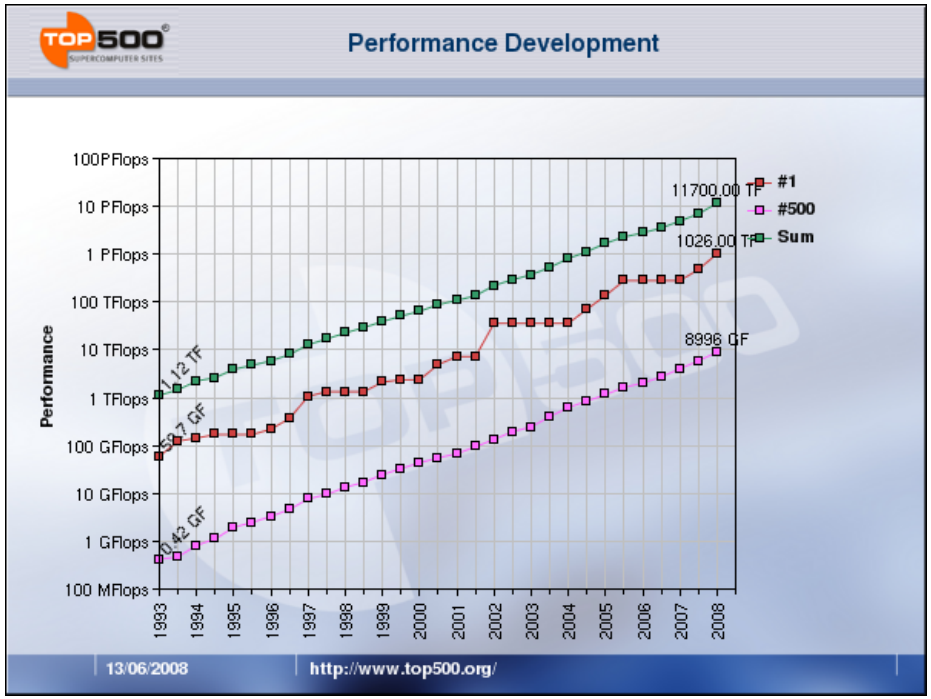
Parallel Computing

- Basic concepts
- Shared vs. distributed memory
- OpenMP (shared)
- MPI (shared or distributed)

Moore's Law: Processor speed doubles every 18 months.

$\Longrightarrow$ factor of 1024 in 15 years.

Going forward: Number of cores doubles every 18 months.



Top: Total computing power of top 500 computers

Middle: #1 computer

Bottom: #500 computer

`http://www.top500.org`

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has it's own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

General purpose GPU computing: (Graphical Processor Unit)

Hybrid: Often clusters of multicore/GPU machines!

For example, multi-threaded program on dual-core computer.

Thread:

A thread of control: program code, program counter, call stack, small amount of thread-specific data (registers, L1 cache).

Shared memory and file system.

Threads may be spawned and destroyed as computation proceeds.

Languages like OpenMP.

Some issues:

Limited to modest number of cores when memory is shared.

Multiple threads have access to same data — convenient and fast.

Contention: But, need to make sure they don't conflict (e.g. two threads should not write to same location at same time).

Dependencies, synchronization: Need to make sure some operations are done in proper order!

May need cache coherence: If Thread 1 changes x in its private cache, other threads might need to see changed value.

A process is a thread that also has its own private address space.

Multiple processes are often running on a single computer
(e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.

Greater latency in sharing data.

Processes communicate by passing messages.

Languages like MPI — Message Passing Interface.

Some issues:

Often more complicated to program.

High cost of data communication between processes.
Want to maximize processing on local data relative to communication with other processes.

Often need to partition problem domain into subdomains,
(e.g. domain decomposition for PDEs)

Generally requires coarse grain parallelism.

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.

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.

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

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

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	$0.051T_S$

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.

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 this depends on many factors:

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

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?

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 this depends on many factors:

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

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!!

