Introduction to Parallel Computing

Overview

- Parallel Computing
  - The name of the game
  - Programming models
  - Caches revisited
  - Parallel architectures
  - Multi-core everywhere
  - Hardware examples

Parallelism is everywhere

- In today's computer installations one has many levels of parallelism:
  - Instruction level (ILP)
  - Chip level (multi-core, multi-threading)
  - System level (SMP)
  - GP-GPUs
  - Grid/Cluster

What is Parallelization?

The Name of the Game

What is Parallelization?

An attempt of a definition:

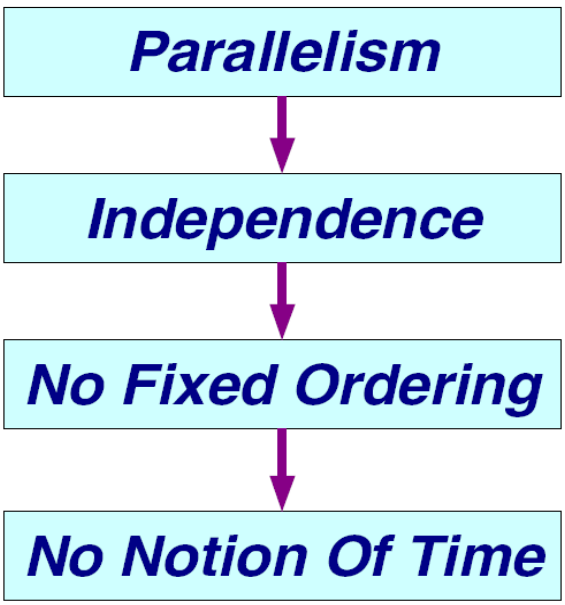
"*Something*" is parallel, if there is a certain level of independence in the order of operations

"*Something*" can be:

- A collection of program statements
- An algorithm
- A part of your program
- The problem you are trying to solve

granularity

Parallelism – when?



Something that does not follow this rule is not parallel !!!

Parallelism – Example 1

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

Every iteration in this loop is independent of the other iterations

Thread	T=1	T=2
1	a[1]=a[1]+b[1]	a[5]=a[5]+b[5]
2	a[2]=a[2]+b[2]	a[8]=a[8]+b[8]
3	a[3]=a[3]+b[3]	a[12]=a[12]+b[12]
4	a[4]=a[4]+b[4]	a[7]=a[7]+b[7]

Parallelism – Example 2

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

This operation is not parallel !

Proc	T=1	T=2
1	a[1]=a[2]+b[1]	a[4]=a[5]+b[4]
2	a[2]=a[3]+b[2]	a[8]=a[9]+b[8]
3	a[3]=a[4]+b[3]	a[12]=a[13]+b[12]
4	a[5]=a[6]+b[5]	a[7]=a[8]+b[7]

Parallelism – Results example 2

```
Results for P=1
   12512501.0
   12512501.0
   12512501.0
   12512501.0
Results for P=8
   12512508.0
   12512508.0
   12512508.0
   12512508.0
Results for P=32
   12512526.0
   12512530.0
   12512528.0
   12512527.0
Results for P=64
   12512548.0
   12512545.0
   12512549.0
   12512547.0
```

- parallel version of example 2 was run 4 times each on 1, 8, 32 and 64 threads/processors
- Output: sum over all elements of vector a
- Except for P=1, the results are:
  - Wrong
  - Inconsistent
  - NOT reproducable
- This is called a 'Data Race'

Parallelism

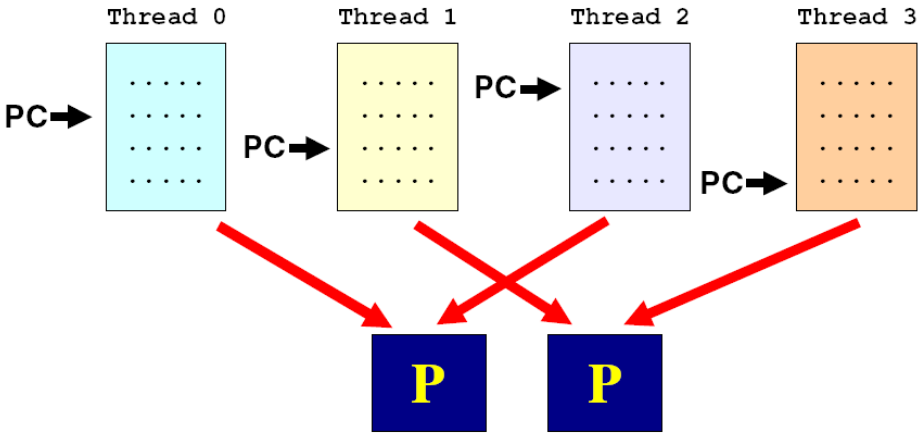
Fundamental problem:

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

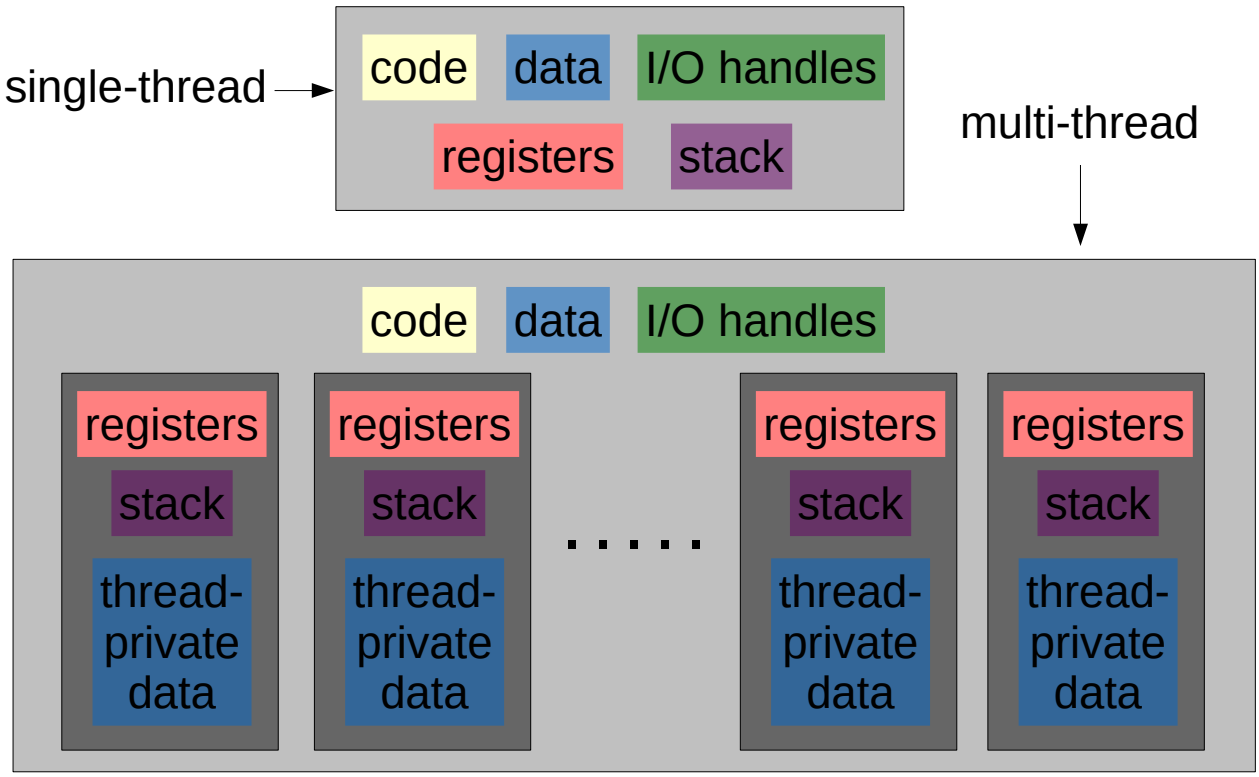
M = 0 : parallel
M >= 1 : not parallel

What is a Thread?

- Loosely said, a thread consists of a series of instructions with it's own program counter (“PC”) and state
- A parallel program will execute threads in parallel
- These threads are then scheduled onto processors



Single- vs. multi-threaded



The problem with threads

Prof. Eward A. Lee, Berkeley (2006):

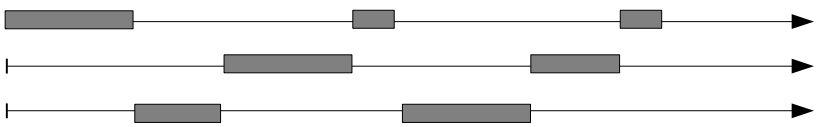
"I conjecture that most multi-threaded general-purpose applications are, in fact, so full of concurrency bugs that, as multicore architectures become commonplace, these bugs will begin to show up as system failures.

This scenario is bleak for computer vendors: their next generation of machines will become widely known as the ones on which many programs crash."

from: "The Problem with Threads", Technical Report No. UCB/EECS-2006-1

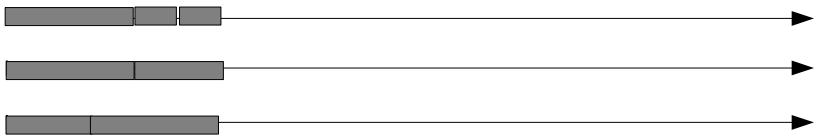
Parallelism vs Concurrency

Concurrent, non-parallel execution:

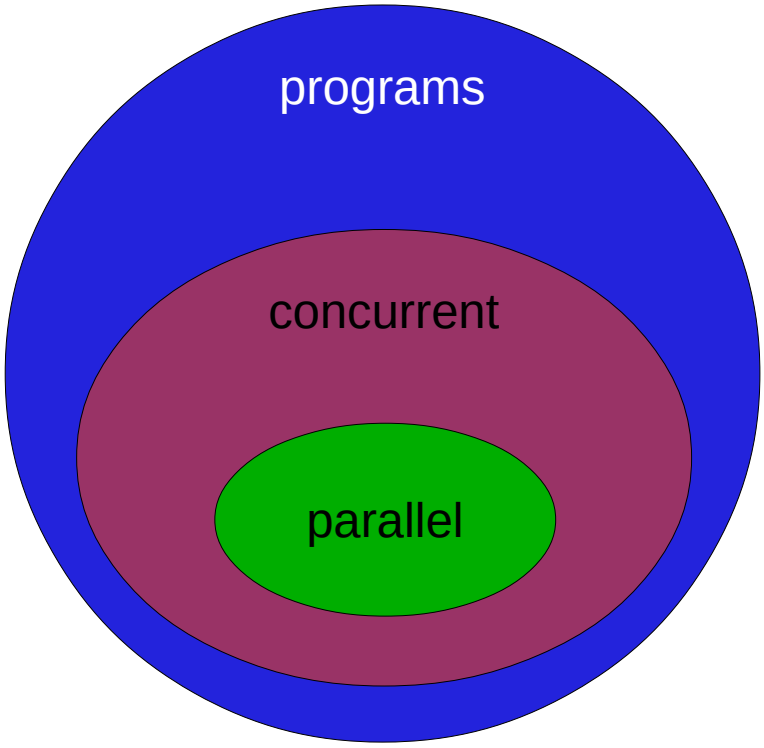


e.g. multiple threads on a single core CPU

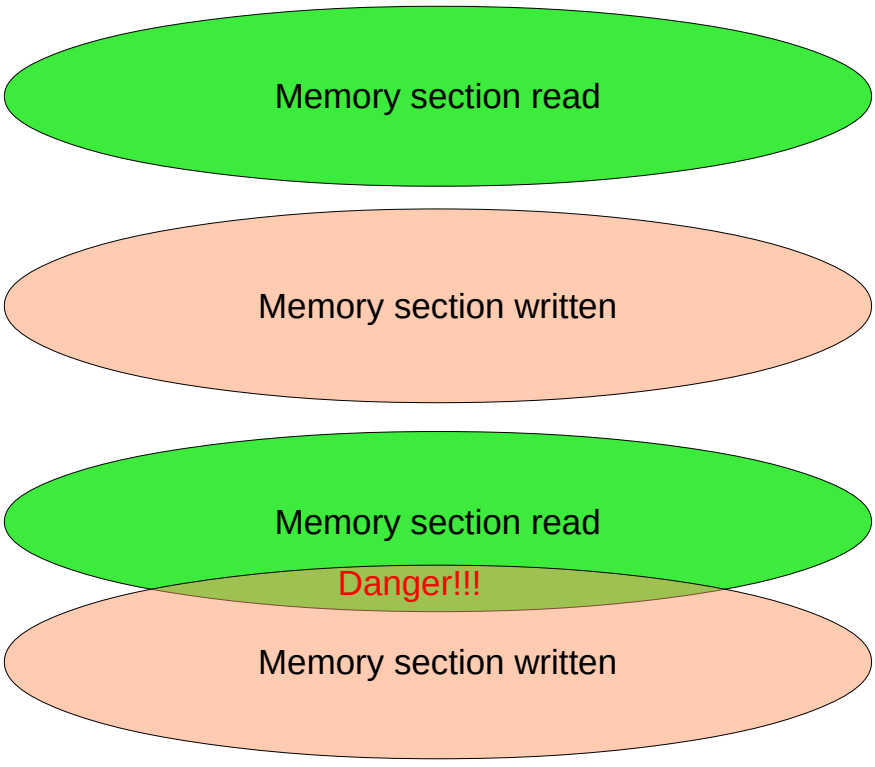
Concurrent, and parallel execution



Parallelism vs Concurrency



Parallelism – Memory Access



parallel

not parallel! -
... unless one can protect the overlap area

Parallelism – Data Races

- The update of shared variables is not well protected
- Race conditions can be nasty – and difficult to detect:
  - Numerical results differ (slightly) from run to run
  - Difficult to distinguish from numerical side effects
  - Changing the number of threads can make the problem disappear – or appear again
  - Shows very often first when using many threads, i.e. late in development

Parallelism – Data Race Detection

- A few data race detection tools are available:
  - Intel: ThreadChecker (now: Inspector XE)
    - license needed
  - Oracle: Solaris Studio Thread Analyzer
    - part of Solaris Studio 12.x
    - free
  - GCC: has a thread checker library
    - compiler option: -fsanitize-thread
- Those tools instrument your code, i.e. a detection run takes substantially longer

Numerical Results

Consider:

A = B + C + D + E

Serial Processing	Parallel Processing	
	Thread 0	Thread 1
A = B + C	T1 = B + C	T2 = D + E
A = A + D	T1 = T1 + T2	
A = A + E		

- ☞ *The roundoff behaviour is different and so the numerical results may be different too*
- ☞ *This is natural for parallel programs, but it may be hard to differentiate it from an ordinary bug ....*

Basic concepts

- Consider the following code with two loops

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

```
for (i = 0; i < n; i++)
    d[i] = a[i] + e[i];
```

- Running this in parallel over i might give the wrong answer.

