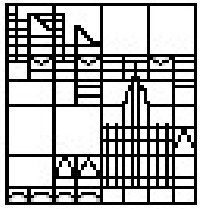
Parallel Processing

Kay Diederichs

Universität
Konstanz



Parallelization

Doing 2 or more computations at the same time, making use of multiple CPUs or computers, leading to reduced wallclock time (speedup).

Examples in crystallography:

- Processing several datasets
- Spot finding in raw data frames
- Estimating signal and background for all pixels of a detector frame
- Varying one or more parameters of a long calculation ("grid search" e.g. Molecular Replacement, or weight optimization in Phenix.refine)
- Scaling data sets, or electron density or structure factor calculation, using many reflections/atoms, performing essentially the same calculation for each reflection/atom

Overview

A hierarchy of parallelization opportunities

„graininess“	*ordering requirements („synchronisation“)*	*Hardware*	*Software*	*Examples*
very coarse	none: calculations are ~ independent	computers + access to storage	e.g. `xterm` sessions	processing several experimental data sets at the same time
coarse	weak: a few synchronization points, e.g. 1/sec	+ LAN	`ssh` + public keys; GNU `parallel`	XDS with MAXIMUM_NUMBER_OF_JOBS >1
medium	frequent synchronization, e.g. 100/sec	+ dedicated fast network protocol	MPI, CoArray Fortran	climate models; QM; hydrodynamics
fine	e.g. 10.000/sec	RAM=shared memory	OpenMP	XDS, Phaser, SHELXL+D, ...
very fine	> 1.000.000/sec	CPU + 1/2/3-level caches	Vectorization (SIMD)	high-level programming language +/- directives

Why parallelize?

- because human time is not well spent when waiting
- we can try more parameters to get the best result
- sometimes new ideas/new science needs heavy computing

Should I parallelize?

- only if the time saved (by you and other users of your program) is significantly longer than the time spent for parallelization (algorithm adapation; implementation; tests)

How to parallelize?

- coarse grained: often „embarassingly parallel“; easy
- fine grained: OpenMP basics are rather intuitive
- very fine grained: mostly a matter of compiler options

