AMath 483/583 — Lecture 13 — April 25, 2011

Today:

- Amdahl's law
- Speed up, strong and weak scaling
- OpenMP

Monday:

- More OpenMP

Read: Class notes and references

Chapter 2 and Section 6.3 of T. Rauber and G. Rünger,
Parallel Programming For Multicore and Cluster Systems

There is a new directory `$CLASSHG/codes/openmp`

Amdahl's Law

Typically only part of a computation can be parallelized.

Suppose 50% of the computation is inherently sequential,
and the other 50% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Amdahl's Law

Typically only part of a computation can be parallelized.

Suppose 50% of the computation is inherently sequential,
and the other 50% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Answer: At most a factor of 2, no matter how many processors.

The sequential part is taking half the time and that time is still required even if the parallel part is reduced to zero time.

Amdahl's Law

Suppose 10% of the computation is inherently sequential,
and the other 90% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Amdahl's Law

Suppose 10% of the computation is inherently sequential,
and the other 90% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Answer: At most a factor of 10, no matter how many processors.

The sequential part is taking 1/10 of the time and that time is still required even if the parallel part is reduced to zero time.

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential, and the other $(1 - 1/S)$ can be parallelized.

Then can gain at most a factor of S, no matter how many processors.

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential, and the other $(1 - 1/S)$ can be parallelized.

Then can gain at most a factor of S, no matter how many processors.

If T_S is the time required on a sequential machine and we run on P processors, then the time required will be (at least):

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential, and the other $(1 - 1/S)$ can be parallelized.

Then can gain at most a factor of S, no matter how many processors.

If T_S is the time required on a sequential machine and we run on P processors, then the time required will be (at least):

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Note that

$$T_P \to (1/S)T_S \quad \text{as} \quad P \to \infty$$

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential $\Longrightarrow$

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Example: If 5% of the computation is inherently sequential ($S = 20$), then the reduction in time is:

P	T_P
1	T_S
2	$0.525T_S$
4	$0.288T_S$
32	$0.080T_S$
128	$0.057T_S$
1024	$0.051T_S$

Speedup

The ratio T_S/T_P of time on a sequential machine to time running in parallel is the speedup.

This is generally less than P for P processors.

Perhaps much less.

Amdahl's Law plus overhead costs of starting processes/threads, communication, etc.

Speedup

The ratio T_S/T_P of time on a sequential machine to time running in parallel is the speedup.

This is generally less than P for P processors.

Perhaps much less.

Amdahl's Law plus overhead costs of starting processes/threads, communication, etc.

Caveat: May (rarely) see speedup greater than P...

For example, if data doesn't all fit in one cache but does fit in the combined caches of multiple processors.

Scaling

Some algorithms scale better than others as the number of processors increases.

Typically interested on how well algorithms work for large problems requiring lots of time, e.g.

Particle methods for n particles,
algorithms for solving systems of n equations,
algorithms for solving PDEs on $n \times n \times n$ grid in 3D,

Scaling

Some algorithms scale better than others as the number of processors increases.

Typically interested on how well algorithms work for large problems requiring lots of time, e.g.

- Particle methods for n particles,
- algorithms for solving systems of n equations,
- algorithms for solving PDEs on $n \times n \times n$ grid in 3D,

For large n, there may be lots of inherent parallelism.

But depends on many factors:

- dependencies between calculations,
- communication as well as flops,
- nature of problem and algorithm chosen.

Scaling

Typically interested on how well algorithms work for large problems requiring lots of time.

Strong scaling: How does the algorithm perform as the number of processors P increases for a fixed problem size n?

Any algorithm will eventually break down (consider $P > n$)

Scaling

Typically interested on how well algorithms work for large problems requiring lots of time.

Strong scaling: How does the algorithm perform as the number of processors P increases for a fixed problem size n?

Any algorithm will eventually break down (consider $P > n$)

Weak scaling: How does the algorithm perform when the problem size increases with the number of processors?

E.g. If we double the number of processors can we solve a problem "twice as large" in the same time?

Weak scaling

What does “twice as large” mean?

Depends on how algorithm complexity scales with n.

Example: Solving linear system with Gaussian elimination requires $O(n^3)$ flops.

Doubling n requires 8 times as many operations.

Problem is “twice as large” if we increase n by a factor of $2^{1/3} \approx 1.26$.

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.

n^3 grid points $\implies$ linear system with this many unknowns.

If we used Gaussian elimination (very bad idea!) we would require $\sim (n^3)^3 = n^9$ flops.

Doubling n would require $2^9 = 512$ times more flops.