Basic concepts – the barrier

- The problem can be fixed:

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

wait!

```
for (i = 0; i < n; i++)
    d[i] = a[i] + e[i];
```

- The barrier assures that no thread starts working on the second loop before the work on loop one is finished.

Basic concepts – the barrier

When to use barriers?

- To assure data integrity, e.g.
  - after one iteration in a solver
  - between parts of the code that read and write the same variables
- Barriers are expensive and don't scale to a large number of threads

Basic concepts – reduction

- A typical code fragment:

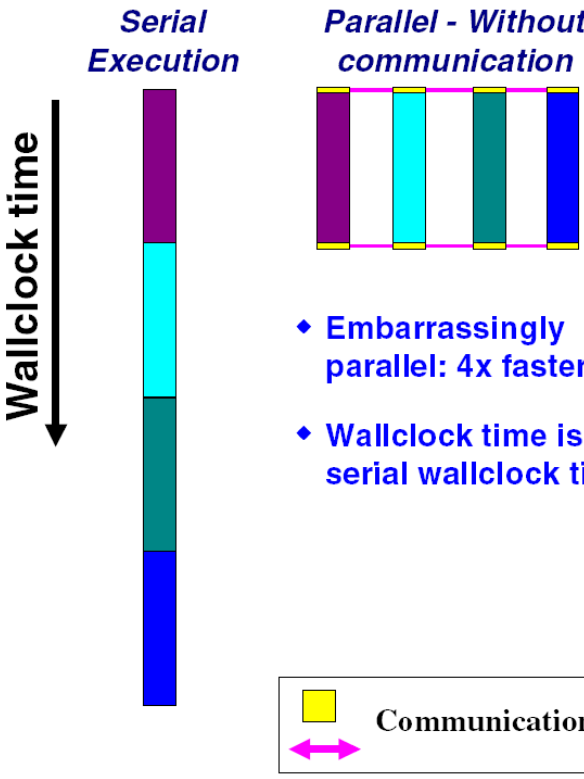
```
for( i = 0; i < n; i++ ) {
    ...
    sum += a[i];
    ...
}
```

- This loop can not run in parallel, unless the update of sum is protected. ☞ serial code
- An operation like the above is called a “reduction” operation, and there are ways to handle this issue (more later...).

Parallel Overhead

- The total CPU time may exceed the serial CPU time:
  - The newly introduced parallel portions in your program need to be executed
  - Threads need time for sending data to each other and for synchronizing (“communication”)
- Typically, things also get worse when increasing the number of threads
- Efficient parallelization is about minimizing the communication overhead

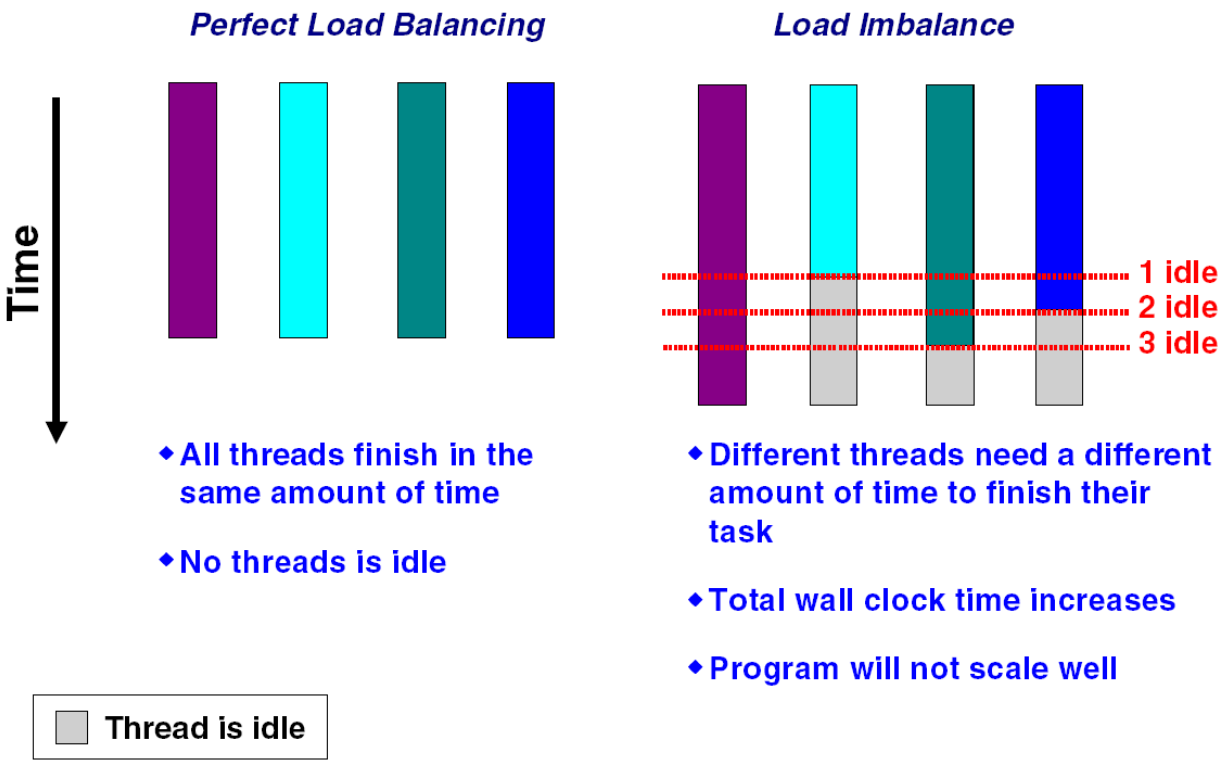
Communication



- Embarrassingly parallel: 4x faster
- Wallclock time is ¼ of serial wallclock time

- Additional communication
- Less than 4x faster
- Consumes additional resources
- Wallclock time is more than ¼ of serial wallclock time
- Total CPU time increases

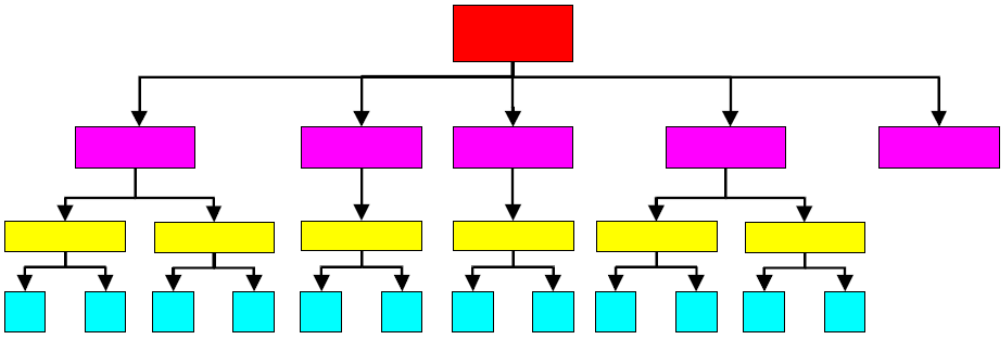
Load Balancing



- All threads finish in the same amount of time
- No threads is idle

- Different threads need a different amount of time to finish their task
- Total wall clock time increases
- Program will not scale well

Dilemma – Where to parallelize?



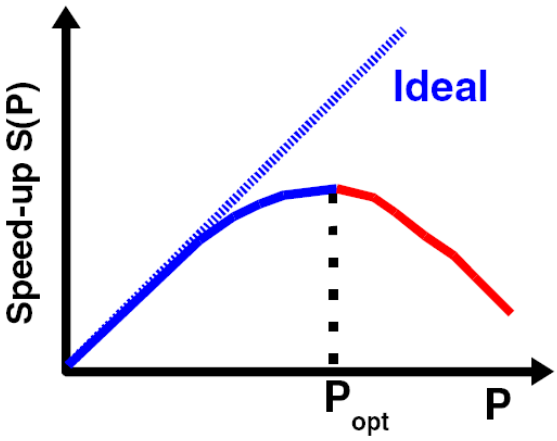
- *Parallelization at the highest (■) level:*
  - *Low communication cost*
  - *Limited to 5 processors only*
  - *Potential load balancing issue*
- *Parallelization at the lowest (■) level:*
  - *Higher communication cost*
  - *Not limited to a certain number of processors*
  - *Load balancing probably less of an issue*

Scalability

We distinguish ...

- ... how well a solution to some problem will work when the size of the problem increases.
  - typically associated with algorithmic complexity
- ... how well a parallel solution to some problem will work when the number of processing units (PUs) increases.
  - Strong scaling (speed-up) or weak scaling

Scalability – speed-up & efficiency



- *Define the speed-up S(P) as S(P) := T(1)/T(P)*
- *The efficiency E(P) is defined as E(P) := S(P)/P*
- *In the ideal case, S(P)=P and E(P)=P/P=1=100%*
- *Unless the application is embarrassingly parallel, S(P) will start to deviate from the ideal curve*
- *Past this point P_{opt}, the application will get less and less benefit from adding processors*
- *Note that both metrics give no information on the actual run-time*
- *As such, they can be dangerous to use*

Amdahl's Law

Assume our program has a parallel fraction "f"

This implies the execution time $T(1) := f*T(1) + (1-f)*T(1)$

On P processors: $T(P) = (f/P)*T(1) + (1-f)*T(1)$

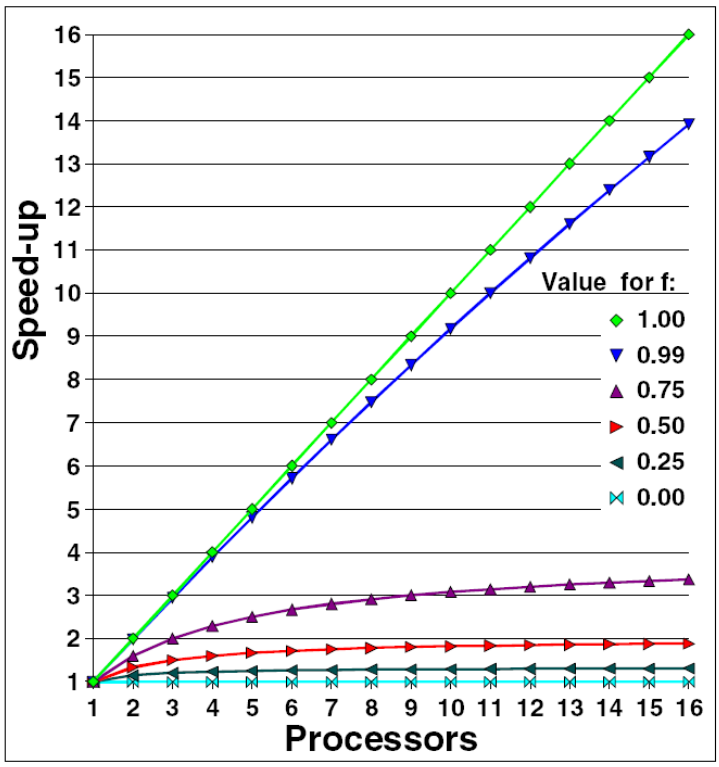
Amdahl's law:

$$S(P) := T(1) / T(P) = 1 / (f/P + 1-f)$$

Comments:

- *This "law' describes the effect that the non-parallelizable part of a program has on scalability*
- *Note that the additional overhead caused by parallelization and speed-up because of cache effects are not taken into account*

Amdahl's Law



- *It is easy to scale on a small number of processors*
- *Scalable performance however requires a high degree of parallelization i.e. f is very close to 1*
- *This implies that you need to parallelize that part of the code where the majority of the time is spent*
- *Use the performance analyzer to find these parts*

Amdahl's Law in Practice

We can estimate the parallel fraction "f"

Recall: *T(P) = (f/P)*T(1) + (1-f)*T(1)*

It is trivial to solve this equation for "f":

$$f = (1 - T(P)/T(1))/(1 - (1/P))$$

Example:

```
T(1) = 100 and T(4)=37 => S(4) = T(1)/T(4) = 2.70
f = (1-37/100)/(1-(1/4)) = 0.63/0.75 = 0.84 = 84%
```

Estimated performance on 8 processors is then:

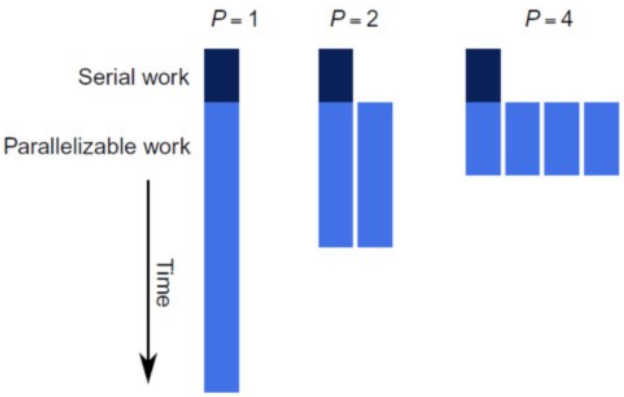
```
T(8) = (0.84/8)*100 + (1-0.84)*100 = 26.5
S(8) = T/T(8) = 3.78
```