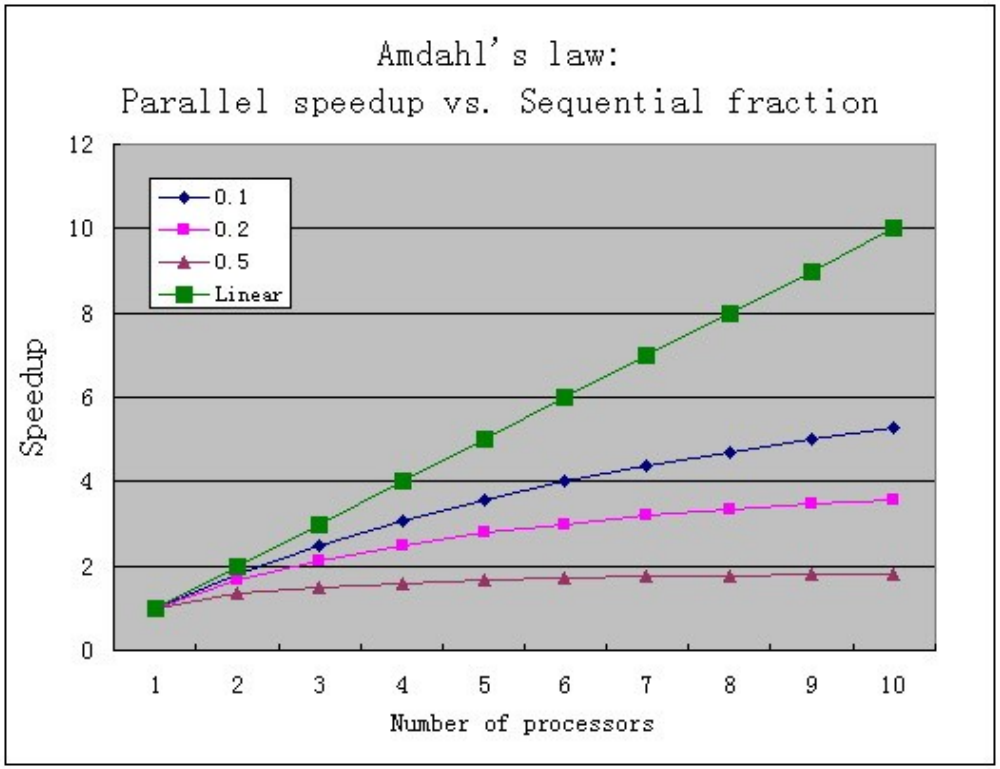
Lessons from past experience

1) parallelization is a bad substitute for a better algorithm
2) only when you are sure that the algorithm is the best and the implementation is clean should parallelization be considered
3) one often has to adapt the algorithm to parallelization
4) the coarser the better – the finer, the more overhead
5) parallelization adds another level of complexity. This makes debugging more difficult

Speedup is limited: Amdahl's Law

If P = time fraction of parallel part, and
1-P = time fraction of serial (sequential) part
N = number of processors

then parallel speedup is $\frac{1}{(1-P)+\frac{P}{N}}$ (Example: P=0.8 N=4: 2.5)



OpenMP Overview

- Hardware aspects
- the OpenMP API (most important facts only!)
- example: parallelization of SHELXL, CNS, XDS
- Speedup findings

Hardware

- since ~1998: affordable 2-socket (Intel/AMD)
- 2002: HyperThreading (Intel)
- 2005: 2-core (AMD, Intel)
- 2007: 4-core (Intel)
- 2008: 4-core (AMD)
- 2009: 6-core/ 8-core (Intel, AMD), + HT
- 2010: 6+6-core (Intel), 12-core (AMD)

2010+ : „cheap“ 2*(6+6) and 4*12 (Intel, AMD)

Speed maxed out at ~ 4 GHz

GPU: potentially thousands of „CPUs“ on one board

OpenMP: **Open M**ulti**P**rocessing

- A **standard** developed under the review of many major software and hardware developers, government, and academia
- facilitates *simple* and *incremental* development of programs to take advantage of SMP architectures
- SMP: Symmetric multi-processing, *n* processors/cores (usually *n*=2, 4, 8, 16, 32 ...) in a single machine..
- Shared memory - memory is local to all processors in a machine: multi-processor / multi-core but also NUMA (large Intel-Xeon or AMD-Opteron)
- *not* for distributed memory (but Intel *Cluster OpenMP* exists, and for Fortran2008 *CoArrays* is standardized)
- May be combined with MPI

OpenMP Architecture Review Board

Compaq / Digital
Hewlett-Packard Company
Intel Corporation
International Business Machines (IBM)
Kuck & Associates, Inc. (KAI)
Silicon Graphics, Inc.
Sun Microsystems, Inc.
U.S. Department of Energy ASC program

Endorsing software vendors

Absoft Corporation
Edinburgh Portable Compilers
GENIAS Software GmbH
Myrias Computer Technologies, Inc.
The Portland Group, Inc. (PGI)

Documentation Release History

Oct 1997: Fortran version 1.0 (63 pages)
Oct 1998: C/C++ version 1.0 (85 pages)
Nov 2000: Fortran version 2.0
Mar 2002: C/C++ version 2.0
May 2008: C/C++ and Fortran version 3.0
Jul 2013: C/C++ and Fortran version 4.0 (320 pages)

http://www.openmp.org

OpenMP: What is it?