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.

n^3 grid points $\implies$ linear system with this many unknowns.

If we used Gaussian elimination (very bad idea!) we would require $\sim (n^3)^3 = n^9$ flops.

Doubling n would require $2^9 = 512$ times more flops.

Good iterative methods can do the job in $O(n^3)\log_2(n)$ work or less. (e.g. multigrid).

Developing better algorithms is as important as better hardware!!

Speedup for problems like steady state heat equation

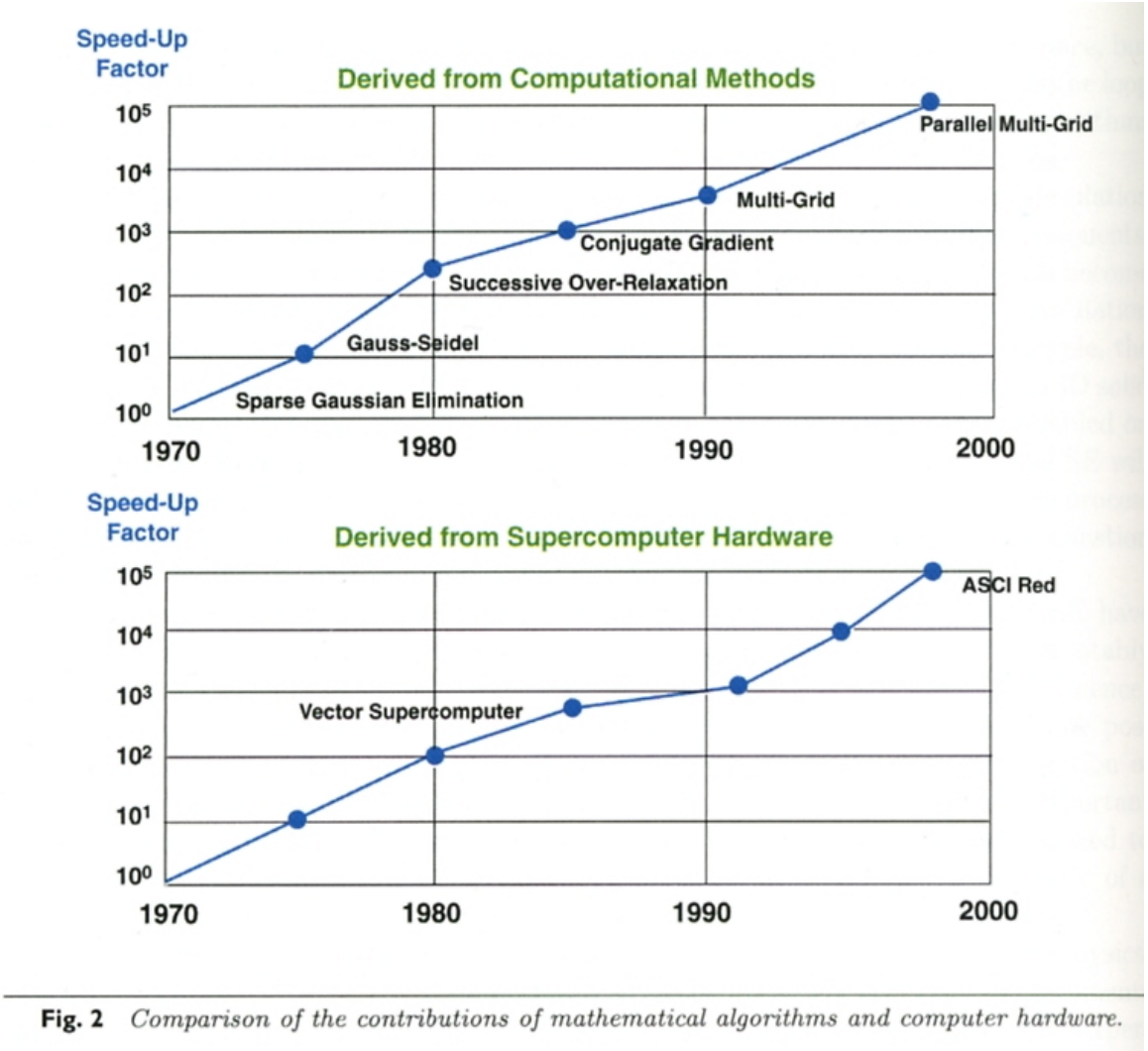


Fig. 2 *Comparison of the contributions of mathematical algorithms and computer hardware.*

Source: SIAM Review

OpenMP

"Open Specifications for MultiProcessing"

Standard for shared memory parallel programming.
For shared memory computers, such as multi-core.