Scaling: strong vs. weak

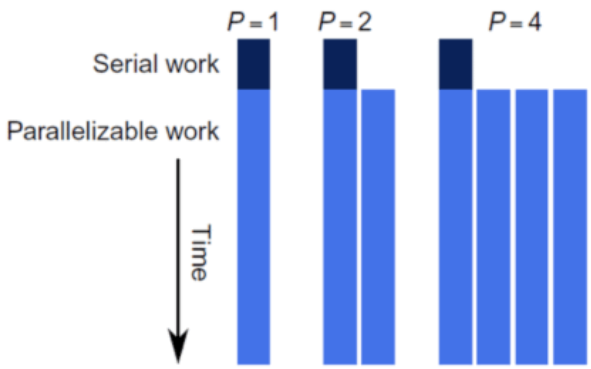
- How does the execution time go down for a fixed problem size by increasing the number of PUs?
  - Amdahl's law ⇒ speed-up, i.e. reduce time
  - also known as “strong scaling”
- How much can we increase the problem size by adding more PUs, keeping the execution time approx. constant?
  - Gustafson's law ⇒ scale-up, i.e. increase work
  - also known as “weak scaling”

Amdahl's vs Gustafson's law

Amdahl: fixed work



Gustafson: fixed work/PU



Amdahl's vs Gustafson's Law

- Amdahl's law
  - Theoretical performance of an application with a ***fixed amount of parallel work*** given a particular number of Processing Units (PUs)
- Gustafson's Law:
  - Theoretical performance of an application with a ***fixed amount of parallel work per PU*** given a particular number of PUs

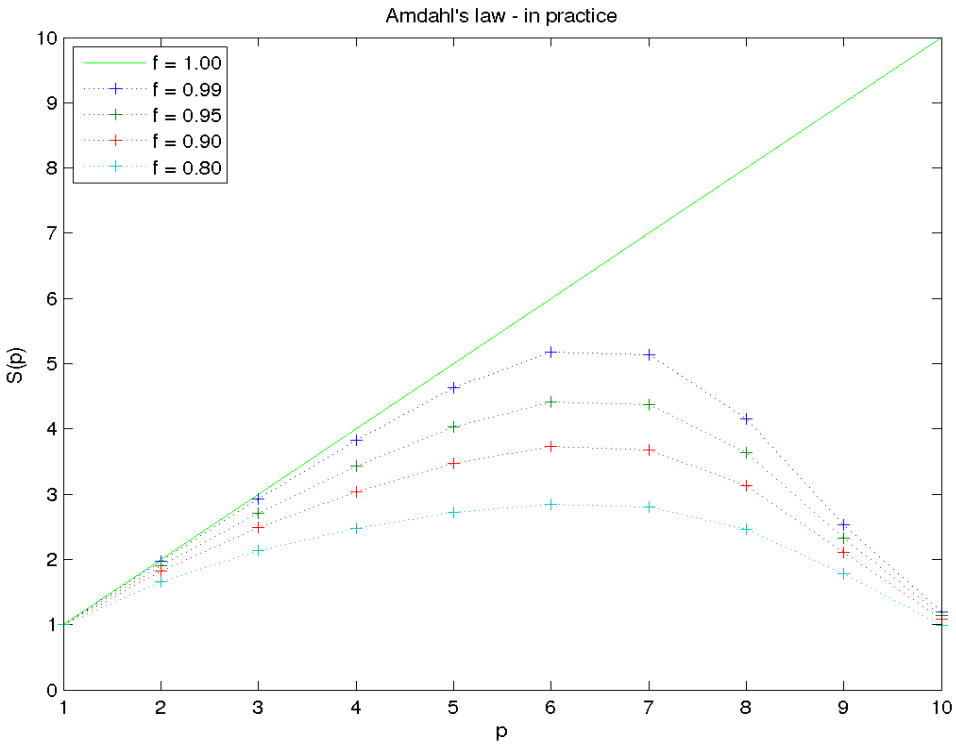
Code scalibility in practice – I

- Although Amdahl and Gustafson provide theoretical upper bounds, eventually real data are necessary for analysis
- Inconsistencies in performance – especially on shared systems – often appear in singular runs
- Best practice: Monitor codes several times and average the results to filter out periods of heavy usage due to other users

Code scalibility in practice – II

- Ideally, HPC codes would be able to scale to the theoretical limit, but ...
  - Never the case in reality
  - All codes eventually reach a real upper limit on speedup
  - At some point codes become “bound” to one or more limiting hardware factors (memory, network, I/O)

Code scalibility in practice – III



What is Parallelization? - Summary

- Parallelization is simply another optimization technique to get your results sooner
- To this end, more that one processor is used to solve the problem
- The “Elapsed Time” (also called wallclock time) will come down, but total CPU time will probably go up
- The latter is a difference with serial optimization, where one makes better use of existing resources, i.e. the cost will come down

Parallel Programming Models

Parallel Programming Models

Two “classic” parallel programming models:

- Distributed memory
  - PVM (standardized)
  - MPI (de-facto standard, widely used)
    - http://mpi-forum.org or http://open-mpi.org/
- Shared memory
  - Pthreads (standardized)
  - C++11 threads
  - OpenMP (de-facto standard) http://openmp.org/
  - Automatic parallelization (depends on compiler)

Clusters, SMPs

SMP only

Parallel Programming Models

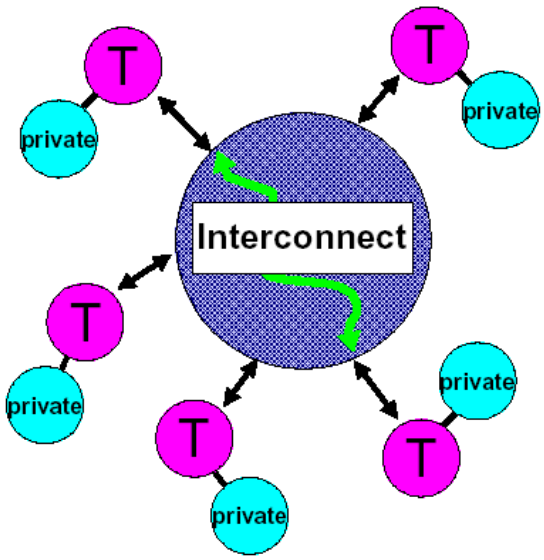
Other programming models

- PGAS (Partitioned Global Address Space):
  - UPC (Unified Parallel C)
  - Co-Array Fortran
- GPUs: massively parallel & shared memory
  - CUDA
  - OpenCL

Parallel Programming Models

Distributed memory programming model, e.g. MPI:

- all data is private to the threads
- data is shared by exchanging buffers
- explicit synchronization



Parallel Programming Models

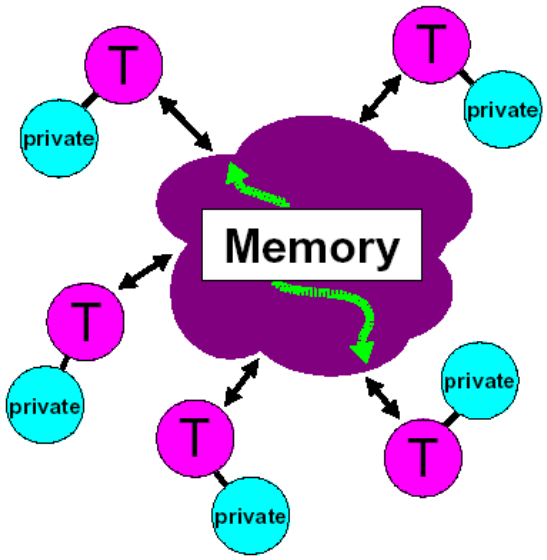
MPI:

- An MPI application is a set of independent processes (threads)
  - on different machines
  - on the same machine
- communication over the interconnect
  - network (network of workstations, cluster, grid)
  - memory (SMP)
- communication is under control of the programmer

Parallel Programming Models

Shared memory model, e.g. OpenMP:

- all threads have access to the same global memory
- data transfer is transparent to the programmer
- synchronization is (mostly) implicit
- there is private data as well



Parallel Programming Models

OpenMP:

- needs an SMP
- but ... with newer CPU designs, there is an SMP in (almost) every computer
  - multi-core CPUs (CMP)
  - chip multi-threading (CMT)
  - or a combination of both
  - or ... (whatever we'll see in the future)

MPI vs OpenMP

OpenMP version of “Hello world”:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
    printf("Hello world!\n");
    } /* end parallel */
    return(0);
}
```


MPI vs OpenMP

```
% suncc -o hello -xopenmp hello.c
% ./hello
Hello world!

% OMP_NUM_THREADS=2 ./hello
Hello world!
Hello world!

% OMP_NUM_THREADS=8 ./hello
Hello world!
Hello world!
Hello world!
Hello world!
```

This behaviour is implementation dependent!

no. of threads: OMP_NUM_THREADS

MPI vs OpenMP

MPI version of “Hello world”:

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

int main(int argc, char **argv) {
    int myrank, p;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    printf("Hello world from %d!\n", myrank);
    MPI_Finalize();
    return 0;
}
```


MPI vs OpenMP

MPI version: compile and run

```
$ module load mpi
$ mpicc -o hello_mpi hello_mpi.c

$ ./hello_mpi
Hello world from 0!

$ mpirun -np 4 ./hello_mpi
Hello world from 1!
Hello world from 3!
Hello world from 0!
Hello world from 2!
```

Automatic Parallelization

- Some compilers are capable of generating parallel code for loops that can be safely executed in parallel.
- This is always loop based!
- Compilers:
  - Intel compilers: -parallel
  - Oracle/Sun compilers: -xautopar
  - GCC: -floop-parallelize-all -ftree-parallelize-loops=N
    - limited, sets no. of threads (N) at compile time!
- For more information see the manual pages

Automatic Parallelization

Loop based parallelization:

- different iterations are carried out in parallel
- same program can run on “any” number of threads
- # threads: set OMP_NUM_THREADS=N
- the compiler generates a serial version as well, e.g. for small problem sizes (short loops)
- threshold might be a compiler option (e.g. Intel)

Automatic Parallelization – Ex. 1

Examples with Sun Studio compilers:

```
 1 void
 2 auto1(int n, double a, double *x, double *y) {
 3     int i;
 4     for (i=0; i<n; i++)
 5         x[i] += a*y[i];
 6 }
```

```
$ cc -c -g -fast -xrestrict -xautopar -xloopinfo auto1.c
"auto1.c", line 4: PARALLELIZED, and serial version
generated
```


Automatic Parallelization – Ex. 2

```
 1 subroutine auto2(n,x,sum)
 2 implicit none
 3 integer :: n
 4 real(kind=8):: a, x(1:n,1:n), sum
 5 integer :: i, j
 6
 7 sum = 0.0
 8 do i = 1, n
 9    do j = 1, n
10       sum = sum + x(i,j)
11    end do
12 end do
13 return
14 end
```

```
$ f95 -g -fast -xautopar -xloopinfo -c auto2.f95
"auto2.f95", line 8: not parallelized, unsafe dependence
 (sum),interchanged
"auto2.f95", line 9: not parallelized, unsafe dependence
 (sum),interchanged
```

Automatic Parallelization – Ex. 2a

```
 1 subroutine auto2(n,x,sum)
 2 implicit none
 3 integer :: n
 4 real(kind=8):: a, x(1:n,1:n), sum
 5 integer :: i, j
 6
 7 sum = 0.0
 8 do i = 1, n
 9    do j = 1, n
10       sum = sum + x(i,j)
11    end do
12 end do
13 return
14 end
```

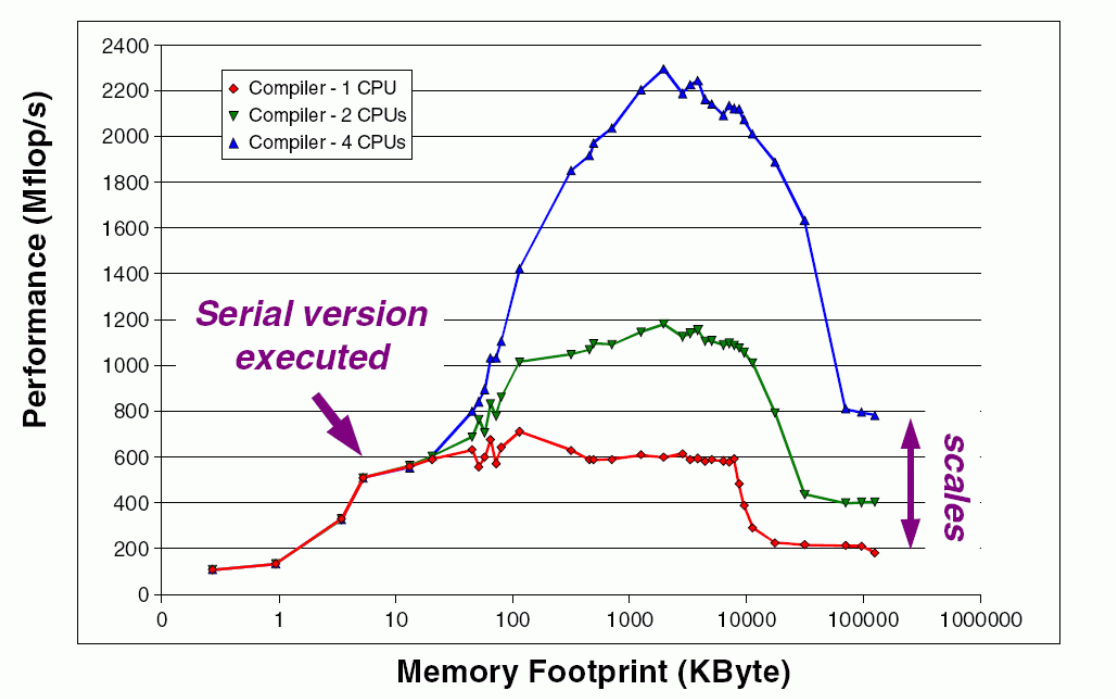
```
$ f95 -g -fast -xautopar -xloopinfo -xreduction -c auto2.f95
"auto2.f95",line 8: PARALLELIZED, reduction, and serial
version generated
"auto2.f95",line 9: not parallelized, not profitable
```


