Introduction to High-Performance Computing

Giorgio Amati

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2024

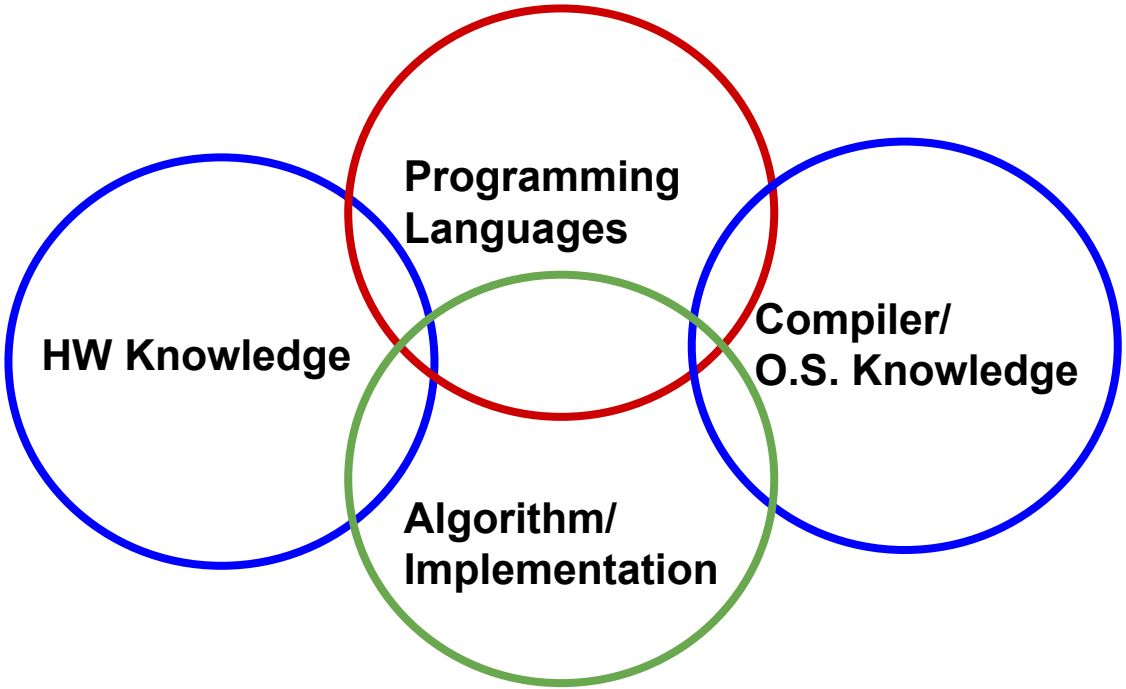
g.amati@cineca.it / g.amaticode@gmail.com

Agenda

- ✓ HPC: What it is?
- ✓ Spoiler…
- ✓ Hardware: how it works
- ✓ Algorithm vs. Implementation
- ✓ Compiler + Floating point + I/O
- ✓ Parallel Paradigm
- ✓ Conclusions & Comments

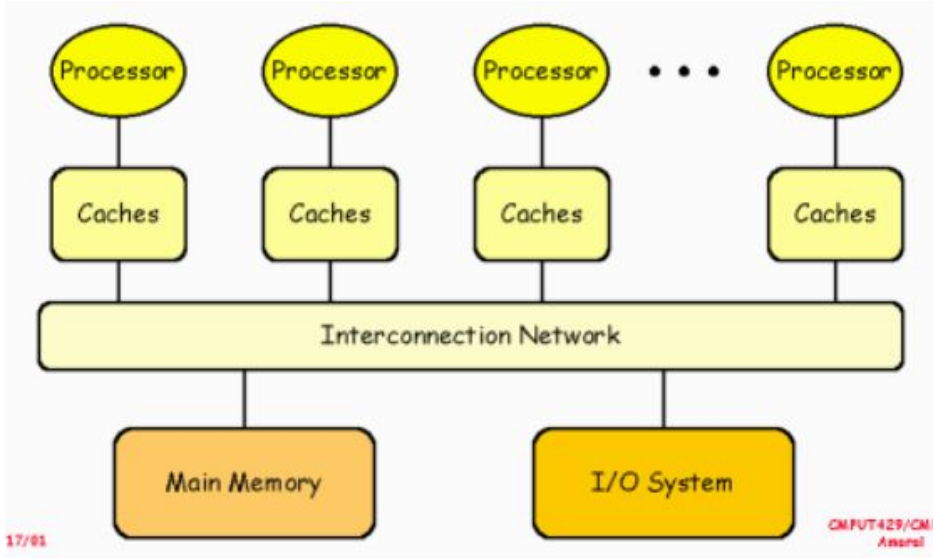
HPC: what it is?

