Fortran for Scientists

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

WiSe 2025/26, 30.1.2026

Parallelization

Parallelization

Many runs in MC simulations required for reliable conclusions ($\sigma \sim \frac{1}{\sqrt{N}}$)

Often: Result of one run (e.g., path of a neutron through a plate) *independent* from other runs

$\rightarrow$ Idea: acceleration by parallelization
Problem: concurrent access to memory resources, i.e. variables (e.g., n_s, f_refl)
Solution: special libraries that enable multithreading (e.g., `OpenMP`) or multiple processes (e.g., `MPI`) for one program

$\rightarrow$ insert: pipelining, vectorization, parallelization

CPU Performance

What influences the performance of a CPU (= runtime of your code)?

- architecture/design: out-of-order execution (all x86 except for Intel Atom), pipelining (stages), vectorization units (width)
- cache sizes (kB . . . MB) and location: L1 cache for each core, L3 for processor
- clock rate ($\sim$GHz): only within a processor family usable for comparison due to different number of instruction per clock (IPC) of design, even more complicated because of variable clock rates (base, peak) to exploit TDP (thermal design power)
  $\rightarrow$ impact on single-thread performance
- number of cores (1 . . .): $\rightarrow$ impact on multi-thread performance

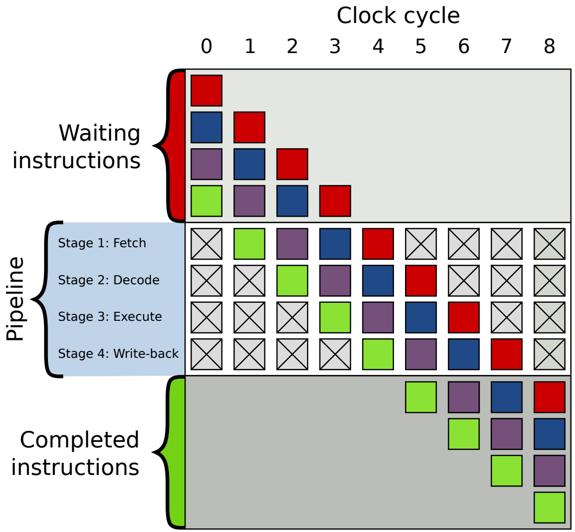
Pipelining I

splitting machine instruction into a sequence

independent execution of instructions, each consisting of

- instruction fetching (IF)
- instruction decoding (ID) + register fetch
- execution (EX)
- write back (WB)

operations of instructions are processed at the same time → quasi parallel execution, higher throughput



Pipelining II

NetBurst disaster

Pentium 4 (2000-2008) developed to achieve $>$ 4GHz (goal: 10 GHz) clockrate by several techniques, i.a., *long* pipeline:

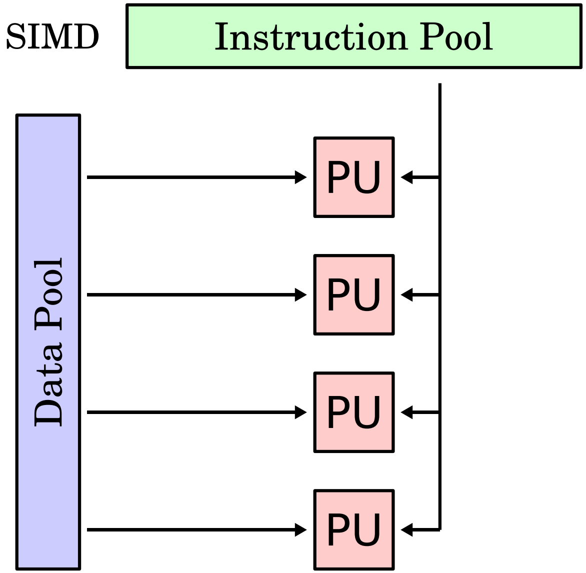
- 20 stages (Pentium III: 10) up to 31 stages (Prescott core)
- smaller number of instructions per clock (IPC) (!)
- increased branch misprediction (also only 10%, improved by 33% for Pentium III)
- larger penalty for misprediction

$\rightarrow$ compensated by higher clock rate

higher clock rate $\rightarrow$ higher power dissipation, especially for 65 (Presler, Pentium D), 90 (Prescott) up to 180 nm (Williamette) structures

$\rightarrow$ power barriere at 3.8 GHz (Prescott)