Automatic Parallelization – Ex. 3

```
 1 void
 2 mxv(int m, int n, double * restrict a,
       double * restrict b, double * restrict c) {
 3
 4    int i, j;
 5    double sum;
 6
 7    for (i=0; i<m; i++) {
 8       sum = 0.0;
 9       for (j=0; j<n; j++)
10            sum += b[i*n+j]*c[j];
11        a[i] = sum;
12    }
13 }
```

```
$ suncc -g -fast -xautopar -xloopinfo -c mxv.c
"mxv.c", line 7: PARALLELIZED, and serial version generated
"mxv.c", line 9: not parallelized, unsafe dependence(sum)
```

Automatic Parallelization – Ex. 3



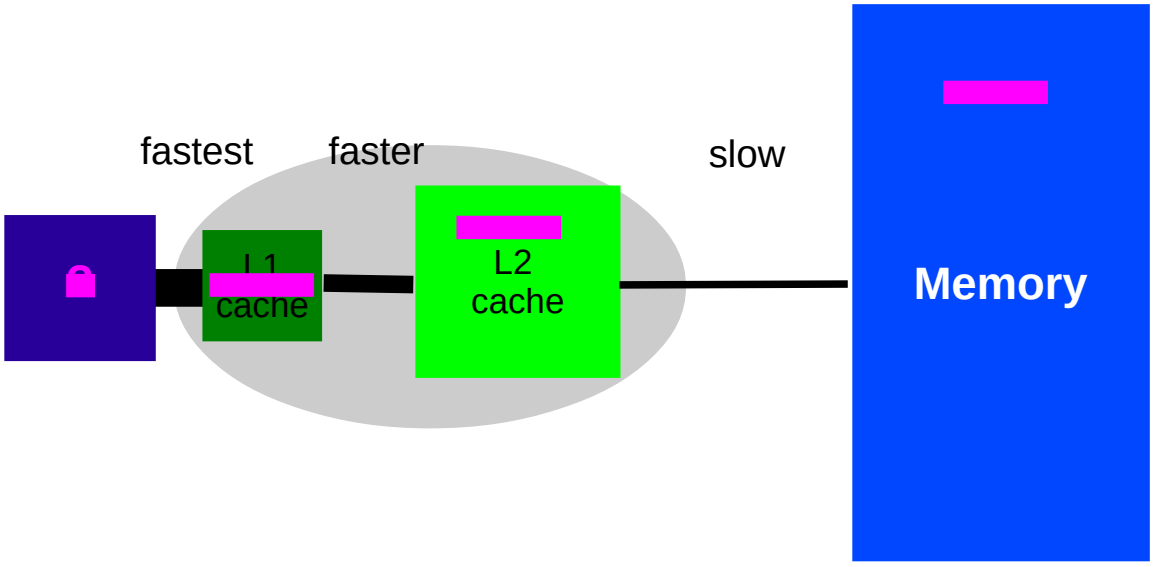
Automatic Parallelization

Best practice (Studio compiler):

- compile and link with -xautopar
- use -xloopinfo to see compiler messages
- -xreduction will allow the compiler to apply reduction operations
- apply automatic parallelization on the most time consuming parts of the code only (efficiency!)

Caches revisited

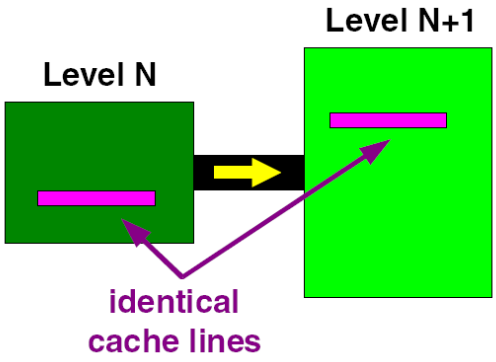
Typical cache based system



How are caches organized?

- Caches contain partial images of memory
- If data gets modified, the state of that data, i.e. the whole cache line, changes
- This has to be made known to the system
- There are two common approaches:
  - Write-through
  - Write-back

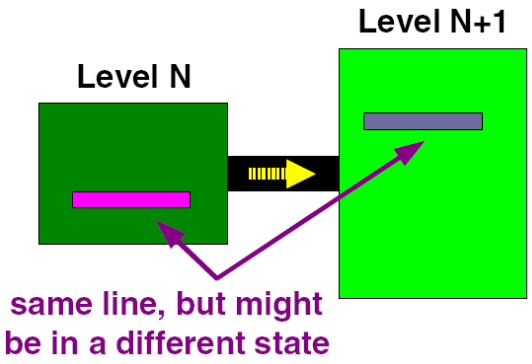
Write-through cache



Notes:
- simple to implement
- easy to find the right copy
- can result in waste of bandwidth

- Always flushes a modified cache line back to a higher level in the memory hierarchy
  - e.g. from L1-cache to main memory
- This assures, that the system always knows where to access the correct cache line

Write-back cache



Notes:
- minimizes cache traffic
- need to keep track of status
- this mechanism is called *'cache coherency'*

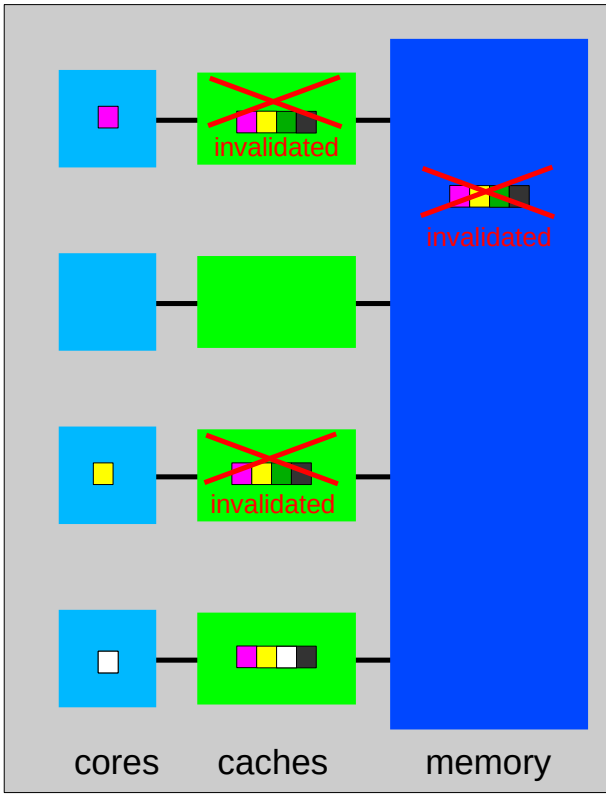
- Write a modified cache line back only if needed
  - capacity issues
  - another cache line maps onto this line
  - another CPU might need this cache line

Caches in MP/multi-core systems

- A cache line always starts in memory
- Over time multiple copies may exist

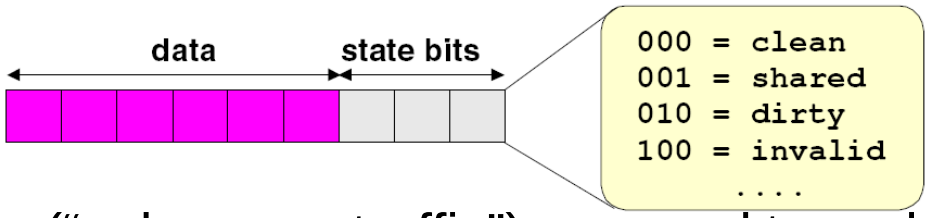
Cache coherency ('cc')

- tracks changes in copies
- assures correct cache line is used
- many implementations
- hardware support to be efficient



Cache coherency ('cc')

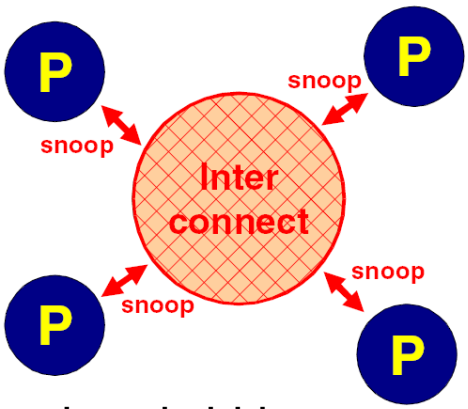
- Needed in write-back cache systems
- Keeps track of the status of all cache lines
- State information:



- Signals ("coherency traffic") are used to update the state bits of the cache lines
- This allows to build efficient SMP systems

Snoopy based cache coherence

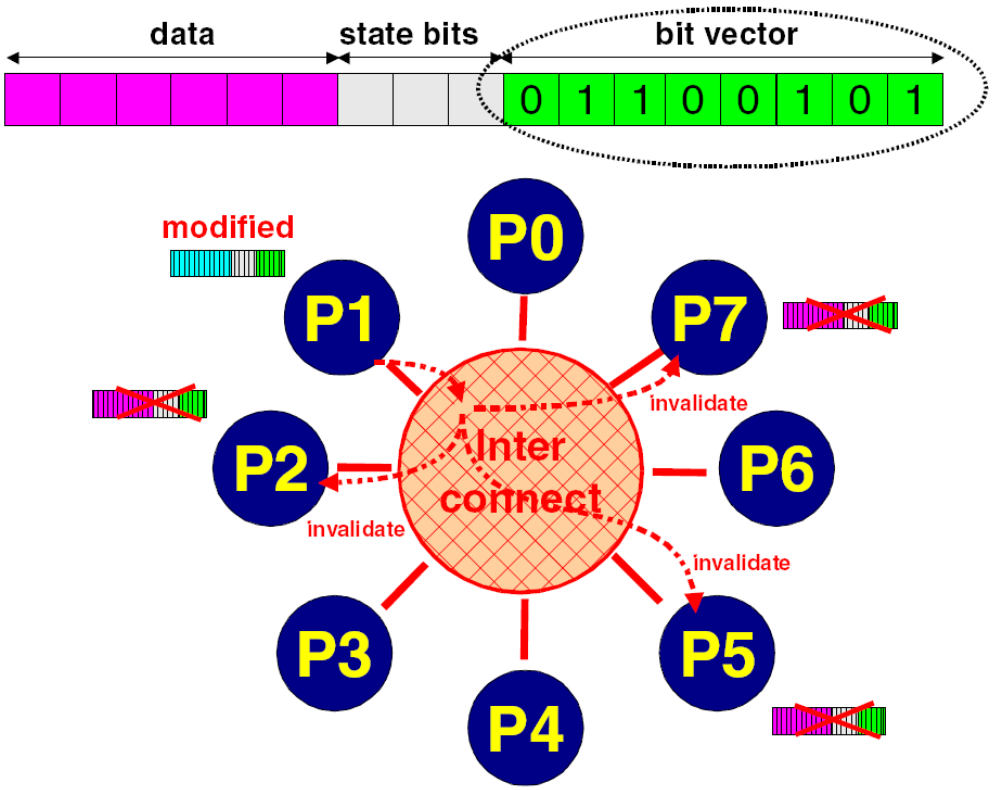
- Also known as "broadcast cache coherence"
  - all addresses are sent to all CPUs
  - result takes only a few cycles
- Advantages:
  - low latency
  - fast cache-to-cache transfer
- Disadvantages:
  - data bandwidth limited by snoop bandwidth
  - difficult to scale to many CPUs



Directory based cache coherence

- Also known as SSM (Scalable Shared Memory)
- point-to-point protocol, i.e. a directory keeps track which CPUs are involved with a particular cache line
- requests are sent to the involved CPUs only
- Advantages:
  - larger bandwidth & scalable to many CPUs
- Disadvantages:
  - longer & non-uniform latency
  - slower cache-to-cache transfer
  - need to store the directory entries

SSM example:

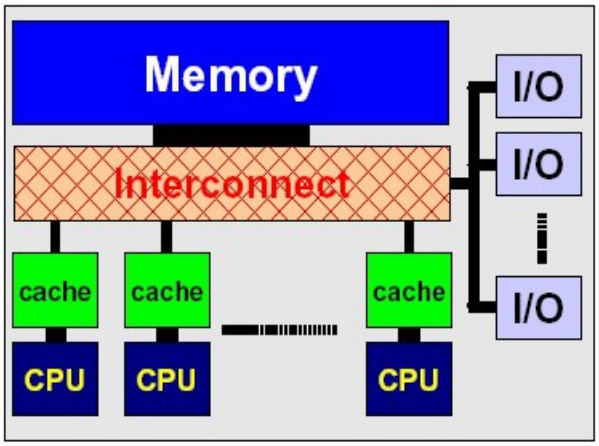


Parallel Architectures

Parallel Architectures

- It is difficult to label systems:
  - most systems share some characteristics, but not all
  - the variety of systems is increasing
- In the (“*historical*”) overview presented here, systems are labelled based on main memory:
  - Shared or Distributed:
    - can all CPUs access all memory, or only a subset?
  - Memory access times
    - uniform vs non-uniform