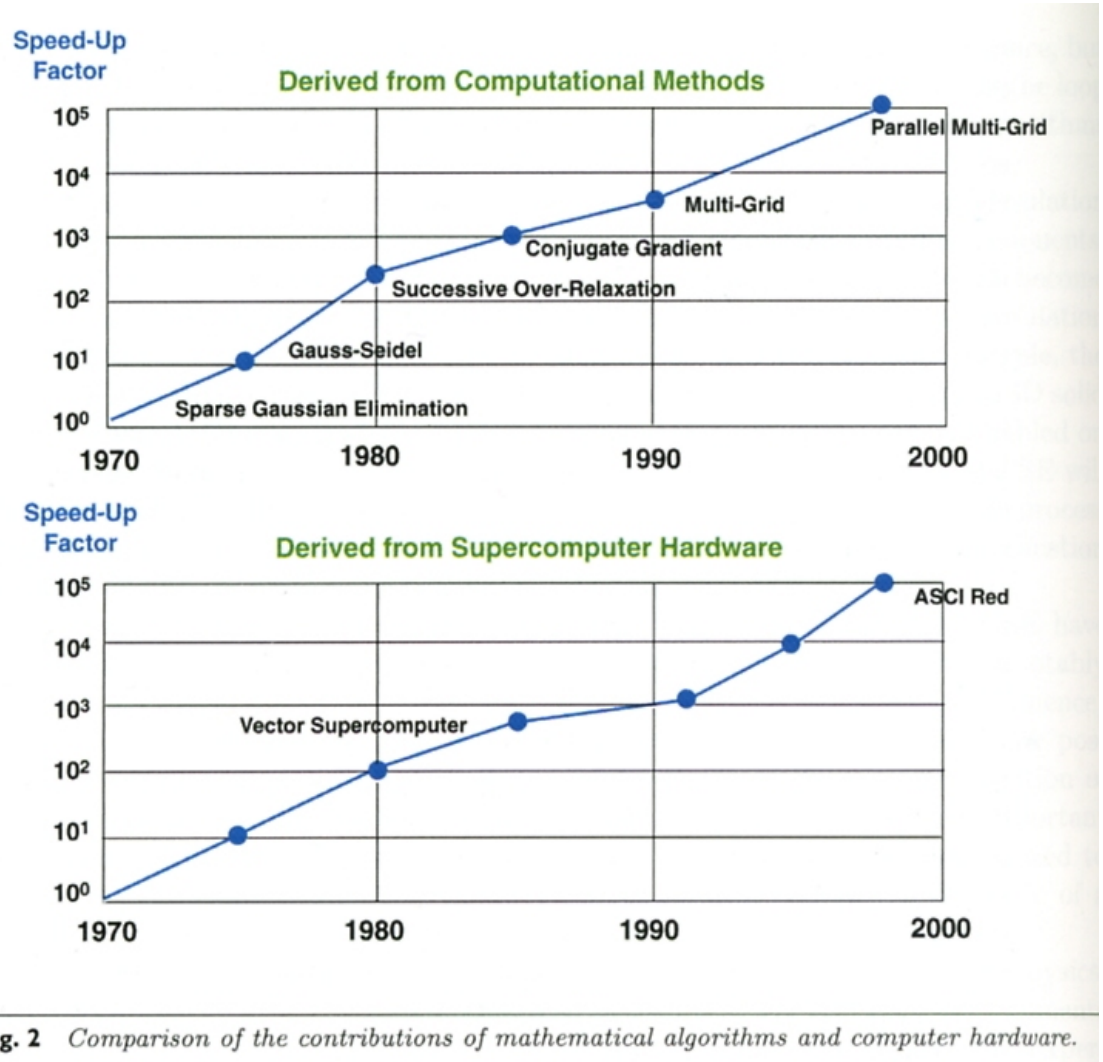


Fig. 2 *Comparison of the contributions of mathematical algorithms and computer hardware.*

Source: SIAM Review 43(2001), p. 168.

“Open Specifications for MultiProcessing”

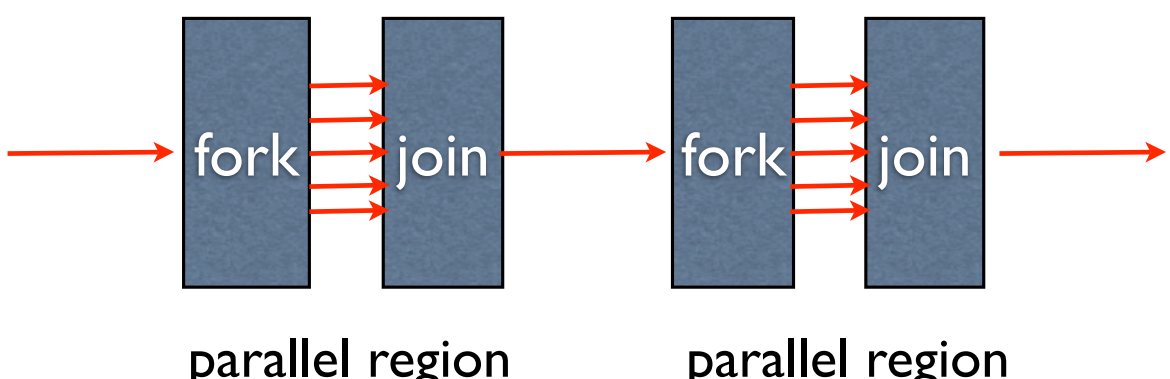
Standard for shared memory parallel programming.
For shared memory computers, such as multi-core.