- OpenMP language support:

  Fortran (Fortran77+), C, C++

- OpenMP API is comprised of:

  - Compiler directives
  - Library routines
  - Environment variables

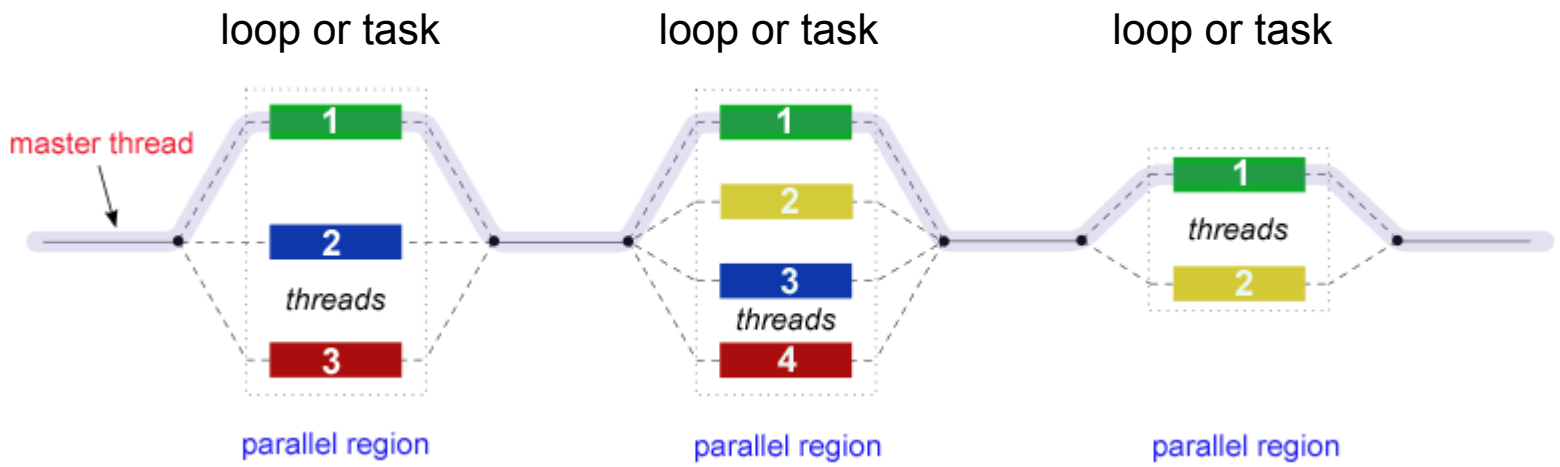
- Compilers supporting OpenMP:

  GCC (free on all OS), Intel (Linux: free for developers), Portland Group (PGI; Linux: free for academics), Oracle Solaris studio (Linux: free), Microsoft, IBM, HP, Cray …

- compatibility is usually very good

Fork-Join Parallelism

threads (processes) form a “Master-Worker Team”



The threads communicate through shared variables (shared memory)

- Overhead! The savings in wallclock time need to *amortize* thread fork and communication costs
- Again, parallelization at outer loop (coarsest) level is the most efficient; finest grain is > 1000 operations

The most fundamental OpenMP directive:

PARALLEL DO

```
     integer i, n
     real x(100000)
       ...
     n = 100000
!$omp parallel do shared(x,n) private(i)     ! directive with clauses
     do i = 1, 100000
          x(i) = x(i) + exp(i/n)
          call doalotofwork(x(i))
     end do
       ...
```

… but parallelization is not always easy/possible:

```
1 // Do NOT do this. It will fail due to data dependencies.
2 // Each loop iteration writes a value that a different
  iteration reads.
3     #pragma omp parallel for
4     for (i=2; i < 10; i++)
5     {
6        factorial[i] = i * factorial[i-1];
7     }
```

There are several types of such „data dependencies“: see e.g. http://www.ncsu.edu/hpc/Courses/8shared.html#classify

If not removed: „data **races**“=wrong results, or **deadlocks**

Data scope clauses:

Shared memory programming: OpenMP defaults to *shared* data (which can be accessed by all threads)

- `shared(var,...)` : explicitly share variables across all threads.