Uniform Memory Access (UMA)

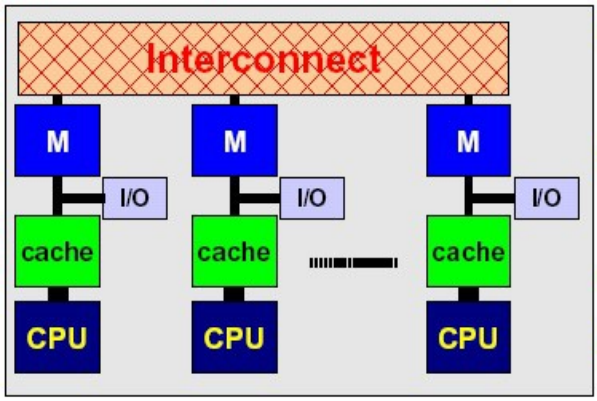


- Also called "SMP" (Symmetric Multi Processor)
- Memory Access time is Uniform for all CPUs
- Interconnect is "cc":
  - Bus
  - Crossbar
- No fragmentation - Memory and I/O are shared resources

Pro
- ✔ Easy to use and to administer
- ✔ Efficient use of resources

Con
- ✔ Said to be expensive
- ✔ Said to be non-scalable

Non-Uniform Memory Access (NUMA)

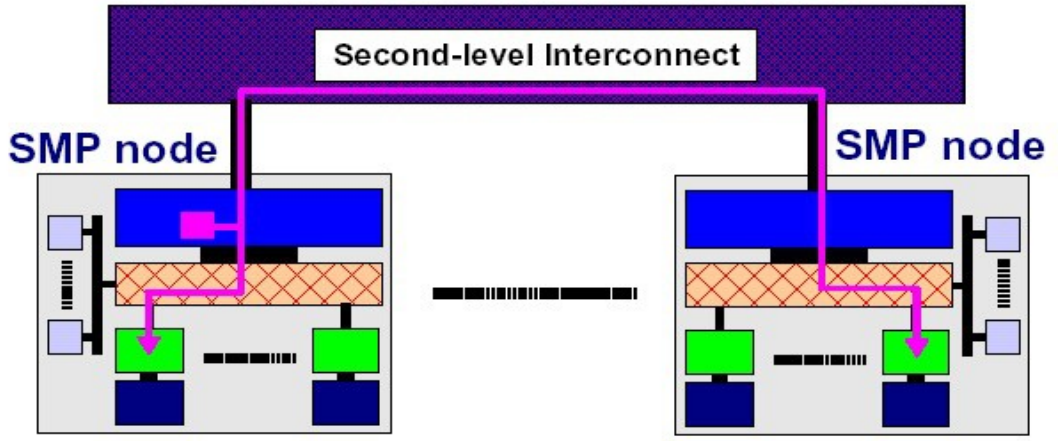


- Also called "Distributed Memory"
- Memory Access time is Non-Uniform
- Hence the name "NUMA"
- Interconnect is not "cc":
  - Ethernet, ATM, Myrinet
  - .....
- Runs 'N' copies of the OS
- Memory and I/O are distributed resources

Pro
- ✔ Said to be cheap
- ✔ Said to be scalable

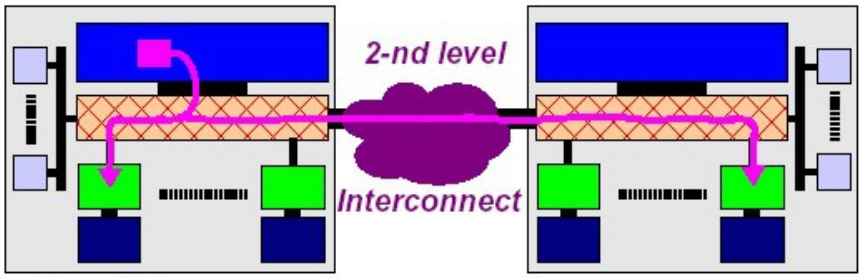
Con
- ✔ Difficult to use and administer
- ✔ In-efficient use of resources

Cluster of SMP Nodes



- *Second-level interconnect is not cache coherent*
  - *Ethernet, ATM, Myrinet, ...*
- *Hybrid Architecture with all Pros and Cons:*
  - *UMA within one SMP node*
  - *NUMA across nodes*

cc-NUMA



- *Two-level interconnect:*
  - *UMA/SMP within one system*
  - *NUMA between the systems*
- *Both interconnects support cache coherency i.e. the system is fully cache coherent*
- *Has all the advantages ('look and feel') of an SMP*
- *Downside is the Non-Uniform Memory Access time*

Programming Models Revisited

Architecture	Shared Memory Efficient ?	Distributed Memory Efficient ?
UMA/SMP	yes	yes (very !)
NUMA	not available	maybe *
Cluster of SMPs	yes (within one node)	maybe *
cc-NUMA	depends	yes

- ✔ *One can map any programming model onto any architecture*
- ✔ *Making it efficient is the key problem to solve*

**) Depends on interconnect*

Programming Models Revisited

- ❑ *The question whether an application is parallel, or not, has nothing to do with the programming model*
- ❑ *Two possibilities (for the time consuming part):*
  - ❶ *If parallel, decide on the programming model:*
    - ☞ *Message Passing*
      - ◆ *Do It Yourself*
    - ☞ *Shared Memory*
      - ◆ *Use the compiler*
      - ◆ *May need directives to assist the compiler*
  - ❷ *If not parallel: Try to rewrite or change the algorithm and go back to step ❶*

Multi-core – everywhere!

Welcome to a “threaded” world

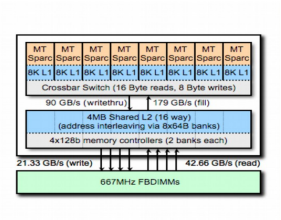
The first multi-core chips

- 2004 – multi-core arrives:
  - IBM POWER5
  - Sun UltraSPARC-IV
- 2005 is the year of the x86 dual-core CPUs:
  - AMD Opteron “Denmark” (August 2005)
  - Intel Xeon “Paxville DP” (October 2005)
- 2008/2009: quad-cores
  - AMD Opteron 'Barcelona'
  - Intel Xeon 'Nehalem'

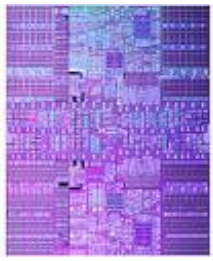
2009's Multi-cores

99% of Top500 Systems Are Based on Multi-core

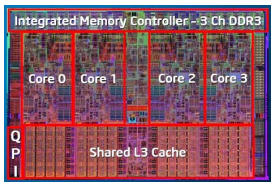
282 use Quad-Core
204 use Dual-Core
3 use Nona-core



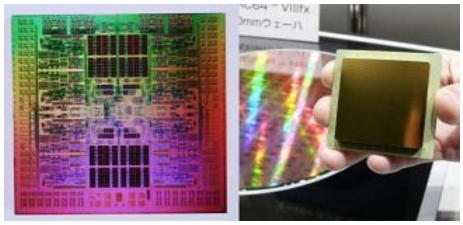
Sun Niagara2 (8 cores)



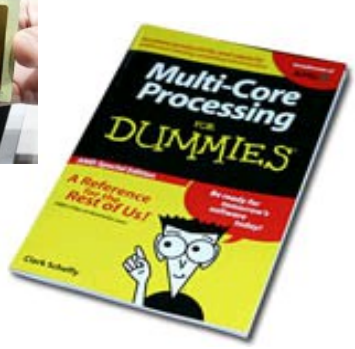
IBM Power 7 (8 cores)

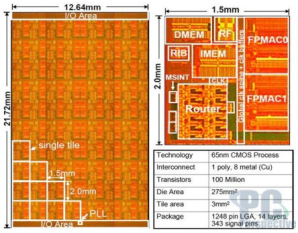


Intel Nehalem (4 cores)

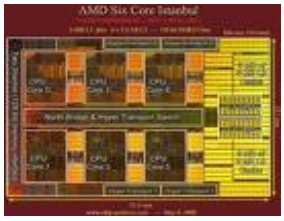


Fujitsu Venus (8 cores)

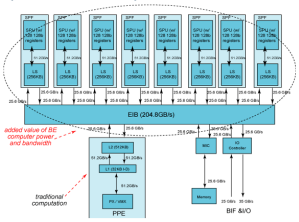




Intel Polaris [experimental] (80 cores)

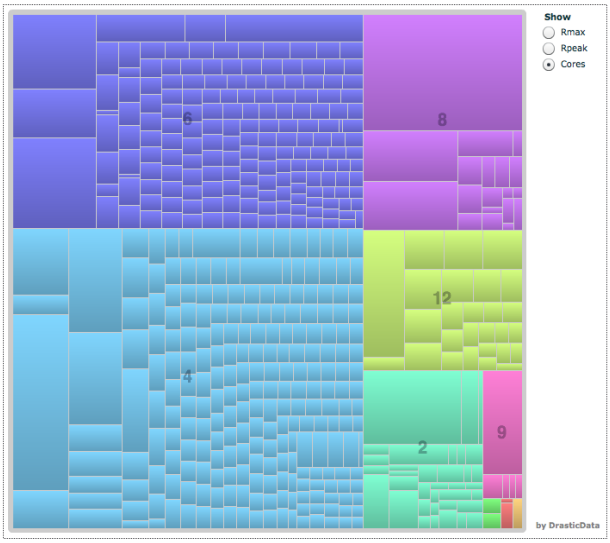


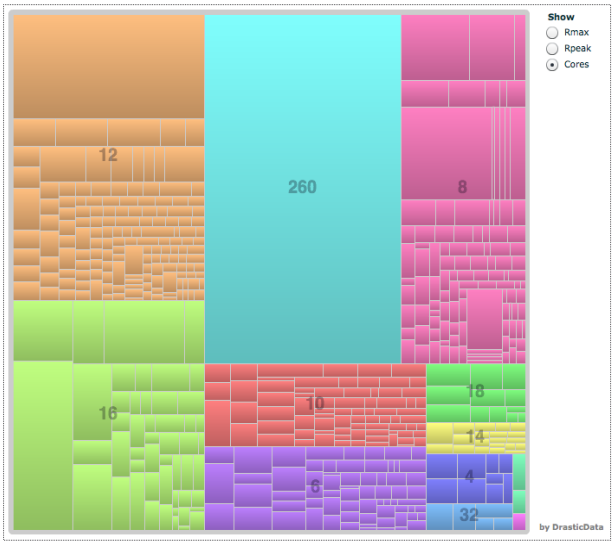
AMD Istanbul (6 cores)



IBM Cell (9 cores)

TOP500: multi-cores 2011 and 2016

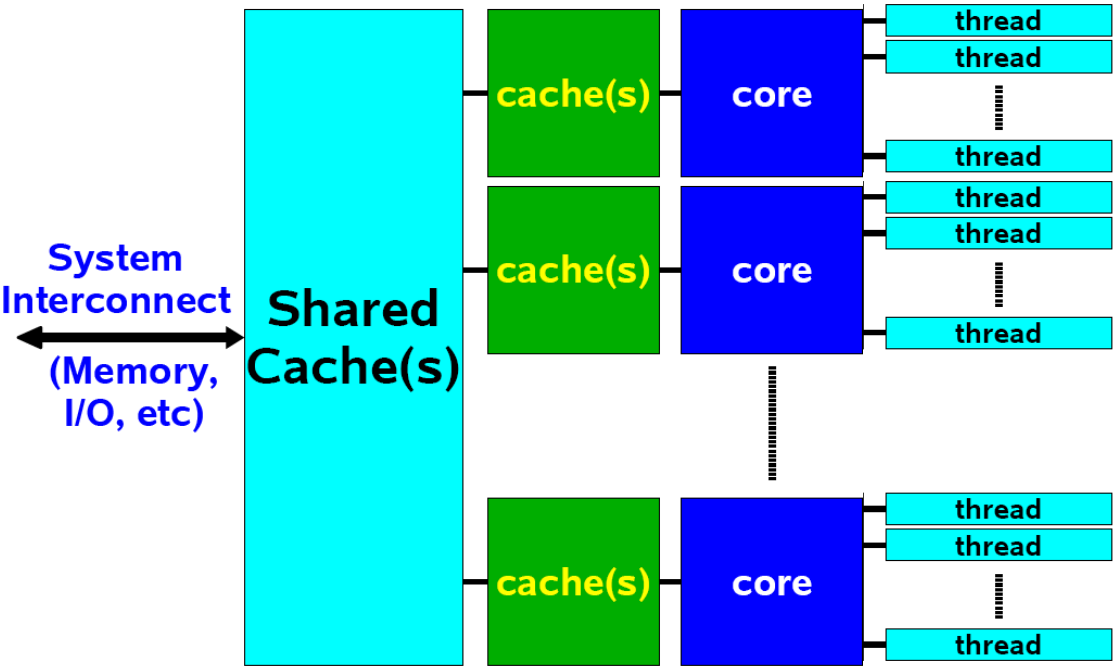




What is a multi-core chip?

