Introduction to Computational Astrophysics

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

SoSe 2019

Techniques of parallelization

Neutron transport with packets I

So far: single neutron n^0
Improvement/speed up: consider "neutron packets", i.e. we follow an ensemble of neutrons (which advances with random $\ell, \cos\theta$ as before)
$\rightarrow$ determine *fraction* of the scattered and captured neutrons

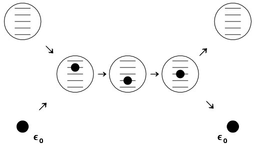
1. scattering:	fraction of scattered n^0: p_s, fraction of absorbed n^0: p_c
2. scattering:	fraction of scattered n^0: p_s^2, fraction of absorbed n^0: $p_c p_s$
*m*th scattering	fraction of scattered n^0: p_s^m, fraction of absorbed n^0: $p_c p_s^{m-1}$

so, after *m*th scattering:
$\rightarrow$ total fraction of captured neutrons:
$f_c = p_c + p_c p_s + p_c p_s^2 + \ldots + p_c p_s^{m-1}$
$\rightarrow$ total fraction of scattered neutrons:
$f_s = p_s^m$
$\rightarrow$ if position $x < 0$: add f_s to f_{refl}
$\rightarrow$ if position $x > t$: add f_s to f_{trans}

Neutron transport with packets II

$\rightarrow$ see also: Lucy (2002), A&A, 384, 725: "Monte Carlo transition probabilities"

- instead of individual photons, use energy packets of same frequency ν ($\epsilon(\nu) = nh\nu$) and same energy ϵ_0 $\rightarrow$ different n
- scattering: $\nu_e = \nu_a$
- absorption leads to re-emission following: $\epsilon(\nu_e) = \epsilon(\nu_a)$, no packet (= energy) lost or created $\rightarrow$ divergence-free radiation field
- macro-atoms with discrete internal states, activation via r-packet (radiative) of appropriate CMF frequency or k-packet (kinetic); active macro-atom performs internal transitions and gets inactive by emission of r- or k-packet



Parallelization

Parallelization

Many runs in MC simulations required for reliable conclusions
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 $\rightarrow$ impact on single-thread performance
- number of cores (1 . . .): $\rightarrow$ impact on multi-thread performance

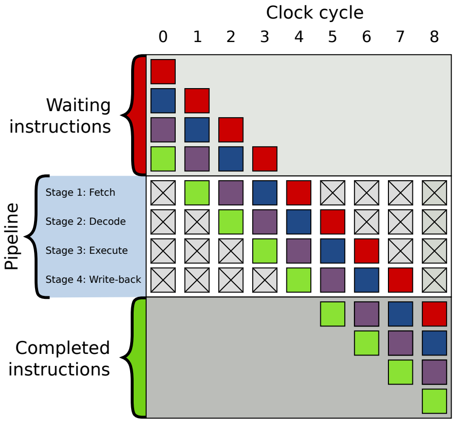
Pipelining I

splitting machine instruction into a series

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 $\rightarrow$ quasi parallel execution, higher throughput



By en:User:Cburnett - Own workThis vector image was created with Inkscape., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1499754

Pipelining II

NetBurst disaster

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

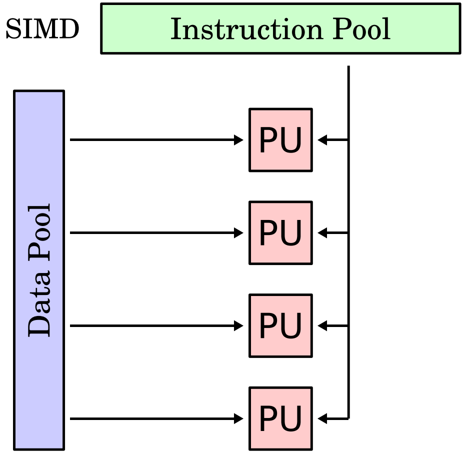
- 20 stages (Pentium III: 10) up to 31 stages (Prescott core)
- smaller number of instructions per clock (IPC) (!)
- increased branch misprediction (also only 10%, improved by 33% for Pentium III)
- larger penalty for misprediction

$\rightarrow$ 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 (6th generation, e.g., Core i7-6700)

SSE and AVX III

- supported by all common compilers, e.g.,
  `ifort -sse4.2`
  `g++ -msse4.1`
- very easy (automatic) optimization, e.g., for unrolled loops
  $\rightarrow$ 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

- originally one core per processor, sometimes several processors per machine/board (supercomputer)
- many units (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 32 cores for desktop (AMD Zen: Ryzen Threadripper 2990WX) or server (AMD Epyc 75. . .), or even more, 72 (Xeon Phi 7290)
- arms race of cores instead of clock rate (NetBurst disaster)

Multi-cores and compilers

- parallelization done by multithreading (from *thread*)
- because of "The free lunch is over" $\rightarrow$ no simple acceleration more of a *single-thread* program (exceptions: Turbo Boost, Turbo Core, in some ways larger caches may help) by pure increase in clock rate
- supported by, e.g., `OpenMP` (shared memory), see below
- different from: multiprocessor (NUMA: Non-Uniform Memory Access) parallelization via MPI $\rightarrow$ distributed computing (cf. Co-array Fortran)

GPGPU

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

- e.g., Tesla, Fermi (Nvidia)
  $\rightarrow$ *Piz Daint* (Swiss, 5th in Top500) with 5 272 Nvidia Tesla K20X processors (each with 2 688 CUDA cores) + 5 272 $\times$ 8 Xeon Cores reaches more than 25 petaflops
- 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 I

OpenMP - Open Multi-Processing

- for shared-memory systems (e.g., multi core)
- 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 ( ... )
{ ... }
```

Fortran

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

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

number of threads = number of 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 III

Including the OpenMP library:

C++

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

Fortran

```
!      only needed for ifort:
!$     use omp_lib
```

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

Compile via

```
g++ -fopenmp
gfortran -fopenmp
ifort -openmp
```

OpenMP IV

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

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

OpenMP V

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

`shared`

→ default
data are visible in all threads and can be modified

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

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 . . .

OpenMP VI

Example `private`

C++:

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

Fortran:

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

→ loop variable `i` and variable `j` as "local" copies in each thread
→ variable `m` implicitly `shared`

OpenMP VII

Some more OpenMP directives:

`#pragma omp parallel`

$\rightarrow$ `parallel` section also possible without a loop, section is executed per thread, `{ }` block required:

C++:

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

Fortran:

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

`#pragma omp critical`

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

OpenMP VIII

`schedule(runtime)`

e.g.,

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

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

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

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

```
export OMP_SCHEDULE "static"
```

$\rightarrow$ 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

OpenMP IX

Useful for performance measurement:

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

`omp_get_thread_num()` // $\rightarrow$ 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`