✓ These are the main skills for an efficient HPC



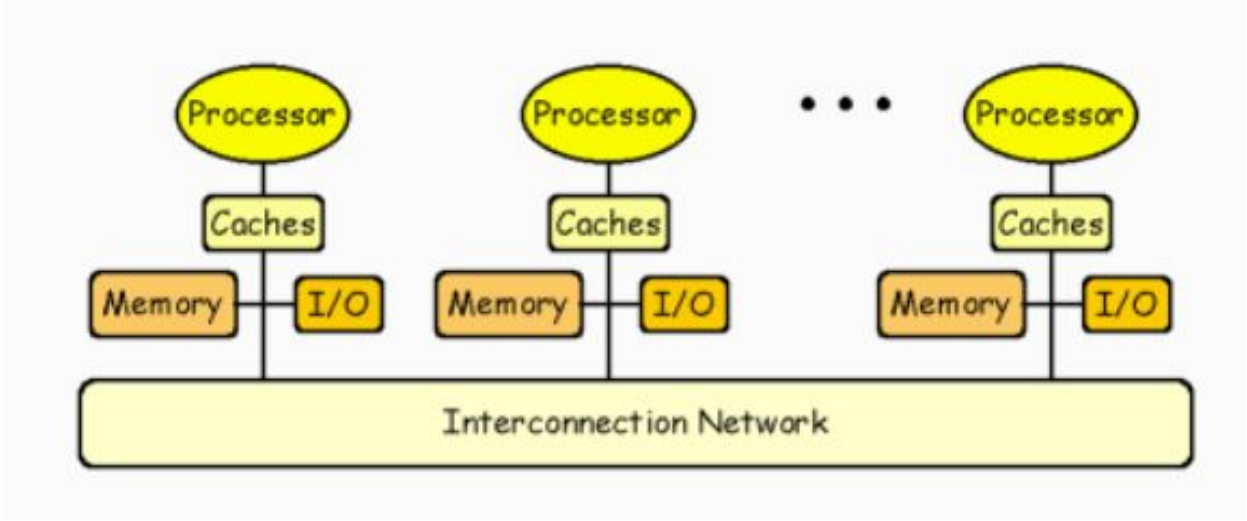
Shared Memory Machine

- ✓ A shared-memory system is an architecture consisting of a number of processors, all of which have direct (i.e. hardware) access to all the main memory in the system. This permits **any** of the system processors to access data that **any** of the other processors has created or will use
  - **UMA**: uniform memory access
  - **NUMA**:non uniform memory access



Distributed Memory Machine

✓ Distributed memory refers to a computing system in which each processor (or a node) has its memory. Computational tasks efficiently operate with local data, but when remote data is required, the task must communicate (using explicit messages) with remote processors to transfer the right data.



GPU vs. CPU

- ✓ General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple video cards in one computer, or large numbers of graphics chips, further parallelism the already parallel nature of graphics processing.

- ✓ In brief.
  - GPU are less “flexible” respect CPU
  - GPU could be really more performing respect CPU
  - Classical example:
    - GPU → BUS
    - CPU → sport car

GPU

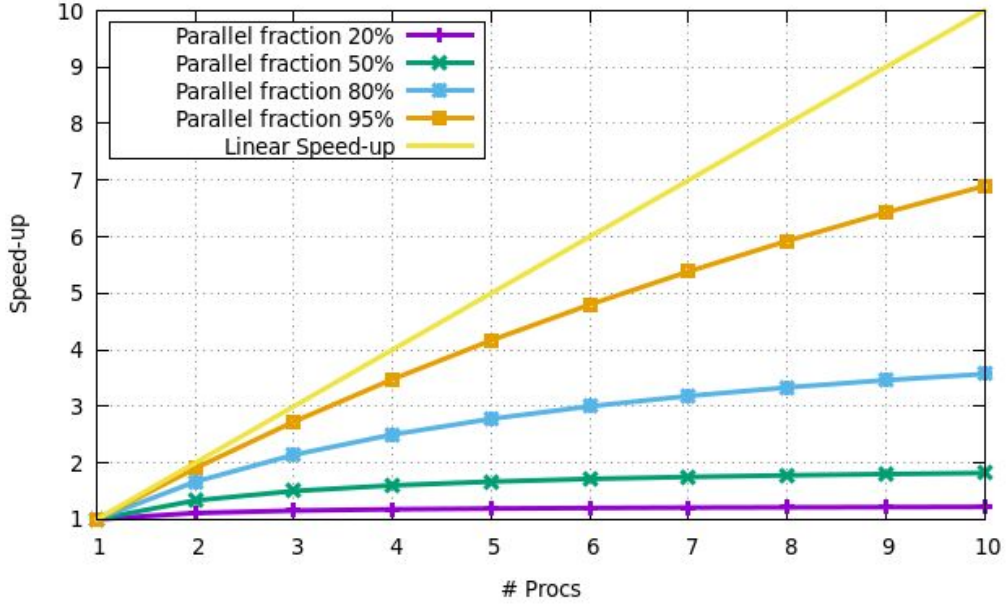
NVIDIA GA100

- ✓ up to 128 Streaming multiprocessor (SM)
- ✓ Each SM has
  - 64 FPU@32bit
  - 32 FPU@64bit
  - 64 INT@32bit

Amdhal's law

- ✓ F = parallel fraction of the code (F < 1,00)
- ✓ N = #of processors
- ✓ S = velocity increment (Speed-up)

