRESS

**MPI\_GET\_ADDRESS (location, address)**

IN location: location in caller memory (choice)

OUT address: address of location (integer)

The address of the variable is returned, which can then be used to determine the correct relative dispacements

Using this function helps with portability



# USING DISPLACEMENTS

```
MPI_Datatype ParticleType;
int count = 3;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];
```

```
MPI_Get_address(&particle[0].class, &disp[0]);
MPI_Get_address(&particle[0].d, &disp[1]);
MPI_Get_address(&particle[0].b, &disp[2]);
/* Make displacements relative */
disp[2] -= disp[0]; disp[1] -= disp[0]; disp[0] = 0;

MPI_Type_create_struct(count, blocklen, disp, type, &ParticleType);
MPI_Type_commit(&ParticleType);

MPI_Send(particle, 100, ParticleType, dest, tag, comm);
MPI_Type_free (&ParticleType);
```

```
struct PartStruct {
 char class;
 double d[6];
 int b[7];
} particle[100];
```



# Derived Datatype Summary

- Provide a portable and elegant way of communicating non-contiguous or mixed types in a message.
- By optimising how data is stored, should improve efficiency during MPI send and receive (perhaps avoiding buffering).
- Derived datatypes are built from basic MPI datatypes, according to a template. Can be used for many variables of the same form.
- Remember to commit the datatypes before using them.

Alessandro Marani

a.marani@cineca.it

High Performance Computing department

# Parallel programming with MPI

Communicators and Groups





# MPI communicators and groups

Many users are familiar with the mostly used communicator:

## **MPI\_COMM\_WORLD**

A **communicator** can be thought as a handle to a **group**.

- a group is a ordered set of processes
  - each process is associated with a rank
  - ranks are contiguous and start from zero

Groups allow collective operations to be operated on a subset of processes

The group routines are primarily used to specify which processes should be used to construct a communicator.