- `private(var,...)`: uninitialized, thread local instance of the variable, cannot be accessed by other threads

- `firstprivate(var,...)` : initialize local instance of the variable from master thread

- Functions called within a parallel region have their own private stack space

`shared` *versus* `private` is the biggest conceptual stumbling block for beginners

Scheduling clauses:

- `schedule(static [,chunk])`

Threads get a chunk of data to iterate over (default)

- `schedule(dynamic [,chunk])`

Threads grab chunk iterations off work queue until all work is exhausted

- `schedule(guided [,chunk])`

Threads grab a large chunk size first, and decrease the size to the specified size as the computation progresses

- `schedule(runtime)`

Threads use the schedule defined at runtime by the OMP_SCHEDULE environment variable

Compatibility with non-OpenMP compilers

Parallelization is transparent and incremental

Conditional compilation using special comment line:

```
!$      integer omp_get_num_threads
        n=1
!$      n=omp_get_num_threads()
```

but also

```
!$      if (n.GT.1000) then
!$        m = 10
!$      else
          m = 100
!$      endif
```

This allows to use the same source code for all (OpenMP and non-OpenMP) compilers, but activates special code if running on a multiprocessor machine.

Reduction

It is often necessary that all threads accumulate (or perform some other operation on) a single variable, and return a single value at the end of the computation

OpenMP provides a reduction clause:

`reduction(op: list)`

`op` can be `+ - * / min() max()` , and certain binary bitwise operators (OpenMP 4.0 allows user-defined reductions). Example:

```
sumx=0.

!$omp parallel do shared(x,n) private(i)
  reduction(+:sumx)
 do i=1,n
   sumx = sumx + x(i)
 end do
```

Synchronization Constructs

If two or more threads write to the *same* shared variables then these updates must be protected from “race conditions”. OpenMP provides:

`!$omp critical`
Creates critical section i.e. serializes: only one thread can enter at a time (ends at `!$omp end critical`). High latency.

`!$omp atomic`
Special version of critical, for atomic ops (e.g. updating a single memory location). Very low latency.

`!$omp barrier`
Synchronization point for all threads in parallel region

`!$omp ordered`
Forces sequential execution of the following block (e.g. for I/O)

Environment Variables

OMP_NUM_THREADS: sets maximum number of threads to use

OMP_SCHEDULE: scheduling algorithm for parallel regions

OMP_DYNAMIC (TRUE, FALSE): dynamic adjustment of number of threads for parallel regions

OMP_NESTED (TRUE, FALSE): enables or disables nested parallelism

OpenMP Library Routines: prefixed with `omp_`

control and query the parallel execution environment e.g.

```
subroutine OMP_SET_NUM_THREADS()
function OMP_GET_MAX_THREADS()
function OMP_GET_THREAD_NUM()
function OMP_IN_PARALLEL()
```

Low-level locking routines

```
subroutine OMP_INIT_LOCK()
subroutine OMP_DESTROY_LOCK()
subroutine OMP_SET_LOCK()
subroutine OMP_UNSET_LOCK()
function OMP_TEST_LOCK()
```

An OpenMP Example:

part of crystallographic program (Fortran77)

```
...
!$omp parallel do shared(fc,nref,natom,xyz,hkl,fcabs,fo,
!$omp& fosum,fcsum,rfnum) private(i,j,fcmplx)
!$omp& reduction(+:fosum,fcsum)
      do i = 1,nref
        fc = (0.,0.)
        do j = 1,natom
          fc = fc+exp(2.*pi*(0.,1.)*(xyz(1,j)*hkl(1,i)+
     &             xyz(2,j)*hkl(2,i)+xyz(3,j)*hkl(3,i)))
        continue
        fosum    = fosum+fo(i)
        fcabs(i) = abs(fc)
        fcsum    = fcsum+fcabs(i)
      continue
!$omp end parallel do
...
```

Tools for analyzing OpenMP code

Correctness: *Thread Checker* (Intel)

Detects thread-correctness issues including data-races, dead-locks, and threads stalls.

Performance: *Thread Profiler* (Intel)

Analyzes threading performance and enables you to visualize thread interactions. *Simple alternative: gprof*

Intel compilers and tools: Linux versions are free for developers, and cheap for academics

Similar tools from Oracle (Sun), for their free compilers.