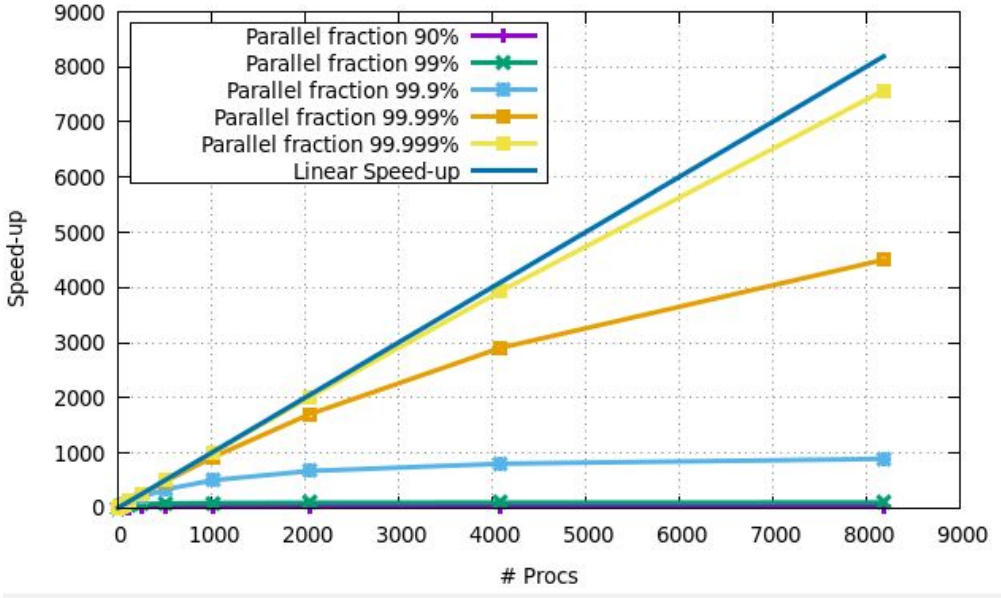
$$S = \frac{1}{(1-F) + \frac{F}{N}}$$



Amdhal's law

- ✓ F = parallel fraction of the code (F < 1,00)
- ✓ N = #of processors
- ✓ S = velocity increment (Speed-up)

$$S = \frac{1}{(1-F) + \frac{F}{N}}$$



Different Level of optimization

- ✓ **Single core optimization**
  - Vectorization
  - Data access
  - Serial Optimized libraries
  - Compiler optimization
- ✓ **Single-node optimizations**
  - Intra-node optimization
  - Data access
  - Shared memory Parallelization/Distributed memory Parallelization
  - Offloading
  - Shared Memory Optimized Libraries
- ✓ **Multi-node optimizations**
  - Distributed memory optimization (i.e. load balancing)
  - Parallel Optimized libraries
- ✓ **I/O issues**

You cannot skip one single level of optimization!!

Optimized Libraries

- ✓ There are many optimized libraries
  - Serial: BLAS, LAPACK,...
  - Parallel: PETSc, Trillinos, SCALAPACK, FFTW…
  - Proprietary: ESSL, MKL, ….
- ✓ Usually could be robust or flexible or fast (but few times all together)
- ✓ Sometimes can help to perform a part of you problem more hard to solve all your problem
- ✓ library maintenance could be a problem, for the open source one

- ✓ **HINT**: do not reinvent the wheel! Look around if you find something that's fine for you (no one needs to write a parallel FFT from scratch)

Heterogeneous Machine/1

- ✓ Real world systems are “heterogenous”.
- ✓ Up to three different level of parallelization to manage
- ✓ Usually they are a cluster (i.e. distributed memory system) of nodes
  - Each of them is a shared memory system of cores (up to 128 or more)
    - Eventually with GPUs (up to 8)

For example Leonardo has

- ✓ 3456 Nodes, interconnect via an infiniband network at 200Gb/s
  - Each node has 32 Core with 512GB Ram
    - Each node has 4 GPU with 64GB HBM

How to handle this “complexity”???

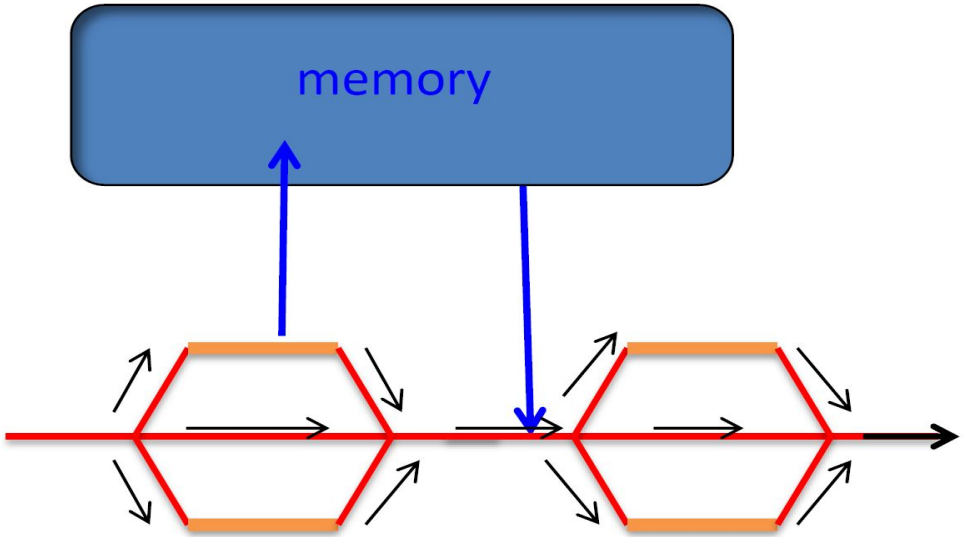
Heterogeneous Machine/2

How to handle this “complexity”???