- A “core” is not well-defined – let us assume it covers the processing units and the L1 caches (a very simplified CPU).
- Different implementations are possible – and available (examples follow), e.g. multi-threaded cores
- Cache hierarchy of private and shared caches
- For software developers it matters that there is parallellism in the hardware, they can take advantage of

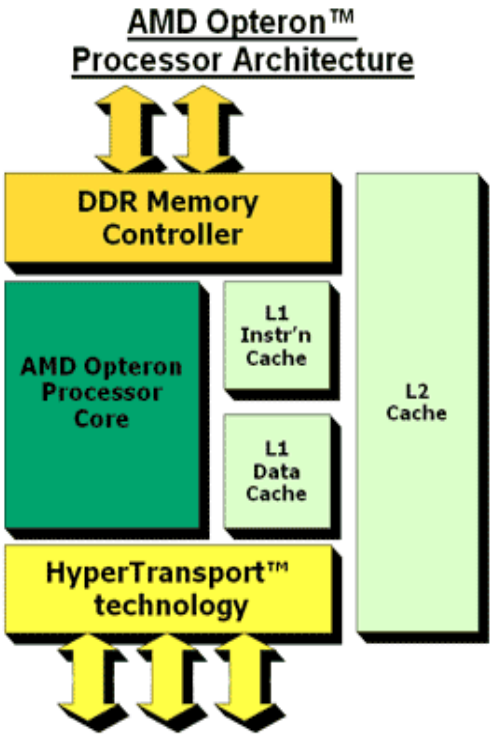
A generic multi-core design



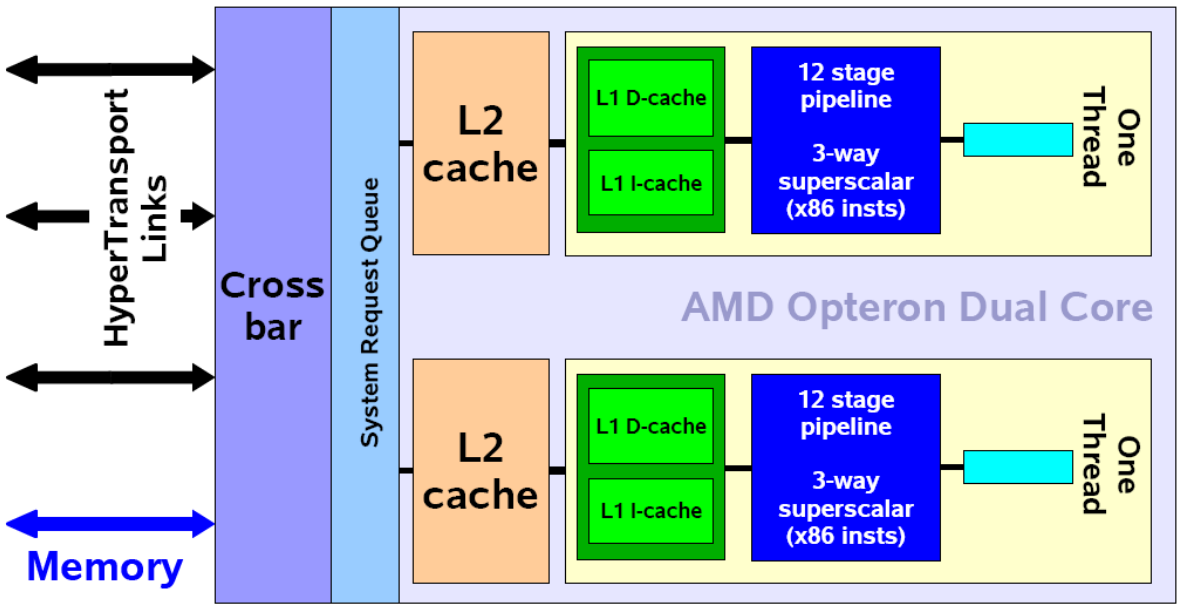
The AMD Opteron – single core

On-chip:

- Memory controller
- L2 cache
- 3 fast HyperTransport links: 6.4 GB/s per link

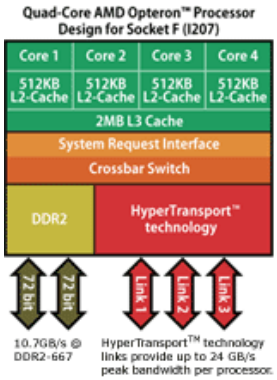


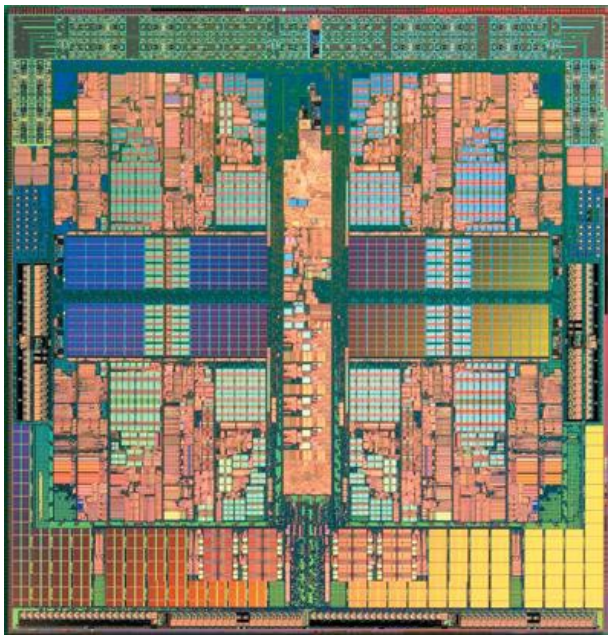
AMD Opteron - dual-core



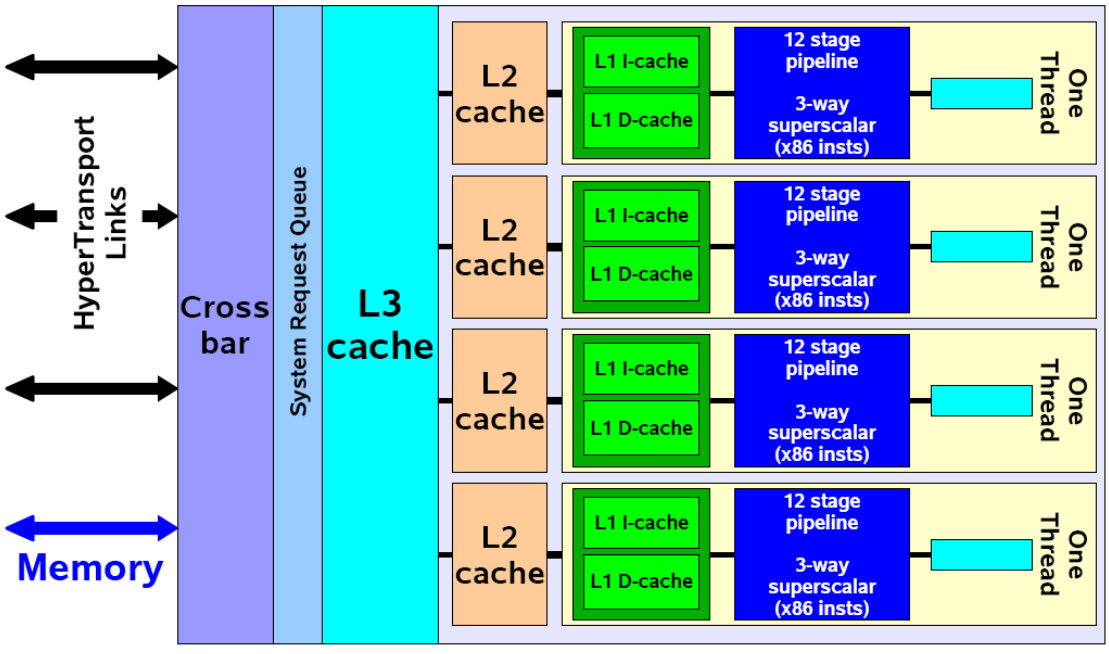
AMD Opteron – quad-core

- dedicated L2 caches
- shared L3 cache



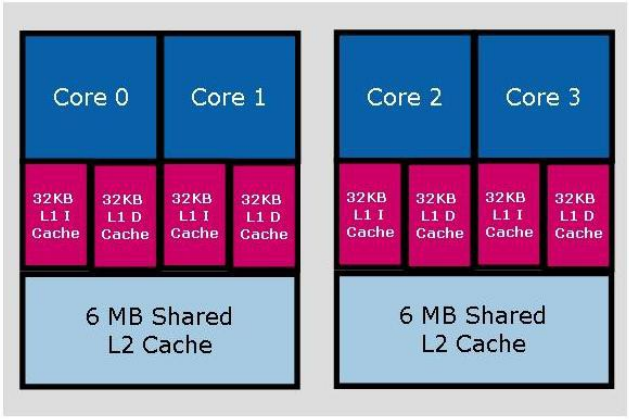


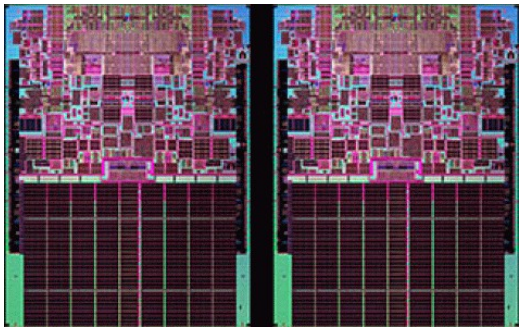
AMD Opteron – quad-core



Quad-core Intel Xeon

Intel® Xeon® processor 5400 series
(Codename 'Harpertown)

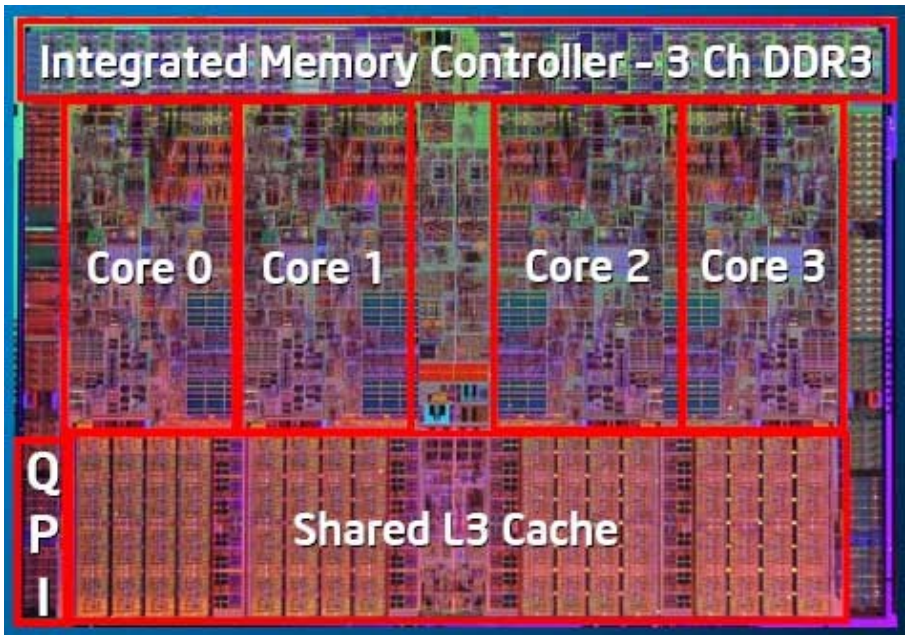




Two dual-core chips "glued" together

The Intel "Nehalem" CPU

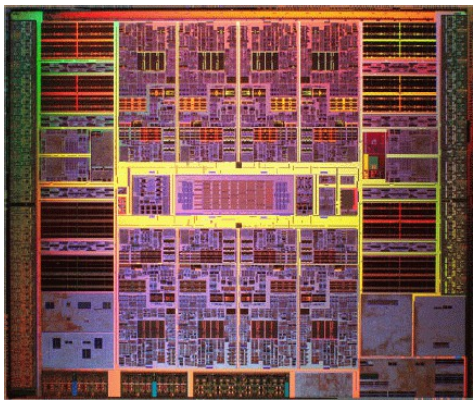
First quad-core CPU with QPI



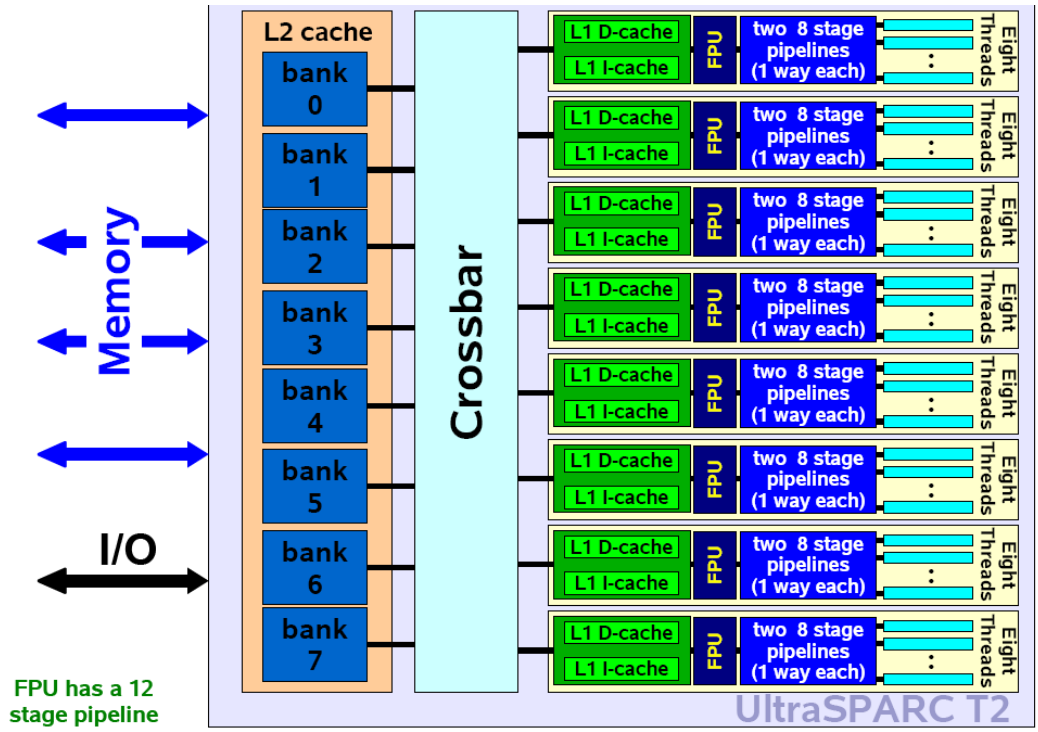
UltraSPARC-T2

System on a chip:

- 8 cores with 8 threads = 64 threads
- integrated multi-threaded 10 Gb/s Ethernet
- integrated crypto-unit per core
- low power (< 95W)
- < 1.5W/thread

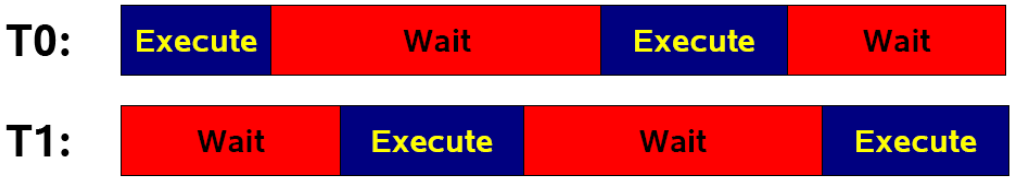


UltraSPARC-T2

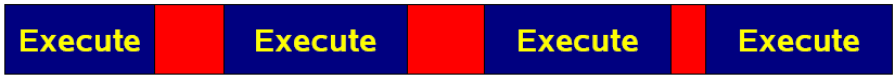


Why adding threads to a core?