# MPI communicators and groups

## Intracomunicators

are used for communications within a single group

## Intercommunicators

are used for communications between two disjoint groups

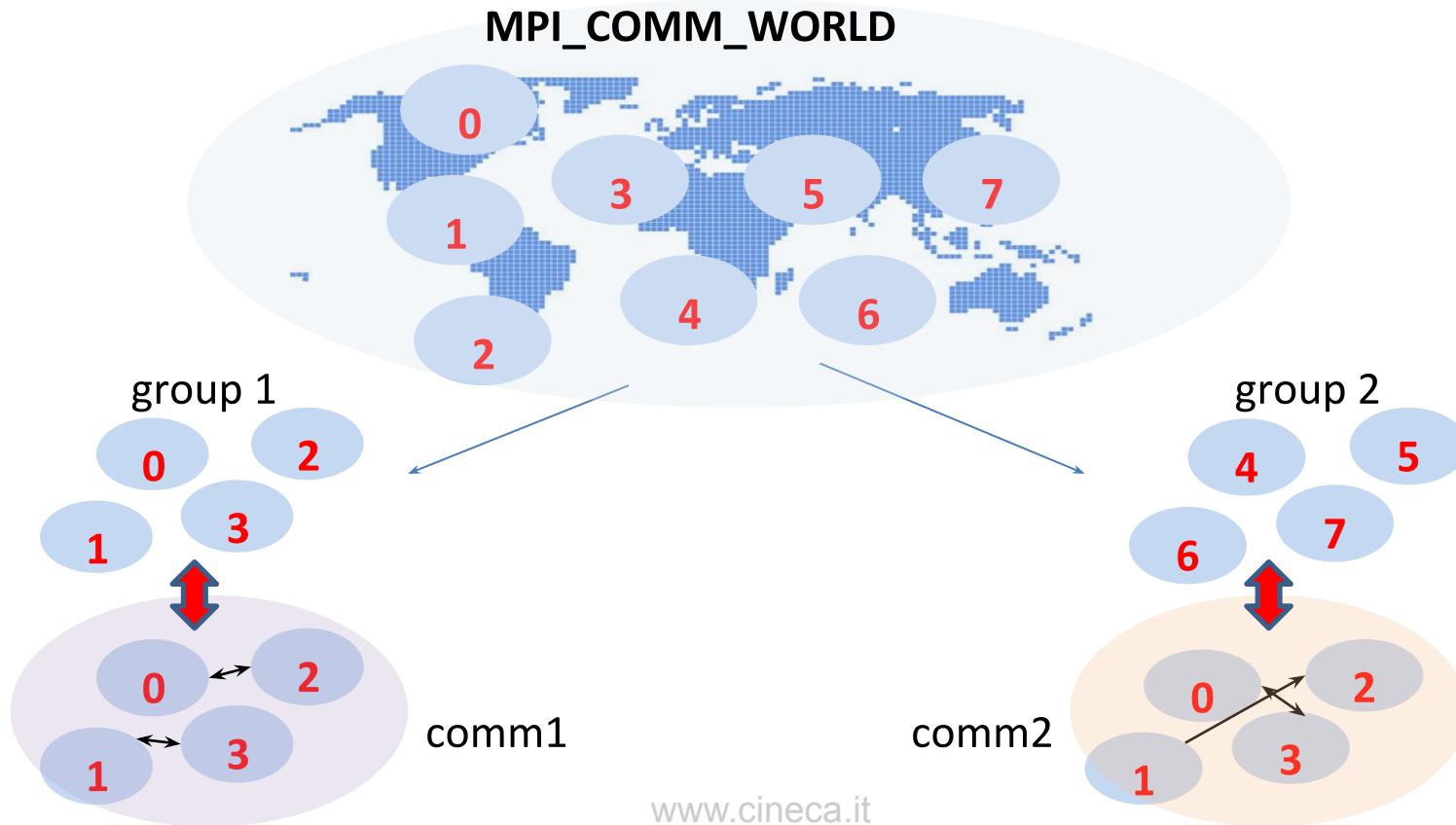


# Using MPI Groups

Typical usage:

1. Extract handle of global group from `MPI_COMM_WORLD` using `MPI_Comm_group`
2. Form new group as a subset of global group using `MPI_Group_incl`
3. Create new communicator for new group using `MPI_Comm_create`
4. Determine new rank in new communicator using `MPI_Comm_rank`
5. Conduct communications using any MPI message passing routine
6. When finished, free up new communicator and group (optional) using `MPI_Comm_free` and `MPI_Group_free`

# Group Creation





# Group accessors

## **MPI\_GROUP\_SIZE**

This routine returns the number of processes in the group

## **MPI\_GROUP\_RANK**

This routine returns the rank of the calling process inside a given group



# Group accessors

**MPI\_COMM\_GROUP**(comm,group,ierr)

This routine returns the group associated with the communicator comm

**MPI\_GROUP\_UNION**(group\_a, group\_b, newgroup, ierr)

This returns the ensemble union of group\_a and group\_b

**MPI\_GROUP\_INTERSECTION**(group\_a, group\_b, newgroup, ierr)

This returns the ensemble intersection of group\_a and group\_b

**MPI\_GROUP\_DIFFERENCE**(group\_a, group\_b, newgroup, ierr)

This returns in newgroup all processes in group\_a that are not in group\_b, ordered as in group\_a



# Group accessors

**MPI\_GROUP\_INCL**(group, n, ranks, newgroup, ierr)

This routine creates a new group that consists of all the n processes with ranks  
ranks[0]... ranks[n-1]

Example:

group = {a,b,c,d,e,f,g,h,i,j}

n = 5

ranks = {0,3,8,6,2}

newgroup = {a,d,i,g,c}



# Group accessors

**MPI\_GROUP\_EXCL**(group,n,ranks,newgroup,ierr)

This routine returns a newgroup that consists of all the processes in the group after removing processes with ranks: ranks[0]..ranks[n-1]