- ✓ Shared Memory parallelism
  - OpenMP (CPU)
  - OpemACC (GPU)
  - OpenMP (GPU)
  - Cuda/Cudafortran/HIP (GPU)
  - …
- ✓ Distributed memory parallelism
  - MPI
  - …

Shared memory parallelism

- ✓ Work-sharing model
- ✓ One process (master thread) that:
  - “forks” in different threads
  - each threads access to a common memory
  - each threads performs some operation
  - all threads joins the master threads
- ✓ Risk of “race conditions”
- ✓ No risk of “deadlocks”



OpenMP in few slide

- ✓ Shared Memory parallelism: work sharing
- ✓ A single program that share work between “workers” (threads)
  - Fork/join
- ✓ **Programmer has to share work between threads and keep data coherence**
- ✓ **Programmer has not to take care of data movement but to say which data is global (i.e. external and accessible to every threads) and which local (i.e. private to one threads)**
- ✓ **Incremental (loop by loop)**
- ✓ From rel. 4.5 support for offloading to GPU (still to work on)
- ✓ **Directive based: no code modification (in principle…)**

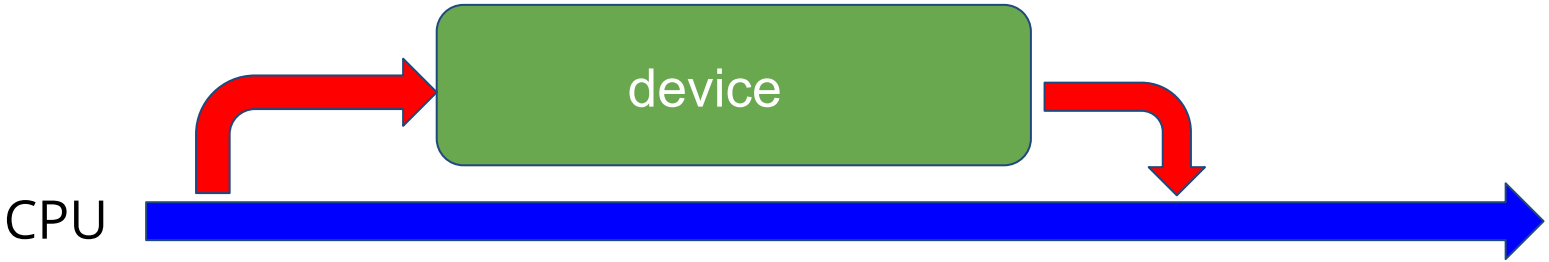
OpenMP in one slide: Fortran

Matrix matrix multiplication:

```
!$OMP PARALLEL DO &
!$OMP DEFAULT(NONE) &
!$OMP PRIVATE(i,j,k)  &
!$OMP SHARED(a,b,c,n)
    do j = 1, n
       do k = 1, n
          do i = 1, n
             c(i,j) = c(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo
!$OMP END PARALLEL DO
```

1 Thread (serial) Mflops = 5682
8 threads Mflops = 37590

Device Offloading

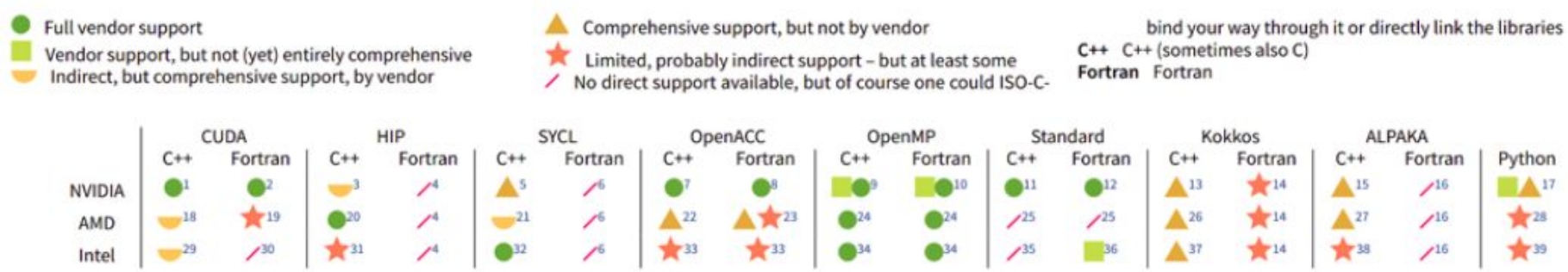


Offloading:

- ✓ Some work is “demanded” to an “external” device (GPU,FPGA)
- ✓ explicit data movement back and from the device
- ✓ the bottleneck is the data movement
- ✓ usually devices has less memory then CPU (M100: 4x16GB vs. 192 GB)

Porting to GPU: state of the art

- ✓ Up to now no clear and portable solution
- ✓ No silver bullet!