For Fortran (77,90,95), C and C++, on Unix, Windows NT and other platforms.

Maintained by the OpenMP Architecture Review Board (ARB) (non-profit group of organizations that interpret and update OpenMP, write new specs, etc. Includes Compaq/Digital, HP, Intel, IBM, KAI, SGI, Sun, DOE. (Endorsed by software and application vendors).

OpenMP References

http://www.openmp.org

R. Chandra, L. Dagum, et. al., *Parallel Programming in OpenMP*, Academic Press, 2001.

https://computing.llnl.gov/tutorials/openMP/

http://www.nersc.gov/nusers/help/tutorials/openmp>

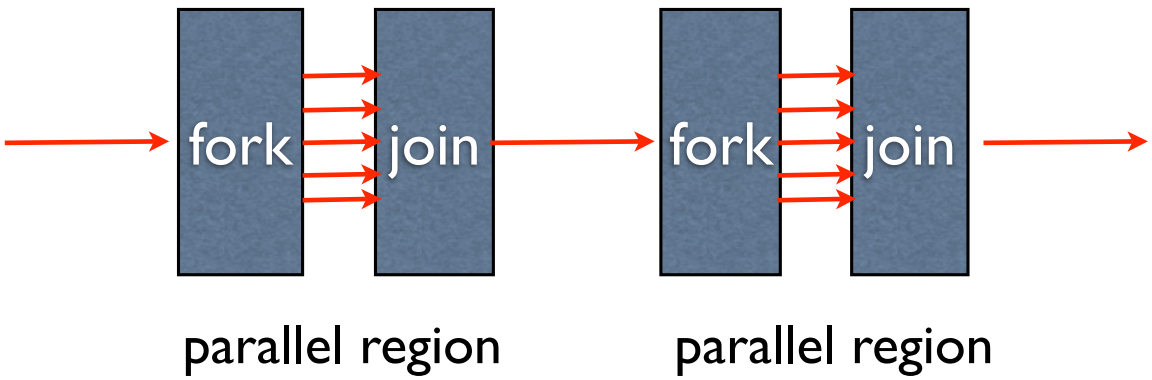
Other references and classes in bibliography of notes

Some slides borrowed from above sources and Marsha Berger, NYU

OpenMP — Basic Idea

Explicit programmer control of parallelization using fork-join model of parallel execution

- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread. (similar to fork-join of Pthreads)



OpenMP — Basic Idea

- Rule of thumb: One thread per processor (or core),
- Dependencies in parallel parts require synchronization between threads
- User inserts compiler directives telling compiler how statements are to be executed
  - which parts are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads
- Compiler generates explicit threaded code
- User's job to remove dependencies in parallel parts or use synchronization. (Tools exist to look for race conditions.)

OpenMP compiler directives

Uses compiler directives that start with `!$` (pragmas in C.)

These look like comments to standard Fortran but are recognized when compiled with the flag `-fopenmp`.

OpenMP statements:

Ordinary Fortran statements conditionally compiled:

```
!$ print *, "Compiled with -fopenmp"
```

OpenMP compiler directives, e.g.

```
!$omp parallel do
```

Calls to OpenMP library routines:

```
use omp_lib     ! need this module
!$ call omp_set_num_threads(2)
```

OpenMP directives

```
!$omp directive   [clause ...]
                  if (scalar_expression)
                  private (list)
                  shared (list)
                  default (shared | none)
                  firstprivate (list)
                  reduction (operator: list)
                  copyin (list)
                  num_threads (integer-expression)
```

A few OpenMP directives

```
!$omp parallel [clause]
   ! block of code
!$omp end parallel
```

```
!$omp parallel do [clause]
   ! do loop
!$omp end parallel do
```

```
!$omp barrier
  ! wait until all threads arrive
```

Several others we'll see later...

OpenMP

API also provides for (but implementation may not support):

- Nested parallelism (parallel constructs inside other parallel constructs)
- Dynamically altering number of threads in different parallel regions

The standard says nothing about parallel I/O.

OpenMP provides "relaxed-consistency" view of memory.

Threads can cache their data and are not required to maintain exact consistency with real memory all the time.

```
!$omp flush
```

can be used as a memory fence at a point where all threads must have consistent view of memory.

OpenMP test code

```
program test
   use omp_lib
   integer :: thread_num

   ! Specify number of threads to use:
   !$ call omp_set_num_threads(2)

   print *, "Testing openmp ..."

   !$omp parallel
   !$omp critical
   !$ thread_num = omp_get_thread_num()
   !$ print *, "This thread = ",thread_num
   !$omp end critical
   !$omp end parallel
end program test
```

OpenMP test code output

Compiled with OpenMP:

```
$ gfortran -fopenmp test.f90
$ ./a.out

 Testing openmp ...
 This thread =            0
 This thread =            1
```

(or threads might print in the other order!)