Example:

group = {a,b,c,d,e,f,g,h,i,j}

n = 5

ranks = {0,3,8,6,2}

newgroup = {b,e,f,h,j}



# Communicator accessors

## **MPI\_COMM\_SIZE**(comm,size,ierr)

Returns the number of processes in the group associated with the comm

## **MPI\_COMM\_RANK**(comm,rank,ierr)

Returns the rank of the calling process within the group associated with the comm

## **MPI\_COMM\_COMPARE**(comm1,comm2,result,ierr)

Returns:

- MPI\_IDENT if comm1 and comm2 are the same handle
- MPI\_CONGRUENT if comm1 and comm2 have the same group attribute
- MPI\_SIMILAR if the groups associated with comm1 and comm2 have the same members but in different rank order
- MPI\_UNEQUAL otherwise



# Communicator constructors

**MPI\_COMM\_DUP**(comm, newcomm,ierr)

This returns a communicator newcomm identical to the communicator comm

**MPI\_COMM\_CREATE**(comm, group, newcomm,ierr)

This collective routine must be called by all the process involved in the group associated with comm. It returns a new communicator that is associated with the group. **MPI\_COMM\_NULL** is returned to processes not in the group.

Note that group must be a subset of the group associated with comm!



# Communicator constructors

**MPI\_COMM\_SPLIT**(comm, color, key, newcomm, ierr)

This routine creates as many new groups and communicators as there are distinct values of color.

(processes in the same color are in the same communicator)

The **rankings** in the new groups are determined by the value of the key.

**MPI\_UNDEFINED** is used as the color for processes to not be included in any of the new groups

# Example

| Rank    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---------|---|---|---|---|---|---|---|---|---|---|----|
| Process | a | b | c | d | e | f | g | h | i | j | k  |
| Color   | U | 3 | 1 | 1 | 3 | 7 | 3 | 3 | 1 | U | 3  |
| Key     | 0 | 1 | 2 | 3 | 1 | 9 | 3 | 8 | 1 | 0 | 0  |

Both process a and j are returned MPI\_COMM\_NULL  
3 new groups are created

{i, c, d}

{k, b, e, g, h}

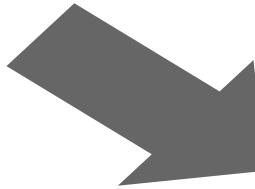
{f}

# MPI\_comm\_split

MPI provides functions to manage and to create groups and communicators.

**MPI\_comm\_split**, for example, creates a communicator...

```
if(myid%2==0){
 color=1;
}else{
 color=2;
}
MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,
 &subcomm);
MPI_COMM_RANK(subcomm,mynewid);
printf("rank in MPICOMM_WORLD %d",myid,
 "rank in Subcomm %d",mynewid);
```



I am rank 2 in MPI\_COMM\_WORLD, but 1 in Comm 1.  
I am rank 7 in MPI\_COMM\_WORLD, but 3 in Comm 2.  
I am rank 0 in MPI\_COMM\_WORLD, but 0 in Comm 1.  
I am rank 4 in MPI\_COMM\_WORLD, but 2 in Comm 1.  
I am rank 6 in MPI\_COMM\_WORLD, but 3 in Comm 1.  
I am rank 3 in MPI\_COMM\_WORLD, but 1 in Comm 2.  
I am rank 5 in MPI\_COMM\_WORLD, but 2 in Comm 2.  
I am rank 1 in MPI\_COMM\_WORLD, but 0 in Comm 2.