Performance portability: recap

✓ Up to now no clear and portable solution

A. **Intrinsic language constructs** (i.e., do concurrent Fortran or parallel stl C++)
B. **Vendor specific paradigms** (i.e., CUDA/HIP/OneAPI)
C. **OpenCL standard**
D. **Directive-based standards** (i.e., OpenMP/OpenACX)
E. **high-level portability framework** (i.e., Kokkos/Legion/OpenSYCL/alpaka/RAJA)
F. **External GPU-based subroutine libraries** (i.e., PeTSc)

A --> Not "complete" or even "partial" support by compilers
B --> it should give best performance. Translation between paradigm (e.g., CUDA --> HIP) could be not so simple (e.g., for Fortran)
C --> Code verbose, performance could be hard to achieve, only C.
D --> Not "complete" support and performance could be difficult to achieve
E --> Implies (serious) code rewriting and only some languages are supported (e.g., C/C++)
F --> Only few Algorithm were ported to GPU

DO CONCURRENT & C++ parallel algorithms

- ✓ Language construct, defined in the standard, that allow to express loop-level parallelism with (possible) back-end to GPU.
- ✓ Parallelization/porting is left to the compiler
- ✓ Not suitable for the best performance but (not even supposed to), if compiler is smart enough, good performance
- ✓ Supported by few compilers (no AMD GPU)

- ✓ “Cross-platform programming model for many-core lattice Boltzmann simulations”. Jonas Latt et al, (https://doi.org/10.1371/journal.pone.0250306)
- ✓ https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/

DO CONCURRENT: example

- Matrix-Matrix multiplication
- IBM Pwr9, V100
- size = 2048

```
do concurrent(j=1:n)
  do i = 1, n
    do k = 1, n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

- serial (**-fast**): 3584 Mflops
- multiprocessor (**-stdpar=multicore -mp**): 22554 Mflops
- gpu (**-stdpar=gpu**): 162890 Mflops

OpenACC in few slides

- ✓ Offload parallelism: work is demanded to an external accelerator
- ✓ **Similar philosophy to OpenMP but**
  - **Implicit data movement back/from GPU, via PCI bus or “ad hoc” link (i.e. NVlink)**
  - **More sensible to data movement, this is the bottleneck for many codes**
- ✓ Programmer has to take care of data movement back/from GPU and which piece of work offload to gpu
- ✓ Incremental (loop by loop) but **not in performance (data movement hides performace)**
- ✓ Directive based: no code modification (in principle)
- ✓ Performance heavily dependent from the compiler (and release) used
  - Not supported by all compilers (Nvidia, CRAY-Amd)

OpenACC in few slides: Fortran

Matrix matrix multiplication:

```
!$acc parallel copyin(a,b) copyout(c)
!$acc loop parallel
  do j=1, n
     do k=1, n
        do i=1, n
           c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
     end do
  end do
!$acc end parallel
```

1 CPU: **Mflops=2259**
1 GPU: **Mflops=167544**

GPU Offload in few slides

- ✓ Offload parallelism: work is demanded to an external accelerator
- ✓ Similar philosophy to OpenACC but inserted in OpenMP framework
- ✓ **Same issues for OpenACC: programmer has to take care of data movement back/from GPU and which piece of work offload to gpu**
- ✓ **Incremental (loop by loop) but...**
- ✓ Directive based: no code modification (in principle)
- ✓ Performance heavily dependent from the compiler (and release) used
  - in general slower the OpenACC (to maintain backward compatibility)
  - e.g. pointers are not supported by gnu fore release < 12.0

GPU Offload in few Slide

Matrix matrix multiplication (4096^2):

```
!$OMP target teams distribute parallel do map(to:a,b) map(from:c)
    do j = 1, n
       do i = 1, n
          do k = 1, n
             c(i,j) = c(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo
```

- ✓ GPU V100: (nvfortran rel. 21.5) Mflops=127943
- ✓ GPU V100: (xlf rel.16.0) Mflops=109268
- ✓ GPU V100: (gfortran rel.9.3) Mflops= 6821

CUDA in few slides

- ✓ **C**ommon **U**nified **D**evice Architecture
- ✓ Nvidia Proprietary language
- ✓ Offload parallelism: work is demanded to an external accelerator
- ✓ C-based language with extension (Python,Fortran,...)
- ✓ Kernel based parallelism
- ✓ heavy code modification
- ✓ Very fine control
- ✓ **HIP**: AMD version very similar philosophy

CUDA Example/1

```
* cudaArray* cu_array;
texture<float, 2> tex;
// Allocate array
cudaChannelFormatDesc description = cudaCreateChannelDesc<float>();
cudaMallocArray(&cu_array, &description, width, height);
// Copy image data to array
cudaMemcpy(cu_array, image, width*height*sizeof(float),
cudaMemcpyHostToDevice);
// Bind the array to the texture
cudaBindTextureToArray(tex, cu_array);
// Run kernel
dim3 blockDim(16, 16, 1);
dim3 gridDim(width / blockDim.x, height / blockDim.y, 1);
kernel<<< gridDim, blockDim, 0 >>>(d_odata, height, width);
cudaUnbindTexture(tex);
```

CUDA example/2

```
__global__ void kernel(float* odata, int height, int width)
{
   unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
   float c = tex2D(tex, x, y);
   odata[y*width+x] = c;
}