SSE and AVX I



- SSE - Streaming SIMD Extensions (formerly: ISSE - Internet SSE)
- SIMD - Single Instruction Multiple Data ($\rightarrow$ cf. Multivec, AMD3Dnow!), introduced with Pentium III (Katamai, Feb. 1999)

SSE and AVX II

- enables vectorization of instructions (not to be confused with pipelining or parallelization), often new, complex machine instructions required,
  e.g., `PANDN` $\rightarrow$ bitwise `NOT` + `AND` on *packed integers*
- comprises 70 different instructions, e.g., `ADDPS` – add packed single-precision floats (two "vectors" each with 4 32 bit) into a 128 bit register
- works with 128 bit registers (3Dnow! only 64 bit), but first execution units (before *Core* architecture) only with 64 bit
- AVX - Advanced Vector Extensions with 256 bit registers, theoretically doubled speed!
  since Sandy Bridge (Intel Core 2nd generation, e.g., i7-2600 K) and Bulldozer (AMD)
  $\rightarrow$ AVX-512 with 512 bit registers in Skylake-X (6th generation, e.g., Core i7-6700); AMD Zen 4, 5
  Note: AVX-512 instructions may reduce the clockrate on Intel CPUs (heat limit)

SSE and AVX III

- supported by all common compilers, e.g.,

  ```
  ifort -sse4.2
  ifort -axcode COMMON-AVX512
  g++ -msse4.1
  g++ -mavx512f
  ```

- very easy (automatic) and efficient optimization, e.g., for unrolled loops → vectorization

Caution!

Different precisions for SSE-`doubles` (e.g., 64 bit) and FPU-`doubles` (80 bit), especially for buffering, so results of `doubles`, e.g.,

```
xx = pow(x,2) ;
sqrt( xx - x*x) ;
```

usually not predictable

Multi-cores

Multi-cores

- originally one core per processor, sometimes several processors per machine/board (supercomputer)
- many units, e.g., arithmetic logic unit (ALU), register, already multiply existing in one processor
- first multi-core processors: IBM POWER4 (2001);
  desktop $\rightarrow$ Smithfield (2005), e.g., Pentium D
- Hyper-threading (HT): introduced in Intel Pentium 4 $\rightarrow$ for better workload of the computing units by simulation of another, logical processor core (compare: AMD Bulldozer design with modules)
- today: up to 64 cores for desktop (AMD Zen: Ryzen Threadripper 5995WX, TDP 280 W) or 96 for servers (e.g., AMD EPYC 9654, TDP 360 W – even 2 CPUs per board)
  + Hyperthreading
- arms race of cores instead of clock rate (NetBurst disaster)

Multi-cores and compilers

Acceleration by parallelization

- parallelization done, e.g., by multithreading (from *thread*) for shared memory (RAM on one "node", usually on one mainboard)
- "The free lunch is over" $\rightarrow$ no simple acceleration more of *single-thread* programs by pure increase of clock rate (exceptions: Turbo Boost, Turbo Core, in some ways larger caches may help)
- multithreading supported by, e.g., `OpenMP` (shared memory), see below
- different from: multiprocessing parallelization via `MPI` (Message Passing Interface) $\rightarrow$ distributed computing (cf. Co-array Fortran) but can be combined: MPI + OpenMP; usually: MPI more complicated (and slower) than OpenMP $\rightarrow$ trend for "larger nodes"

GPGPU

General-purpose computing on graphics processing units $\rightarrow$ further development of graphic cards

- e.g., Nvidia (Tesla, Fermi); AMD (Radeon Instinct)
  $\rightarrow$ *El Capitan* (USA, 1st since Nov 2024 in Top500) with 43 808 nodes (each with AMD-EPYC 24core CPU + GPU MI300A x228) reaches 1.7 ExaFLOPS
  (for comparison: 24 core desktop CPU $\approx$ 8 TeraFLOPS $\rightarrow 7 \times 10^{-6}$ of *Frontier*)
- so-called shaders $\rightarrow$ highly specialized ALUs, often only with single precision (opposite concept: Intel's Larrabee)
- programming (not only graphics) via CUDA (Nvidia) or OpenCL (more general)
- OpenCL $\rightarrow$ parallel programming for arbitrary systems, also NUMA (non-uniform memory access), but very abstract and complex concept and also complicated C-syntax
- CUDA support, e.g., by PGI Fortran compiler $\rightarrow$ simple acceleration without code modifications

OpenMP

OpenMP - Intro and Syntax I

OpenMP - Open Multi-Processing

- for shared-memory systems (e.g., multi core) per node
- directly available in g++, gfortran, and Intel compilers
- insertion of so-called OpenMP (pragma) directives :