Execution of two threads:



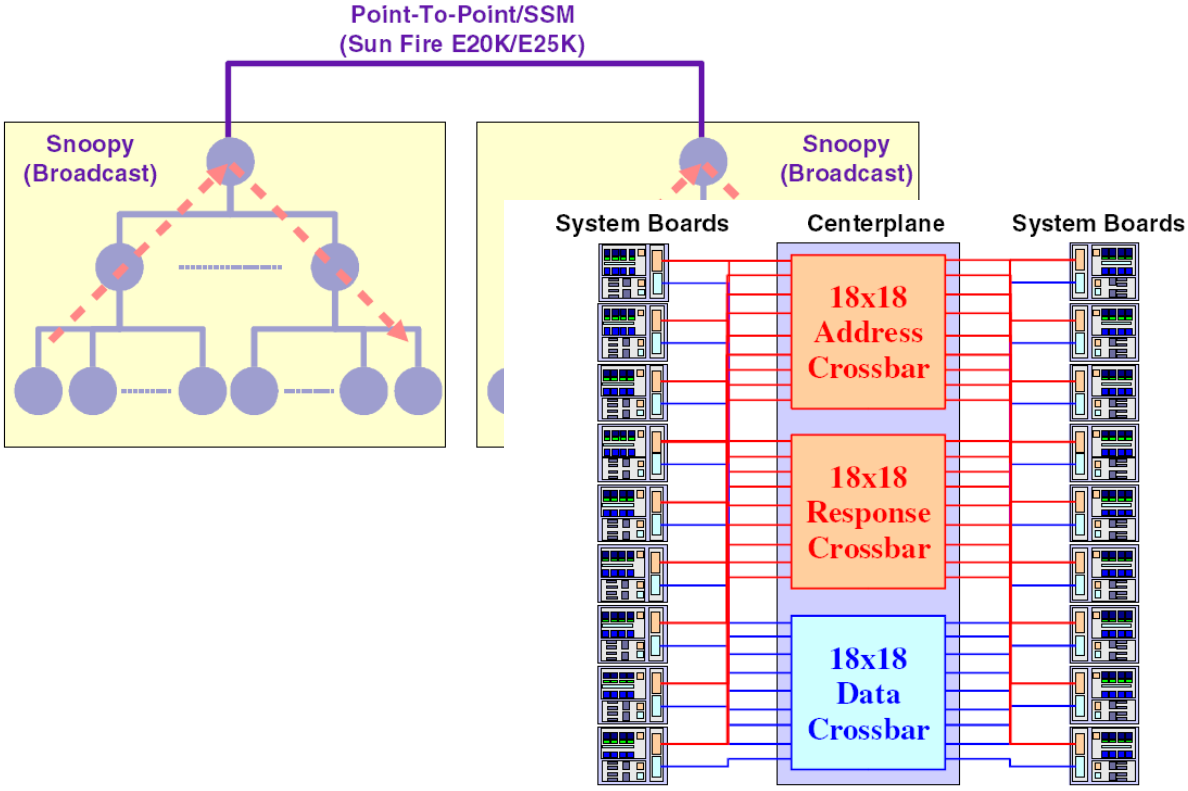
Interleaving the work – better utilization:



Keyword: “Throughput Computing”

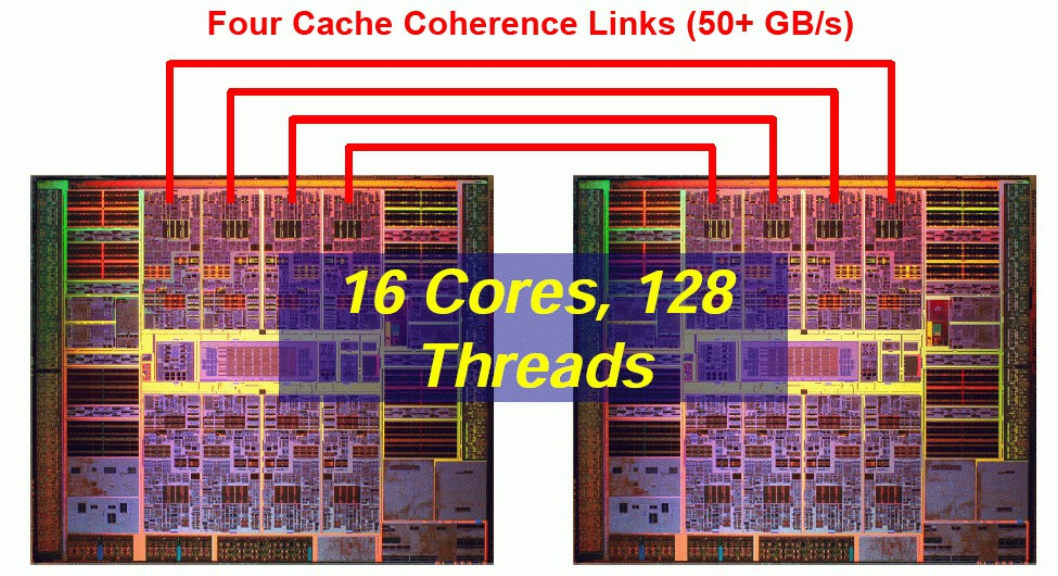
Building parallel systems

Sun Fire E25k – historical view (2008)



Sun Fire T5140/T5240 (2011)

UltraSPARC T2 Plus



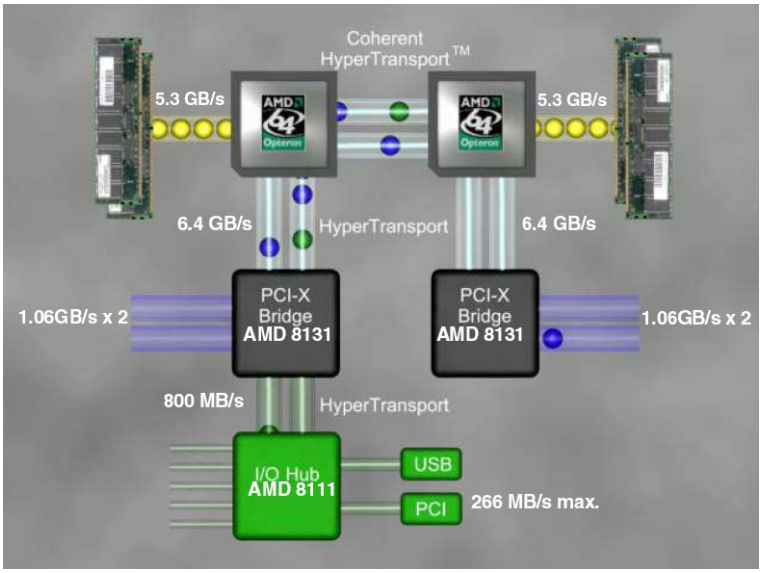
Comparison: SF E25k vs SF T5140

144 threads	vs	128 threads
42 Rack Units	vs	1 Rack Unit

AMD Opteron: multi-CPU systems

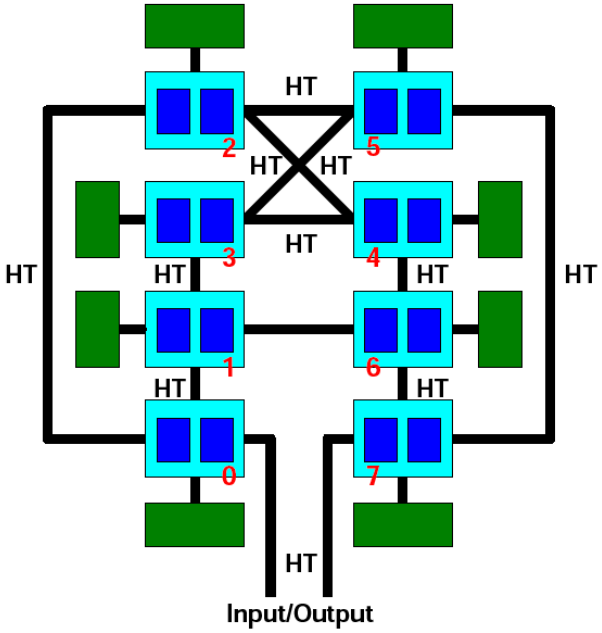
- 5.3 GB/s dedicated memory bandwidth per CPU
- CPU-to-CPU bw: 3.2 GB/s in each direction
- I/O independent of memory access
- adding CPUs adds memory and I/O bandwidth

numbers based on DDR333 RAM



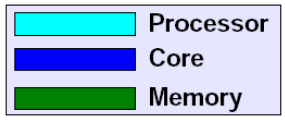
Opteron: eight socket systems

Example: Sun Fire X4600



Number of hops between sockets

	0	1	2	3	4	5	6	7
0	0	1	1	2	2	2	2	3
1	1	0	2	1	2	2	1	2
2	1	2	0	2	1	1	2	2
3	2	1	2	0	1	1	2	2
4	2	2	1	1	0	2	1	2
5	2	2	1	1	2	0	2	2
6	2	1	2	2	1	2	0	1
7	3	2	2	2	2	2	1	0

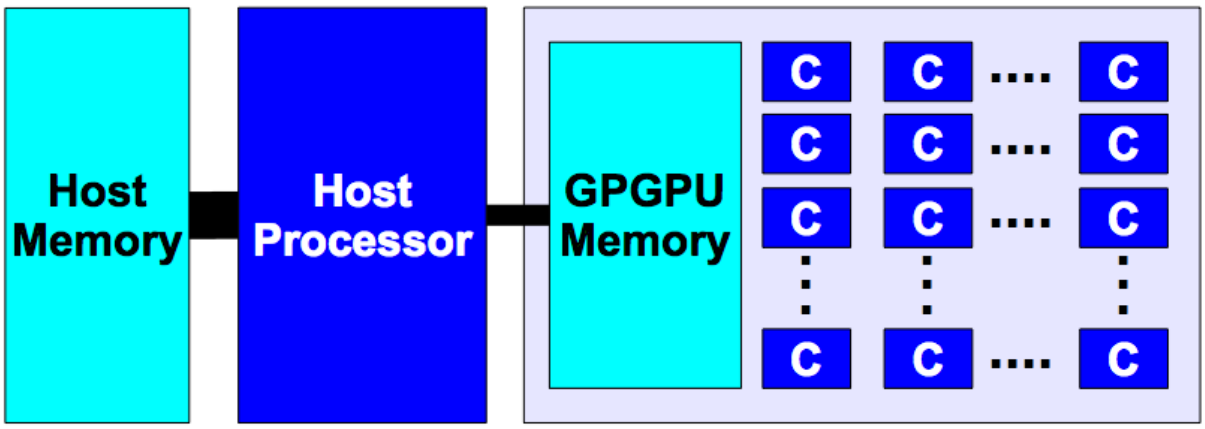


GPU computing - Accelerators

Graphics Processors

- Specialized hardware for operations typical for graphics rendering
- lots of cores (SPs – scalar processors)
- very fast memory (expensive!) - limited in size
- more instructions have been added over the last years to do more general purpose computing
- programming environments (CUDA, OpenCL) to harness the power of the GPUs

A generic GPGPU

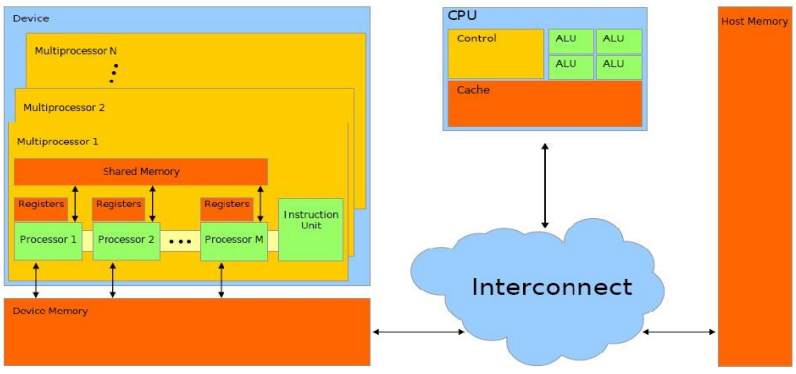


GPU “features”

- every “core” is a very simple processor
- “cores” cannot work independently
- no independent execution of threads, but SIMT (*Single Instruction, Multiple Threads*)
- no global address space, neither within the GPU, nor with the CPU
- no cache-coherency
- latency hiding by executing many threads

Disconnected memory

- Bottleneck: the CPU memory and the GPU memory are not the same, i.e. data has to be moved – in and out.