....
```

CUDA Fortran Example/1

```
real(my_kind), dimension(:,:), allocatable:: a ! matrix (origin)
real(my_kind), dimension(:,:), allocatable:: b ! matrix (origin)
real(my_kind), dimension(:,:), allocatable:: c ! matrix
real(my_kind), dimension(:,:), device, allocatable:: a_gpu ! matrix (origin)
real(my_kind), dimension(:,:), device, allocatable:: b_gpu ! matrix (origin)
real(my_kind), dimension(:,:), device, allocatable:: c_gpu ! matrix

a_gpu = a
b_gpu = b
c_gpu = c
!$cuf kernel  do <<<*,*>>>
do j = 1, n
    do k = 1, n
        do i = 1, n
           c_gpu(i,j) = c_gpu(i,j) + a_gpu(i,k)*b_gpu(k,j)
        enddo
    enddo
enddo
```

OpenCL & SycI

- ✓ OpenCL: Open standard for heterogeneous systems (https://www.khronos.org/opencl/)
- ✓ SyCl: cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++(https://www.khronos.org/sycl/)
- ✓ Itt could be a good solution for C++ programmer/code
- ✓ The (nasty) question is how much it is supported by the vendor respect their “proprietary” solution?

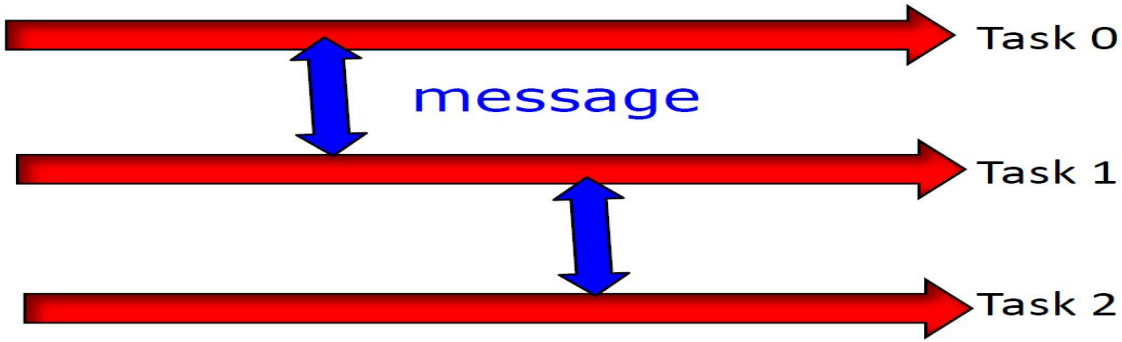
- ✓ Personal comment: It work quite smoothly on AMD MI100 GPU
- ✓ https://www.youtube.com/playlist?app=desktop&list=PLA-vfTt7YHI1SAiiVqviveaSDa8DSt7j9

Make it harder: OpenMP+OpenACC

✓ Each thread controls one GPU

```
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP PRIVATE(i,k,jstart,jstop,idthread) SHARED(a,b,c,n,delta)
     idthread = OMP_GET_THREAD_NUM()
      jstart = 1 + idthread*delta
     jstop =  (idthread+1)*delta
     call acc_set_device_num(idthread, acc_device_nvidia)!
!$acc data copyin(a,b(:,jstart:jstop)) copyin(c(:,jstart:jstop))
!$acc kernels
     do j = jstart, jstop
          do i = 1, n
                    do k = 1, n
                         c(i,j) = c(i,j) + a(i,k)*b(k,j)
                   enddo
          enddo
     enddo
!$acc end kernels
!$acc update host(c(:,jstart:jstop))
!$acc end data
!$OMP END PARALLEL
```

Message Passing Parallelism



Different processes (tasks) that:

- ✓ Run independently one from the other
- ✓ each task has its own memory
- ✓ data exchange & synchronization is done via messages
- ✓ no risk of “race conditions”
- ✓ risk of “deadlock”

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 but obviously Shared Memory doesn’t work on Distributed Memory Systems

MPI in few slides

- ✓ Distributed memory parallelism
- ✓ N different and independent workers (tasks): each do its own work
- ✓ **Programmer has to care of data movement between tasks and data coherence and synchronization between tasks**
- ✓ Not incremental parallelism
- ✓ Deep code modifications
- ✓ Performance deeply depends on the mpi library used
- ✓ send/receive model (i.e. post office model)
  - Send a message in an envelope with an address of the receiver
- ✓ MPI 3.0 introduce many advanced features (e.g. one-sided communications)
  - But it still not fully supported from different mpi libraries

MPI example

Hello world!!!!!!

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

main(int argc, char **argv)
{
   int node;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &node);
   printf("Hello World from Node %d\n",node);
   MPI_Finalize();
}
```

✓ A little more complex in programming…..

Message Passing

- ✓ **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);
  - Can overlap communication with computation
- ✓ **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.
  - No fault tolerant (MPI)

MPI+X

- ✓ With MPI+X thee mixing of different programming languages are presented
  - MPI+OpenMP
  - MPI+OpenACC
  - MPI+OpenMP-Offload
  - MPI+OpenMP+OpenACC
  - ,,,,

…..