Example: for loop

C++

```
#include <omp.h>
 ...
#pragma omp parallel for
for (int i = 1 ; i <= n ; ++i)
{ ... }
```

Fortran

```
      USE omp_lib ! ifort declarations
!$OMP PARALLEL DO
      DO i = 1, n
       ....
      ENDDO
!$OMP END PARALLEL DO
```

instructs parallel execution of the for loop, i.e., there are copies of the loop (different iterations) which run in parallel
$\rightarrow$ only the labeled section runs in parallel

OpenMP - Intro and Syntax II

$\rightarrow$ pragma directives are syntactically seen comments, i.e., invisible for compilers without OpenMP support

- realization during runtime by *threads*
- number of used threads can be set, e.g., by environment variable

```
export OMP_NUM_THREADS=4  # bash
setenv OMP_NUM_THREADS 4  # tcsh
```

$\rightarrow$ obvious: per core only one thread can run at the same time (but: Intel's hyper-threading, AMD's Bulldozer design) $\rightarrow$ in HPC often reasonable:

number of threads = number of physical CPU cores

Caution!

Distributing and joining of threads produces some overhead in CPU / computing time (e.g., copying data) and is therefore only efficient for complex tasks within each thread. Otherwise multithreading can slow down program execution.

OpenMP - Intro and Syntax III

Including the OpenMP library:

C++

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

Fortran

```
!     only needed for declaration of
!     OMP functions etc. with ifort:
!$    use omp_lib
```

→ instructions between `#ifdef _OPENMP` and `#endif` (Fortran: following `!$`) are only executed if compiler invokes OpenMP

Compile with

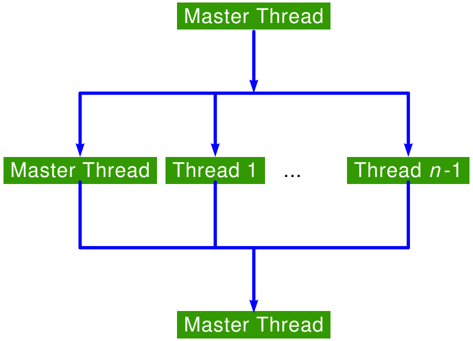
`g++`	`-fopenmp`	
`icpx`	`-fopenmp`	also: `-qopenmp` (deprecated: `-openmp`)
`gfortran`	`-fopenmp`	
`ifort`	`-fopenmp`	also: `-qopenmp` (deprecated: `-openmp`)

OMP functions

Useful: functions specific for OpenMP, e.g., for number of available CPU cores, generated (maximum) number of threads, and current number of threads:

```
omp_get_num_procs()   // number of (logical) processor cores
omp_get_max_threads() // max. number of (automatic) generated threads
omp_get_num_threads() // number of current threads
omp_get_thread_num()  // number of the current thread
```

Join-fork model:
thread that executes `parallel` directive becomes master of thread group with ID$=0$



OMP – Access to variables: `shared` and `private` I

Very important: organization of the accessibility of the involved data, i.e. assign attributes `shared` or `private` to thread variables

`shared`

→ default for variables declared outside the parallel section
data are visible in all threads and can be modified (concurrent access)

```
int sum = 0 ;
#omp pragma parallel for
for (int k = kmax ; k > 0 ; --k) {
     sum += k ; // sum is implicitly shared
```

```
      NSUM  = 0
!$OMP PARALLEL DO
      DO K = KMAX, 1, -1
         NSUM = NSUM + K    ! NSUM is implicitly shared
```

OMP – Access to variables: shared and private II

in contrast to:

private

each thread has its own copy of the data, which are invisible for other threads, especially from outside of the parallel section.
Loop iteration variables are private by default and should be declared in the loop header for clarity:

```
#omp pragma parallel for
for (int k = kmax ; k > 0 ; --k)  // k is implicitly private

!$OMP PARALLEL DO
      DO K = KMAX, 1, -1           ! K is implicitly private
```

Moreover, there are further so-called `data clauses`, e.g., `firstprivate` (initialization before the parallel section), `lastprivate` (last completed thread determines the value of the variable after the parallel section) and many more . . .

→ This is the complicated part of OpenMP!