# Example

```
int MPI_Comm_split(MPI_Comm old_comm, int color, int key, MPI_Comm *new_comm)
```

For a 2D logical grid, create subgrids of rows and columns

```
!! logical row number
```

```
irow = Iam/mcol
```

```
!! logical column number
```

```
jcol = mod(Iam, mcol)
```

```
comm2D = MPI_COMM_WORLD
```

```
call MPI_Comm_split(comm2D, irow, jcol, row_comm, ierr)
```

```
call MPI_Comm_split(comm2D, jcol, irow, col_comm, ierr)
```

[www.cineca.it](http://www.cineca.it)

Figure a.  
2D logical Grid

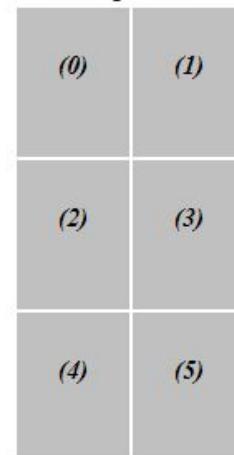
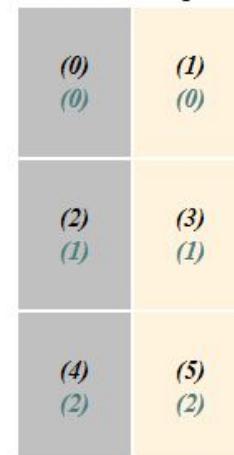


Figure b.  
3 Row Subgrids



Figure c.  
2 Column Subgrids



|      |   |   |   |   |   |   |
|------|---|---|---|---|---|---|
| Iam  | 0 | 1 | 2 | 3 | 4 | 5 |
| irow | 0 | 0 | 1 | 1 | 2 | 2 |
| jcol | 0 | 1 | 0 | 1 | 0 | 1 |

# Virtual Topology

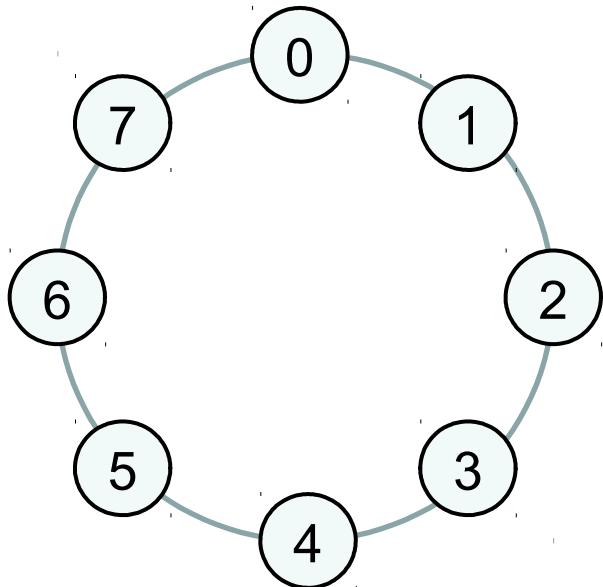
## Virtual topology:

- logical process arrangement in topological patterns such as 2D or 3D grid; more generally, the logical process arrangement is described by a graph.

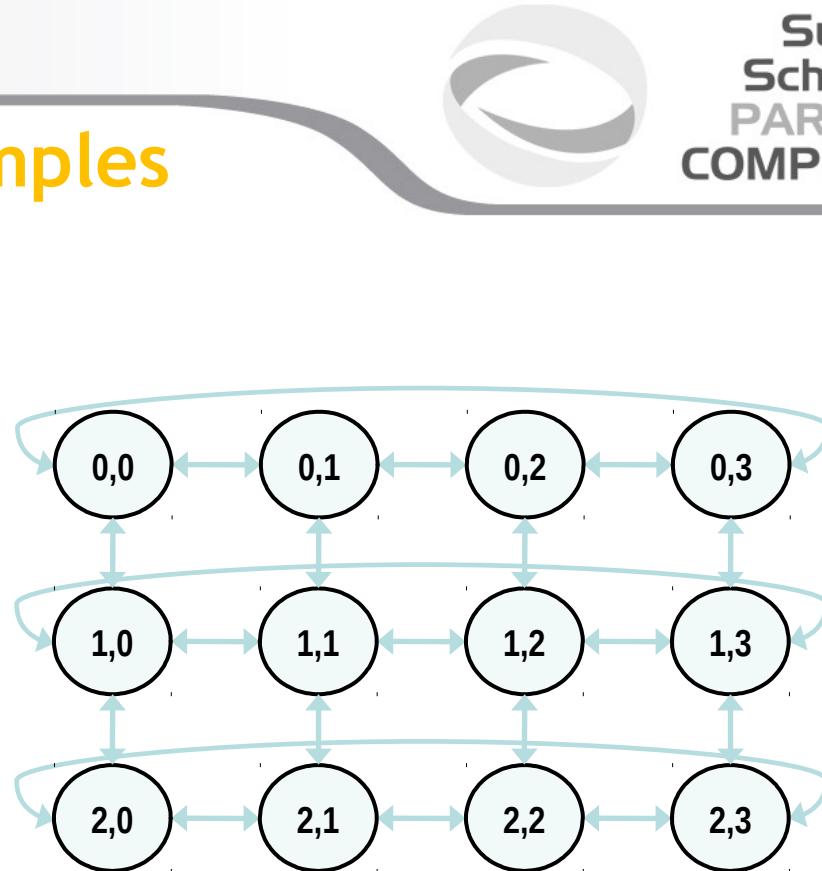
## Virtual process topology .vs. topology of the underlying, physical hardware:

- virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine.
- the description of the virtual topology depends only on the application, and is machine-independent.

# Virtual Topology - Examples



**RING**

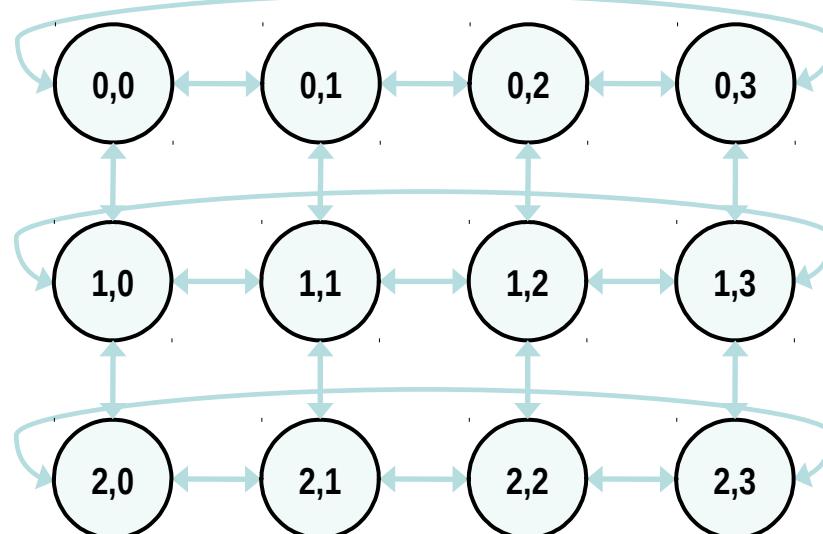


**2D-GRID WITH  
PERIODIC  
BOUNDARY  
CONDITIONS<sub>5</sub>**

# Cartesian Topology

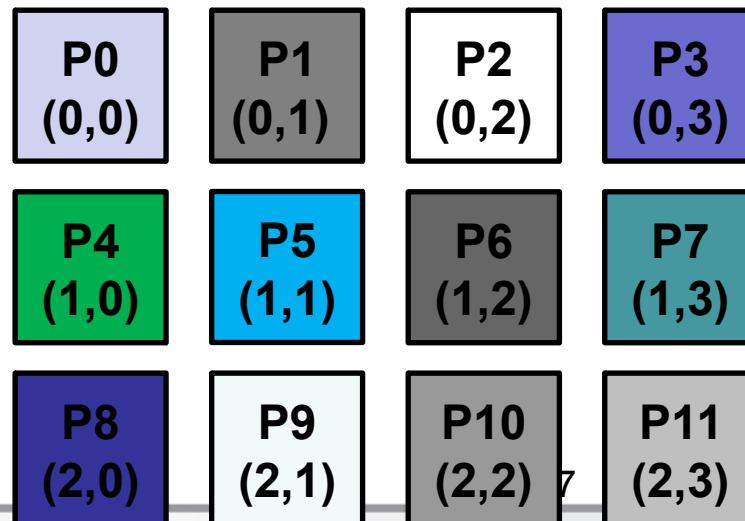
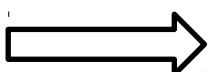
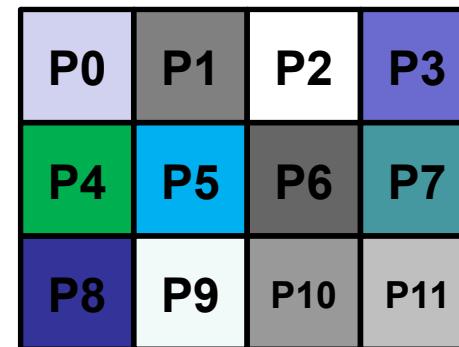
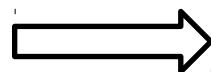
A grid of processes is easily described with a cartesian topology:

- each process can be identified by cartesian coordinates
- periodicity can be selected for each direction
- communications are performed along grid dimensions only



## Example: 2D Domain decomposition

**DATA**



# Cartesian Constructor

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
comm_cart)
```

IN `comm_old`: input communicator (handle)

IN `ndims`: number of dimensions of Cartesian grid (integer)

IN `dims`: integer array of size `ndims` specifying the number of processes in each dimension

IN `periods`: logical array of size `ndims` specifying whether the grid is periodic (true) or not (false) in each dimension

IN `reorder`: ranking may be reordered (true) or not (false)

OUT `comm_cart`: communicator with new Cartesian topology (handle)

- Returns a handle to a new communicator to which the Cartesian topology information is attached.
- Reorder:
  - false: the rank of each process in the new group is identical to its rank in the old group.
  - True: the processes may be reordered, possibly so as to choose a good embedding of the virtual topology onto physical machine.
- If `cart` has less processes than starting communicator, left over processes have `MPI_COMM_NULL` as return

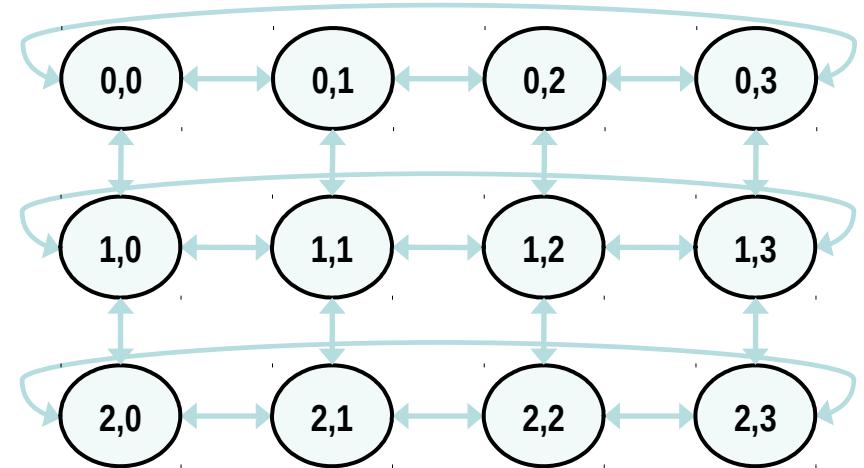
# How to create a Cartesian Topology

```
#include <mpi.h>

int main(int argc, char *argv[])
{
 MPI_Comm cart_comm;
 int dim[] = {4, 3};
 int period[] = {1, 0};
 int reorder = 0;

 MPI_Init(&argc, &argv);

 MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder,
 &cart_comm);
 ...
}
```



# Cartesian Topology Utilities

## **MPI\_Dims\_Create:**

- compute optimal balanced distribution of processes per coordinate direction with respect to:
  - a given dimensionality
  - the number of processes in a group
  - optional constraints

## **MPI\_Cart\_coords:**

- given a rank, returns process's coordinates

## **MPI\_Cart\_rank:**

- given process's coordinates, returns the rank

## **MPI\_Cart\_shift:**

- get source and destination rank ids in SendRecv operations

.

## Rank -> Coordinate: `MPI_Cart_coords`

**`MPI_CART_COORDS(comm, rank, maxdim, coords)`**

IN comm: communicator with Cartesian structure

IN rank: rank of a process within group of comm

IN maxdims: length of vector coords in the calling program

OUT coords: integer array (of size ndims) containing the Cartesian coordinates of specified process

For each MPI process in Cartesian communicator, the coordinate within the cartesian topology are returned

# Usage of MPI\_Cart\_coords

```
. . .
ndim = (int*)calloc(dim,sizeof(int));
ndim[0] = row; ndim[1] = col;

period = (int*)calloc(dim,sizeof(int));
period[0] = period[1] = 0;

reorder = 0;

// 2D grid creation
MPI_Cart_Create(MPI_COMM_WORLD,dim,ndim,period,reorder, &comm_grid);
MPI_Comm_rank(comm_grid,&menum_grid);

// Coordinate of each mpi rank within the cartesian communicator
MPI_Cart_coords(comm_grid,menum,dim,coordinate);

printf("Procs %d coordinates in 2D grid (%d,%d)
\n",menum,*coordinate,*((coordinate+1));
. . .
}
```

# Sendrecv with Cartesian Topologies: MPI\_Cart\_shift

```
MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
```

IN comm: communicator with Cartesian structure

IN direction: coordinate dimension of shift

IN disp: displacement (>0: upwards shift; <0: downwards shift

OUT rank\_source: rank of source process

OUT rank\_dest: rank of destination process

Returns the shifted source and destination ranks, given a shift direction and amount

# Sendrecv with 2D Cartesian Topologies

...

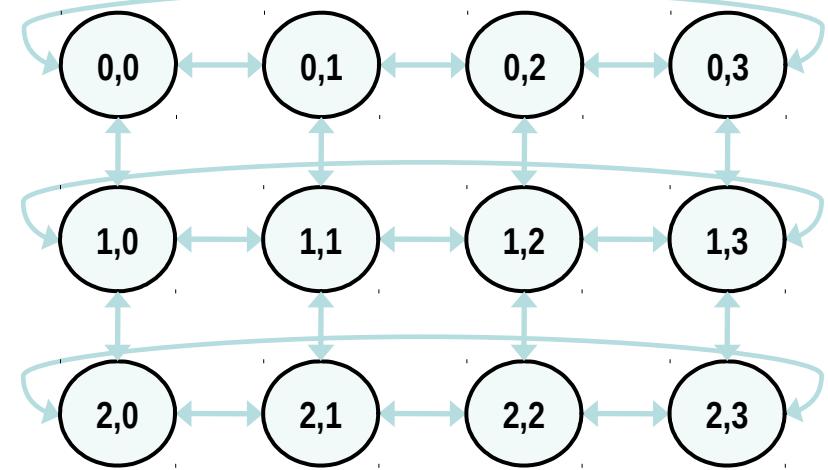
```

int dim[] = {4, 3};
int period[] = {1, 0};
MPI_Comm grid_comm;

MPI_Cart_create(MPI_COMM_WORLD, 2,
 dim, period, 0, &grid_comm);

int source, dest;
for (int dimension = 0; dimension < 2; dimension++) {
 for (int versus = -1; versus < 2; versus+=2) {
 MPI_Cart_shift(ring_comm, dimension, versus, &source, &dest);
 MPI_Ssendrecv(buffer, n, MPI_INT, source, stag,
 buffer, n, MPI_INT, dest, dtag,
 grid_comm, &status);
 }
}

```





# Destructors

The communicators and groups from a process' viewpoint are just handles.

Like all handles, there is a limited number available: you could (in principle) run out!

**MPI\_GROUP\_FREE**(group, ierr)

**MPI\_COMM\_FREE**(comm,ierr)