Optimized Libraries (for GPU)

- ✓ Some libraries has some algorithm ported on GPU
  - Usually only few
  - e.g. PETSc
- ✓ Also some intrinsic function are ported to GPU
  - e.g. matmul
- ✓ Some vendor offers some library ported to GPU
  - NVIDIA: cuFFT, cuBLAS, AMGX, ...
  - AMD: ROCm (FFT, BLAS, ...)
  - INTEL: OneAPI
- ✓ Portability issues
- ✓ Spend some time to look if there's something fine for your problem already ported on GPU

intrinsic function

Matrix matrix multiplication (16384^2):

```
#ifdef _OPENACC
    use cutensorex
#endif
…
    real(my_kind), dimension(:,:), allocatable:: a ! matrix (origin)
    real(my_kind), dimension(:,:), allocatable:: b ! matrix (origin)
    real(my_kind), dimension(:,:), allocatable:: c ! matrix
…
    c =  matmul(a,b)
```


✓ GPU A100@64GB: (nvfortran rel. 23.1) Mflops=6721790

Agenda

- ✓ HPC: What it is?
- ✓ Spoiler…
- ✓ Hardware: how it works
- ✓ Algorithm vs. Implementation
- ✓ Compiler + Floating point + I/O
- ✓ Parallel Paradigm
- ✓ Conclusions & Comments

Example: matrix-matrix multiplication/2

- ✓ Performance can really different, depending on HW, implementation etc….
- ✓ Improvement in performance can be really high….
- ✓ ….or you can easily “depress” performance
- ✓ Performance in Mflops: higher is better

#test	Size	HW	MFlops	Ratio
1-Cache unfriendly	2048	CPU	201	-
2-Cache friendly	2048	CPU	4870	**24x**
3-OpenACC	8192	GPU-V100	361328	**1797x**
4-OpenACC+unrolling	8192	GPU-V100	448923	**2233x**
5-Matmul	16384	GPU-A100	6721790	**33441x**

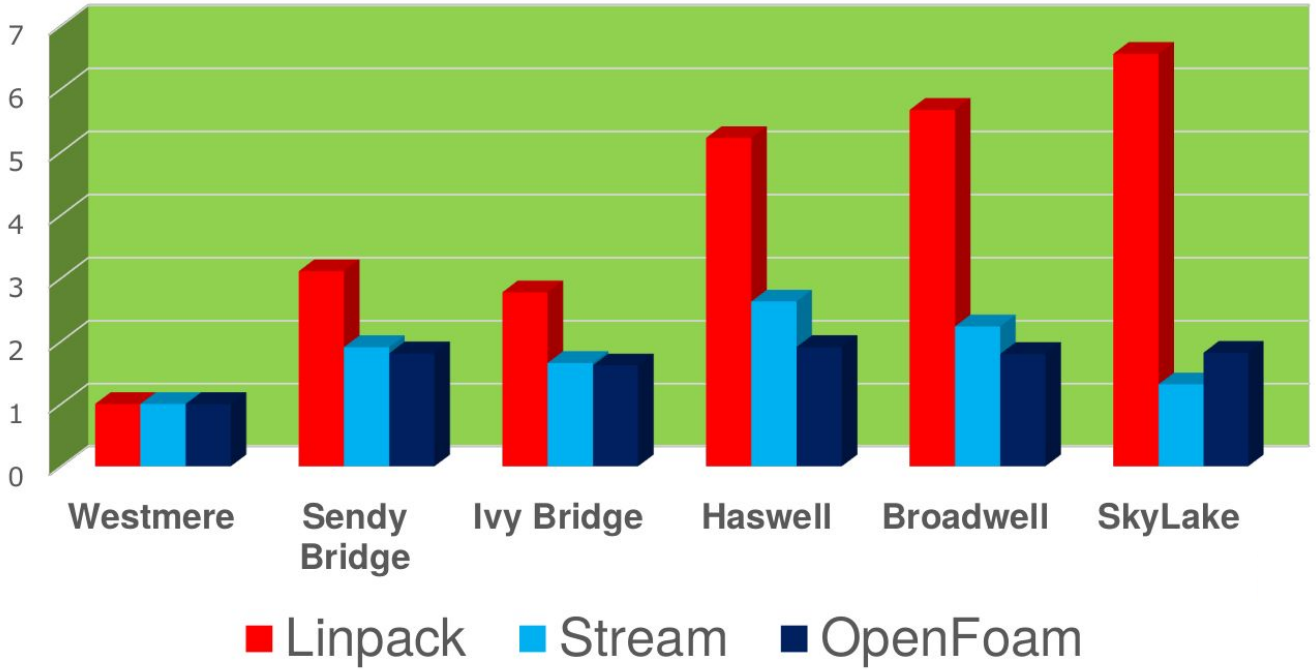
Performance improvements

- Matrix-Matrix multiplication (time in seconds)

	Single prec.	Double prec.
Cache un-friendly loop	7500''	7300''
Cache friendly loop	206''	246''
Compiler Optimization	84''	181''
Handmade Optimization	23''	44''
Optimized library (serial)	6.7''	13.2''
Optimized library (OMP, 2 threads)	3.3''	6.7''
Optimized library (OMP, 4 threads)	1.7''	3.5''
Optimized library (OMP, 8 threads)	0.9''	1.8''
PGI accelerator (GPU)	3''	5''
CUBLAs (GPU)	1.6''	3.2''

(Serial) Performance Evolution

Old slide but still valid now!



Twelves way to fool people about ….