Compiled without OpenMP:

```
$ gfortran test.f90
$ ./a.out
 Testing openmp ...
```

OpenMP test code

```
! Specify number of threads to use:
!$ call omp_set_num_threads(2)
```

Can specify more threads than processors, but they won't execute in parallel.

The number of threads is determined by (in order):

- Evaluation of **if** clause of a directive
  (if evaluates to zero or false $\Longrightarrow$ serial execution)
- Setting the **num_threads** clause
- the **omp_set_num_threads()** library function
- the **OMP_NUM_THREADS** environment variable
- Implementation default

OpenMP test code

```
!$omp parallel
!$omp critical
!$ thread_num = omp_get_thread_num()
!$ print *, "This thread = ",thread_num
!$omp end critical
!$omp end parallel
```

The `!$omp parallel` block spawns two threads and each one works independently, doing all instructions in block.

Threads are destroyed at `!$omp end parallel`.

However, the statements are also in a `!$omp critical` block, which indicates that this section of the code can be executed by only one thread at a time, so in fact they are not done in parallel.

So why do this? The function `omp_get_thread_num()` returns a unique number for each thread and we want to print both of these.

OpenMP test code

Incorrect code without critical section:

```
!$omp parallel
!$ thread_num = omp_get_thread_num()
!$ print *, "This thread = ",thread_num
!$omp end parallel
```

Why not do these in parallel?

1. If the prints are done simultaneously they may come out garbled (characters of one interspersed in the other).

2. `thread_num` is a shared variable. If this were not in a critical section, the following would be possible:

   Thread 0 executes function, sets `thread_num=0`
   Thread 1 executes function, sets `thread_num=1`
   Thread 0 executes print statement: "This thread = 1"
   Thread 1 executes print statement: "This thread = 1"

There is a data race or race condition.

OpenMP test code

Could change to add a private clause:

```
!$omp parallel private(thread_num)

!$ thread_num = omp_get_thread_num()

!$omp critical
!$ print *, "This thread = ",thread_num
!$omp end critical
!$omp end parallel
```

Then each thread has it's own version of the `thread_num` variable.

OpenMP parallel do loops

```
!$omp parallel do
do i=1,n
   ! do stuff for each i
   enddo
!$omp end parallel do    ! OPTIONAL
```

indicates that the do loop can be done in parallel.

Requires:

what's done for each value of i is independent of others
Different values of i can be done in any order.

The iteration variable `i` is private to the thread: each thread has its own version.

By default, all other variables are shared between threads unless specified otherwise.

OpenMP parallel do loops

This code fills a vector y with function values that take a bit of time to compute:

```
! fragment of $CLASSHG/codes/openmp/yeval.f90

dx = 1.d0 / (n+1.d0)

!$omp parallel do private(x)
do i=1,n
   x = i*dx
   y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

Elapsed time for $n = 10^8$, without OpenMP: about 9.3 sec.

Elapsed time using OpenMP on 2 processors: about 5.0 sec.

Memory stack

Note: Parallel threads use stack and you may need to increase the limit (e.g. on the VM):

```
$ gfortran -fopenmp yeval.f90
$ ./a.out
Segmentation fault

$ ulimit -s
8192

$ ulimit -s unlimited

$ ./a.out
Using OpenMP with   2 threads
 Filled vector y of length   100000000
```

Memory: Heap and Stack

Memory devoted to data for a program is generally split up:

Heap: Dynamically allocated memory — memory allocator looks for free block of memory, keeps track of free list, does garbage collection, etc.

Stack: Block of memory where space is allocated on "top" of the stack as needed and "popped" off the stack when no longer needed. Last in – first out (LIFO).

Fast relative to heap allocation.

Natural way to allocate storage for nested subroutine or function calls: If A calls B calls C, then when the variables used by C are popped off the stack, we're back to the variables of B.

Memory: Heap and Stack

Memory devoted to data for a program is generally split up:

Heap: Dynamically allocated memory — memory allocator looks for free block of memory, keeps track of free list, does garbage collection, etc.

Stack: Block of memory where space is allocated on "top" of the stack as needed and "popped" off the stack when no longer needed. Last in – first out (LIFO).

Fast relative to heap allocation.

Natural way to allocate storage for nested subroutine or function calls: If A calls B calls C, then when the variables used by C are popped off the stack, we're back to the variables of B.

Private variables for threads also put on stack, popped off when parallel block ends.