OMP – Access to variables: shared and private III

Example private

C++:

```
int j, m = 4 ;
#pragma omp parallel for private (j)
for (int i = 0 ; i < max ; i++) {
  j = i + m ;
    ... ;
}
```

Fortran:

```
      INTEGER :: j, m
!$OMP PARALLEL DO PRIVATE (j)
      DO i = 0, max
        j = i + m
        ...
      ENDDO
!$OMP END PARALLEL DO
```

→ loop variable `i` and explicitly private variable `j` as "local" copies in each thread
→ variable `m` implicitly `shared` (be careful in Fortran because of implicit declarations within, e.g. loops)

OMP – critical and reduction I

General form of OpenMP directive for parallelization:

`#pragma omp parallel`

→ `parallel` section also possible without a loop, section is executed per thread (in C/C++: `{ }` block required for multiple commands):

C++:

```
#pragma omp parallel
{
 cout << "Hi!" ;
 cout << endl ;
}
```

Fortran:[†]

```
!$OMP PARALLEL
       print *, "Hi!"
!$OMP END PARALLEL
```

[†] for gfortran the `!$` must start in 1st column

OMP – critical and reduction II

#pragma omp critical

→ within a parallel section
is executed by each thread, but never at the same time (avoiding race conditions for shared resources)

C++:

```
#pragma omp critical
{
  WDrawPoint(myworld, x, y, c) ;
}
```

Fortran:

```
!$OMP CRITICAL
      CALL PGDRAW (x, y)
!$OMP END CRITICAL
```

OMP – critical and reduction III

Example: critical access to an array

C++:

```
#pragma omp parallel for private (j)
for (int i = 0 ; i < nymax ; ++i) {
  for (j = 0 ; j < nxmax ; ++j ) {
    ...
    #pragma omp critical
    subset[i][j] = result ;
  }
}
```

Fortran:

```
!$OMP PARALLEL DO private (j)
      DO i = 0, nymax - 1
        DO j = 0, nxmax - 1
          ...
!$OMP CRITICAL
          subset(i,j) = result
        ENDDO
      ENDDO
```

→ `critical` forces threads to queue, hence slows down execution, better: if possible, use `reduction` clause:

OMP – critical and reduction IV

#pragma omp parallel reduction (*operator*: *list of variables*)

The `reduction` clause defines corresponding (scalar) variables in a parallel section.

Example: summing up with `reduction`

C++:

```
#pragma omp parallel for \
 private(x) reduction(+:sum_this)
for (int i = 1; i <= nmax ; i++) {
  x = 0.01 / (i + 0.5) ;
  sum_this += x ;
}
```

Fortran:

```
!$OMP PARALLEL DO PRIVATE(x)
!$    > REDUCTION(+:sum_this)
      DO i = 1, nmax
        x = 0.01 / (i + 0.5)
        sum_this = sum_this + x
      ENDDO
```

There are a number of allowed operators for reduction, e.g.:

operator	meaning	data type	neutral element / initial value
+,-	sum	`int`, `float`	0
*	product	`int`, `float`	1
&	bitwise and	`int`	all bits 1

Syntax II

- Heads up! OpenMP needs clear syntax for loop parallelization:

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

make sure that your loop has *canonical loop form*, especially the loop iteration variable (here: `i`) is integer as well as variables used for comparison (here: `n`). OpenMP is very picky and might otherwise (e.g., if n is `float`) stop compilation:
`error: invalid controlling predicate.`

- Note that `omp parallel for` / `OMP PARALLEL DO` is the contracted form of

C++:

```
#pragma omp parallel
{
 #pragma omp for
 for ( ... ) {
  ...
 }
}
```

Fortran:

```
!$OMP PARALLEL
!$OMP DO
      ...
!$OMP END DO
!$OMP END PARALLEL
```

OMP – Scheduling I

`schedule(runtime)`

Examples:

```
#pragma omp parallel for schedule (runtime)
```

→ way of distributing the `parallel` section to threads is defined at runtime, e.g., by (bash)

```
export OMP_SCHEDULE="dynamic,1"
```

→ each thread gets a *chunk* of size 1 (e.g., one iteration) as soon as it is ready

```
export OMP_SCHEDULE="static"
```

→ the parallel section (e.g., loop iterations) is divided by the number of threads (e.g., 4) and each thread gets a chunk of the same size

→ static is the default

OMP – Performance and infos

Useful for performance measurement:

`omp_get_wtime()` // → returns the so-called *wall clock time* (not the cpu time)

`omp_get_thread_num()` // → returns the number of the current thread

Weblinks:

`http://www.openmp.org/`
especially the documentation of the specifications:
`http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`