1. Quote only 32-bit performance results, not 64-bit results.
2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
3. Quietly employ assembly code and other low-level language constructs.
4. Scale up the problem size with the number of processors, but omit any mention of this fact.
5. Quote performance results projected to a full system.
6. Compare your results against scalar, unoptimized code on Crays.
7. When direct run time comparisons are required, compare with an old code on an obsolete system.
8. If MFLOPS rates must be quoted, base the operation count on the parallel implementation, not on the best sequential implementation.
9. Quote performance in terms of processor utilization, parallel speedups orMFLOPS per dollar.
10. Mutilate the algorithm used in the parallel implementation to match the architecture.
11. Measure parallel run times on a dedicated system, but measure conventional run times in a busy environment.
12. If all else fails, show pretty pictures and animated videos, and don't talk about performance.

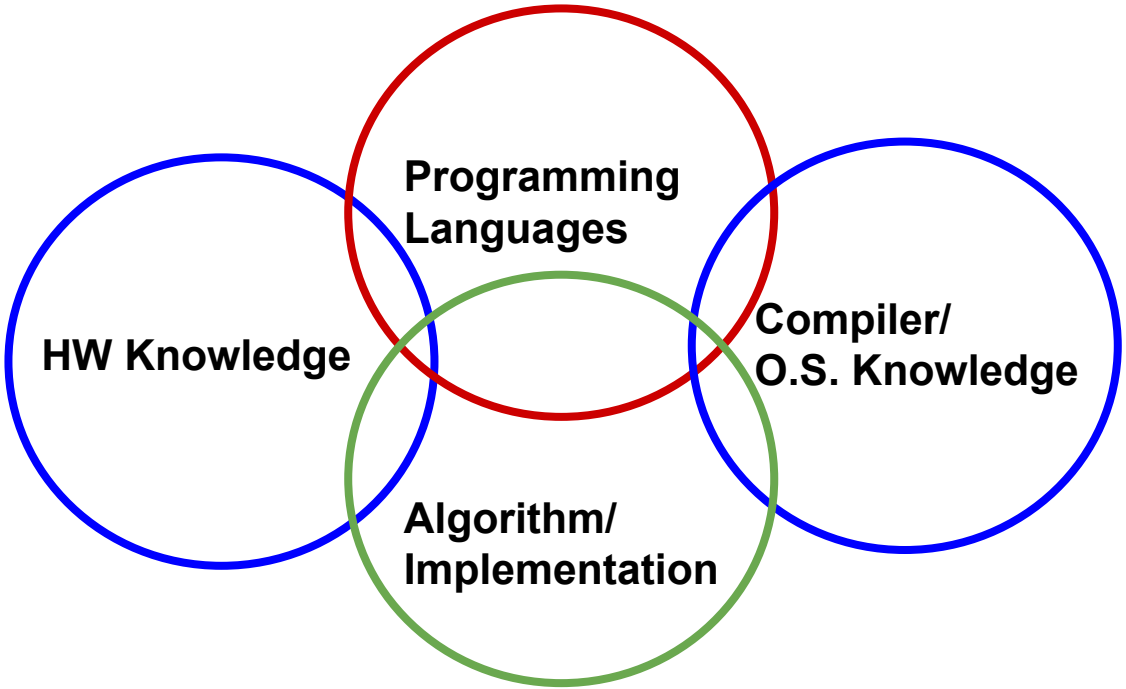
comp.lang.fortran (12/03/03)

How to decrease the run time of a code:

1. Improve the algorithm.
2. Same as 1).
3. Tweak the code.
4. Fiddle with compiler switches.
5. Run it on a faster computer :-).
6. Ignore all compiler-provided error checking (a follow-on from [4]).
7. Minimise small subroutines - make them inline.
8. Check array accesses are 1st-index varying fastest.
9. Ensure that the code is working correctly.
10. Ensure that nothing major is accidentally calculated twice.
11. If differential equations: look at alternative solvers, and whether the tolerances are appropriate.
12. If algebraic equations or optimisation - as [11].
13. Minimise saving data to a file - try & do it only at the end, rather than during the course of calculations.
14. Minimise use of automatic arrays, or deallocating arrays.
15. Ensure that the machine is not limited by memory constraints (too much page-swapping).
16. Return to fortran77-style array parameters, rather than fortran90 usage of array descriptors.

HPC: what it is?

✓ These are the main skills for an efficient HPC



Different Level of optimization

- ✓ **Single core optimization**
  - Vectorization
  - Data access
  - Serial Optimized libraries
  - Compiler optimization
- ✓ **Single-node optimizations**
  - Intra-node optimization
  - Data access
  - Shared memory Parallelization/Distributed memory Parallelization
  - Offloading
  - Shared Memory Optimized Libraries
- ✓ **Multi-node optimizations**
  - Distributed memory optimization (i.e. load balancing)
  - Parallel Optimized libraries
- ✓ **I/O issues**

You cannot skip one single level of optimization

Some Books

- ✓ Charles Severance; Kevin Dowd “High Performance Computing”, O’Reilly, ISBN 13:9781565923126
- ✓ John L. Hennessy, David A. Patterson , “Computer Architecture: A Quantitative Approach” Morgan Kaufmann; ISBN-10 : 0128119055
- ✓ John L. Hennessy, David A. Patterson , “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann;
- ✓ D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic
- ✓ U. Drepper: “What Every Programmer Should Know About Memory”