- PCI Express 3 bandwidth: 8-16 GB/sec

HPC on/with GPUs/accelerators

- offload the right problems to the GPU
- avoid memory traffic between CPU and GPU
- use asynchronous memory transfers

- not all problems are well suited for GPU computing
- large data sets are an issue
- new programming models

The future

Prediction is very difficult,

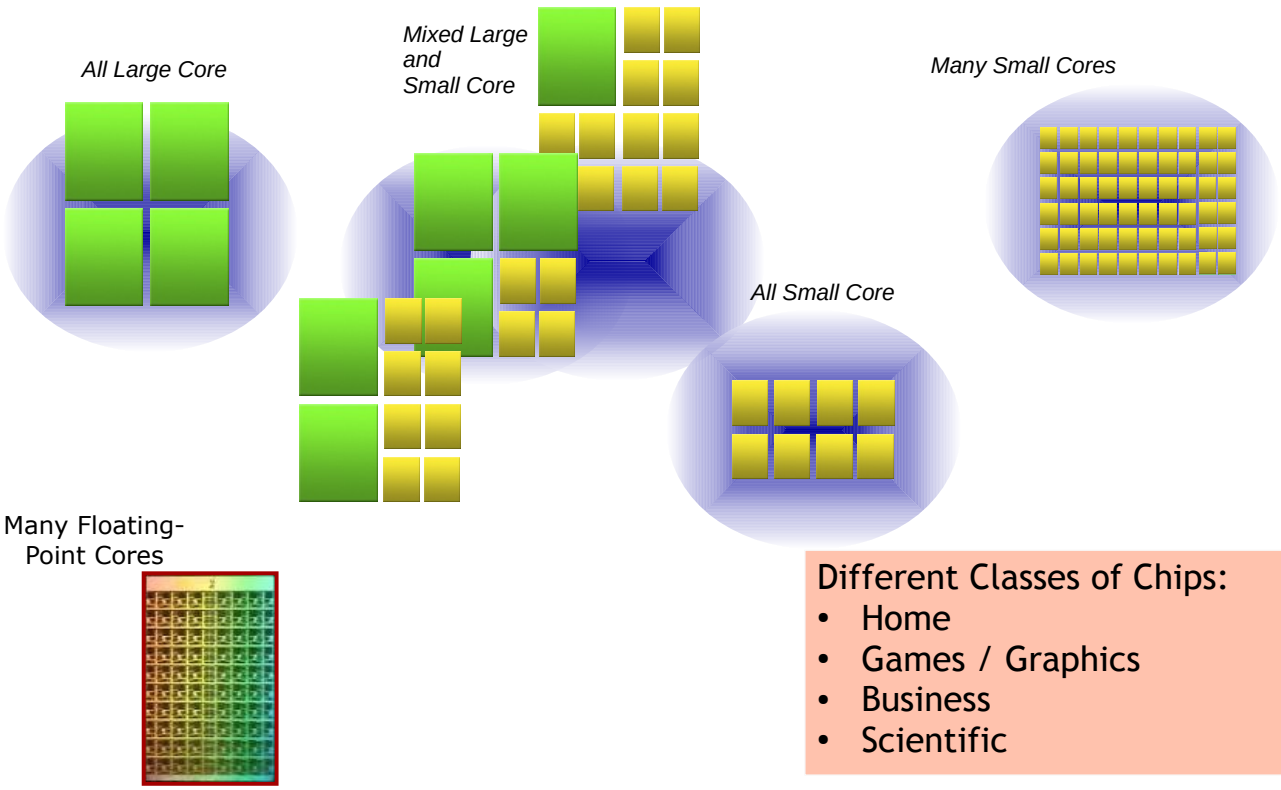
especially if it is about the future."

-- Niels Bohr (1885-1962)

The future ... of GPUs

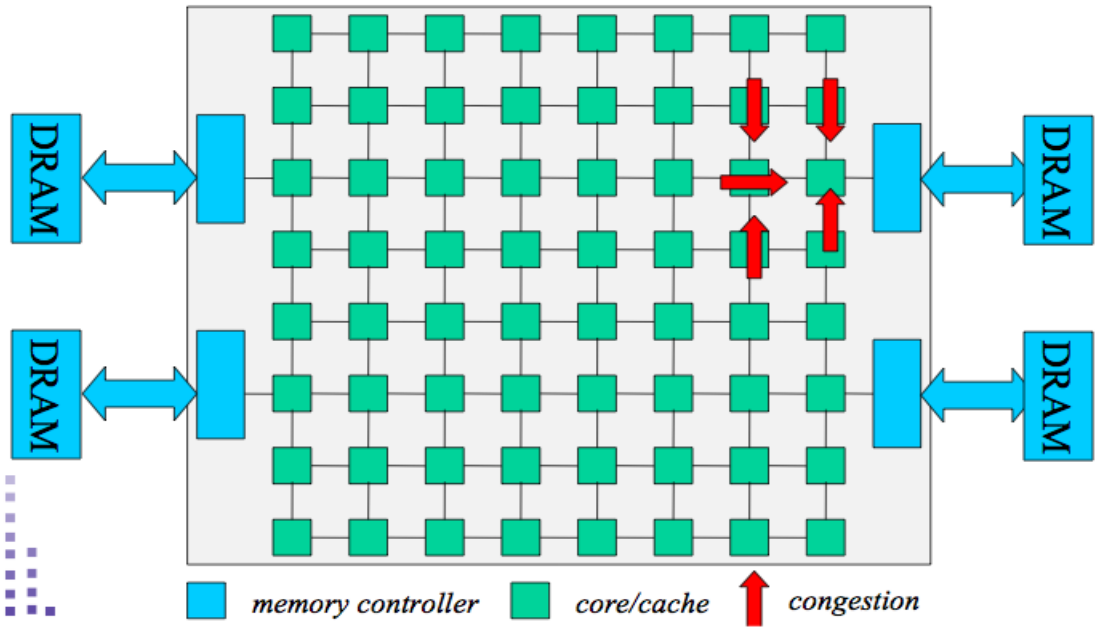
- In the past:
  - we have had “accelerators” before: transputers, etc
  - specialized hardware – not a big market.
  - who remembers them today?
- GPUs are a based on a mass market product
  - continue to play computer games ... it's good for us!
- GPU computing will (probably) not go away
  - … but it will develop/change
  - ... it adapts faster to new trends (e.g. Machine Learning)

The future ... of multi-core

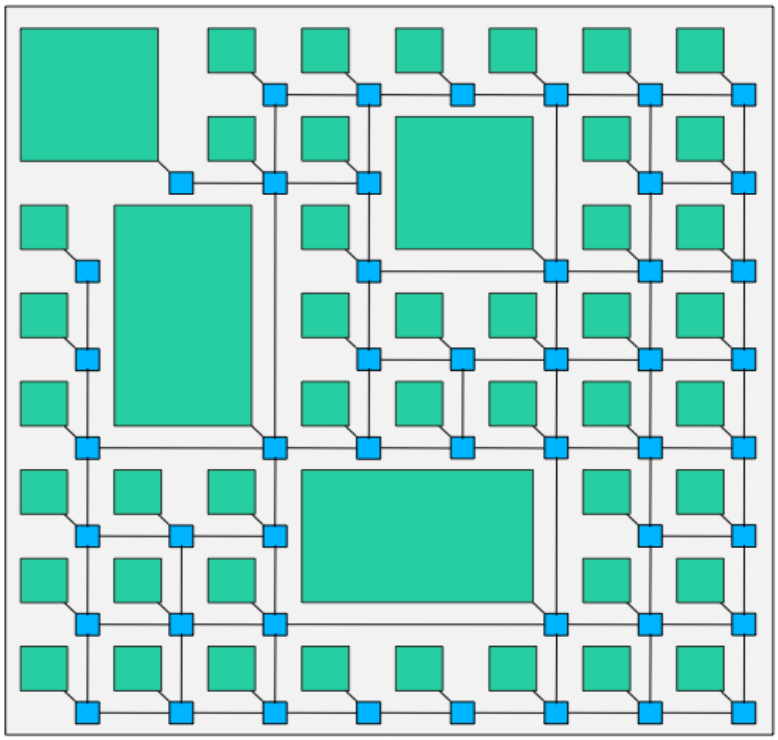


Different Classes of Chips:
- Home
- Games / Graphics
- Business
- Scientific

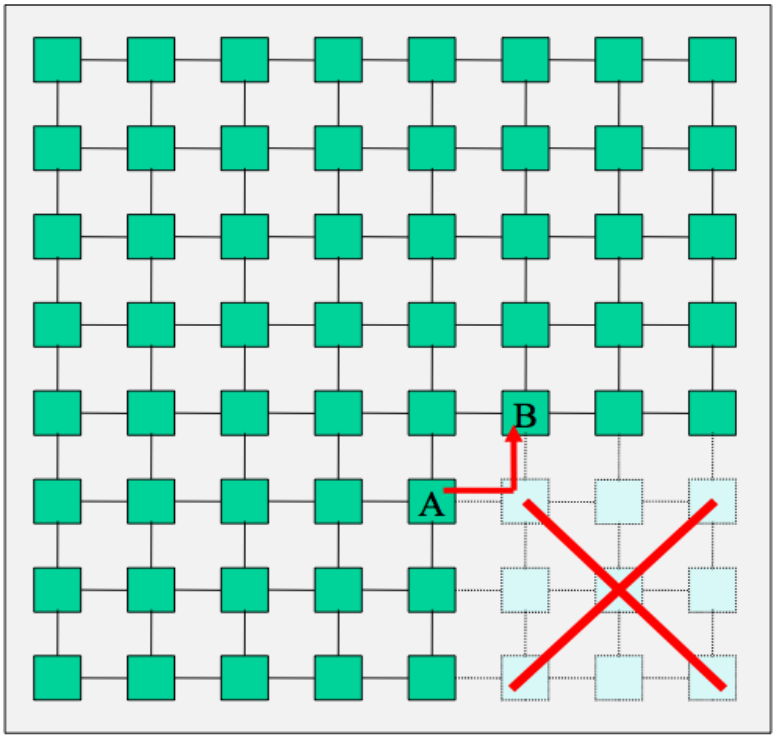
Many-core challenges I



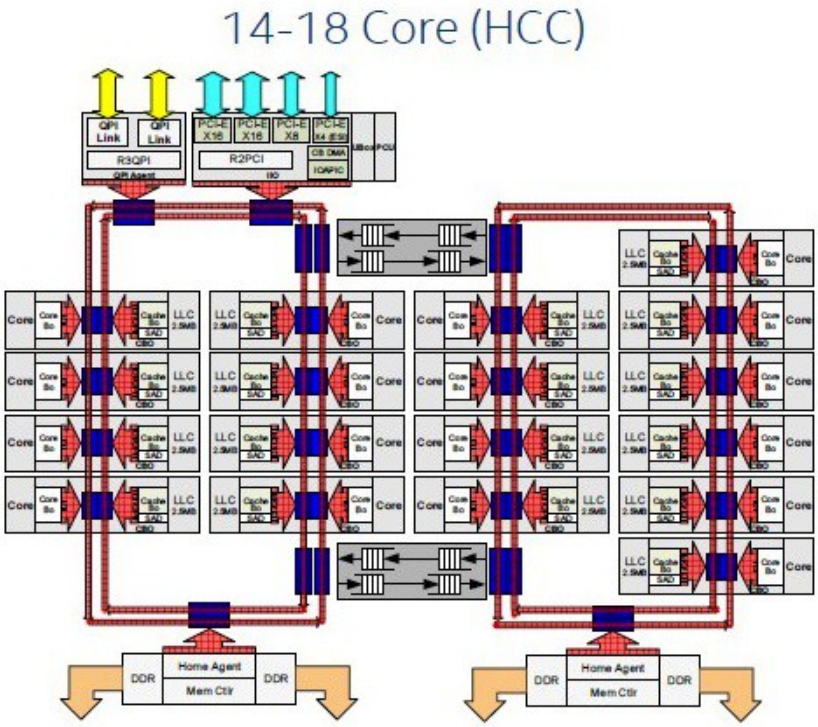
Many-core challenges II



Many-core challenges III



A 2014 multi-core chip: Intel Haswell



The future ...

Trends:

- "GPU cores" will be integrated into future CPUs, e.g. Intel's "Sandy Bridge", AMD's Fusion
- there will be shared-memory (no more copying of data!), and there might be even cache-coherency
- there is a lot of interest, and already some large communities
- ... but who really drives the development?

Commodity

- **Moore's "Law" favored consumer commodities**
  - **Economics drove enormous improvements**
  - **Specialized processors and mainframes faltered**
  - **Custom HPC hardware largely disappeared**
  - **Hard to compete against 50%/year improvement**
- **Implications**
  - **Consumer product space defines outcomes**
  - **It does not always go where we hope or expect**
  - **Research environments track commercial trends**
  - **Driven by market economics**
  - **Think about processors, clusters, commodity storage**

courtesy: Jack Dongarra

The future ... of software

- Challenges:
  - how to handle millions of cores/threads ...

    ... reliably
  - re-design of algorithms: from coarse-grained parallelism to fine-grained parallelism
  - more development tools needed to achieve this
  - we need standards, to assure portability!
  - long-term perspectives

Summary

- You have heard about:
  - Parallel programming models and basic concepts
  - Parallel architectures (shared / distributed memory)
  - Cache-coherency
  - Multi-core architectures
  - GPUs/accelerators
- Next 3 lectures:

  Portable programming of shared memory systems with OpenMP