Crystallographic programs using OpenMP

- *BEAST molecular replacement (precursor of PHASER)*
- *ESSENS real-space molecular replacement*
- *CNS structure factors/derivatives*
- *SHELXL structure factors/derivatives*
- *XDS data reduction (Wolfgang Kabsch)*
- *SHARP heavy atom refinement (Globalphasing)*
- *SHELXD substructure analysis (George Sheldrick)*
- *phenix.refine refinement (Phenix; less well supported due to interference w/ Python threads)*

SHELXL parallelization with OpenMP

- *profiling* - find those loops which take longest wallclock time
- modify structure of the program a bit such that the most time-consuming partscan be changed into PARALLEL DO loops
- test and verify correctness
- has been working quite well
- George Sheldrick has since rewritten SHELXL; kept OpenMP

Timings of *SHELXL (not the current version)*

- **4 parallel regions**
- **Dual-Xeon 2.8GHz (+/- Hyperthreading)**

2nd CPU disabled in BIOS :

1 thread	no HT	3h 3min (100%)
2 threads	HT	2h 32min (83%)

→ Hyperthreading gives 17% speedup

both CPUs enabled in BIOS:

2 threads	no HT	1h 33min (51%)
4 threads	HT	1h 17min (42%)

→ a "logical" processor (HT) gives 17% speedup
→ a "physical" processor (Dual-CPU) gives almost 2-fold (98%) speedup
→ *SHELXL* has 2% serial code

Amdahl's Law: speedup(n) = $\frac{1}{(0.98/n+0.02)}$

[speedup(8) = 1/(0.98/8+0.02)=7.0 speedup(16)=12.3]

CNS

- FFT: possible to parallelize, but little speedup
- Memory subsystem, "false sharing", max. aggregated throughput of bus and other subtleties need to be considered
- Generally: better to use FFTW (BLAS/LAPACK)
- Subgrid algorithm: A. T. Brünger (1989) A memory-efficient fast Fourier transformation algorithm for crystallographic refinement on supercomputers *Acta Cryst.* (1989). A**45**, 42-50
- Speedup on Quad-core: more than 2 (Amdahl's Law!)
- another few % gains by HyperThreading

XDS

two levels of parallelization:

- shell-level (via ssh) dividing a dataset into JOBS (up to 99); synchronization only at end.
- thread-level (OpenMP) dividing a batch of frames (5° rotation) among CPUs (up to 32); synchronization every 5°.
- very clean code
- quite good speedup

Perspectives of OpenMP

• GCC compilers support OpenMP 4.0 since v4.9 (June-2015)

• OpenMP 4.0 can *offload* computations to other devices (Xeon Phi) but not (yet) to GPUs. The latter needs OpenACC.

• Some compilers have auto-parallelization (`-parallel`) of simple DO loops with automatical specification of SHARED, PRIVATE, FIRSTPRIVATE and REDUCTION clauses. In my experience, this is good for finding places where code re-arrangement and/or inserting directives should help. Most of the existing code however cannot be auto-parallelized due to “data dependencies”.

• integration with tools for checking correctness, and performance

• *Cluster OpenMP* (Intel; non-free) is able to use distributed shared memory, i.e. to run OpenMP across several machines, using a few proprietary extensions. But *CoArray Fortran* may be a better (Fortran2008 and GCC v5-supported) alternative.

Summary

- OpenMP: a simple way to make programs run faster on multi-core machines; speedups of >10 are reachable on 32-CPU hardware
- OpenACC: higher speedups are in reach, using GPUs
- OpenMP appears to move towards OpenACC: the future holds promise

References

OpenMP website: http://www.openmp.org/

Excellent introduction to OpenMP (version 3.1):
https://computing.llnl.gov/tutorials/openMP/

Youtube: http://tinyurl.com/OpenMP-Tutorial (2013)

Diederichs, K. (2000): Computing in macromolecular crystallography using a parallel architecture *J. Appl. Cryst.* **33**, 1154-1161

R. Gerber: https://software.intel.com/en-us/articles/getting-started-with-openmp and references therein (2012)

B. Chapman, G. Jost, R. van der Pas: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2008.

M Süß and C. Leopold: Common Mistakes in OpenMP and How To Avoid Them.
http://wwwi10.lrr.in.tum.de/~gerndt/home/Teaching/EfficientHPCProgramming/CommonMistakesInOpenMPAndHowToAvoidThem.pdf