Can be used with Fortran (77/90/95/2003), C and C++.

Complete specifications at **`http://www.openmp.org`**

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.



- Rule of thumb: One thread per processor (or core),
- 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
- Dependencies in parallel parts require synchronization between threads
- User's job to remove dependencies in parallel parts or use synchronization. (Tools exist to look for race conditions.)

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)
```

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

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

```
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
```

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 ...
```

```
! 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

```
!$omp parallel
!$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.

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.

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.

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

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

```
! fragment of $UWHPSC/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.

This code is not correct:

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

By default, `x` is a shared variable.

Might happen that:

- Processor 0 sets `x` properly for one value of `i`,
- Processor 1 sets `x` properly for another value of `i`,
- Processor 0 uses `x` but is now incorrect.

Correct version:

```
!$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
```

Now each thread has its own version of `x`.

Iteration counter `i` is private by default.

Note that `dx, n, y` are shared by default. OK because:

`dx, n` are used but not changed,
`y` is changed, but independently for each `i`

Incorrect code:

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

Specifying `dx` private won't work here.

This will create a private variable `dx` for each thread but it will be uninitialized.

Will run but give garbage.

Could fix with:

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

The firstprivate clause creates private variables and initializes to the value from the master thread prior to the loop.

There is also a lastprivate clause to indicate that the last value computed by a thread (for $i = n$) should be copied to the master thread's copy for continued execution.

```
! from $UWHPSC/codes/openmp/private1.f90

    n = 7
    y = 2.d0
    !$omp parallel do firstprivate(y) lastprivate(y)
    do i=1,n
       y = y + 10.d0
       x(i) = y
       !omp critical
       print *, "i = ",i,"  x(i) = ",x(i)
       !omp end critical
    enddo
    print *, "At end, y = ",y
```

Run with 2 threads: The 7 values of i will be split up, perhaps

$i = 1, 2, 3, 4$ executed by thread 0,

$i = 5, 6, 7$ executed by thread 1.

Thread 0's private y will be updated 4 times, $2 \to 12 \to 22 \to 32 \to 42$

Thread 1's private y will be updated 3 times, $2 \to 12 \to 22 \to 32$

```
! from $UWHPSC/codes/openmp/private1.f90

    n = 7
    y = 2.d0
    !$omp parallel do firstprivate(y) lastprivate(y)
    do i=1,n
       y = y + 10.d0
       x(i) = y
       !omp critical
       print *, "i = ",i,"  x(i) = ",x(i)
       !omp end critical
    enddo
    print *, "At end, y = ",y
```

might produce:

```
      i =             1   x(i) =     12.000000000000
      i =             5   x(i) =     12.000000000000
      i =             2   x(i) =     22.000000000000
      i =             6   x(i) =     22.000000000000
      i =             3   x(i) =     32.000000000000
      i =             7   x(i) =     32.000000000000
      i =             4   x(i) =     42.000000000000
      At end, y =     32.000000000000
```

Order might be different but final y will be from $i = 7$.

Default is that loop iterator is private, other variables shared.

Can change this, e.g.

```
!$omp parallel do default(private) shared(x,z) &
!$omp firstprivate(y) lastprivate(y)
do i=1,n
   etc.
```

With this change, only `x` and `z` are shared.

Note continuation character `&` and continuation line.

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

! master thread continues execution
```

There is an implicit barrier at the end of the loop.

The master thread will not continue until all threads have finished with their subset of `1, 2, ..., n`.

Except if ended by:

```
!$omp end parallel do nowait
```

Loop overhead may not be worthwhile for short loops.
(Multi-thread version may run slower than sequential)

Can use conditional clause:

```
$omp parallel do if (n > 1000)
do i=1,n
    ! do stuff
    enddo
```

If $n \leq 1000$ then no threads are created,
master thread executes loop sequentially.

```
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

The loop on `j` is split up between threads.

The thread handling `j=1` does the entire loop on `i`,
sets `a(1,1), a(2,1), ..., a(n,1)`.

Note: The loop iterator `i` must be declared private!

`j` is private by default, `i` is shared by default.

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

or

```
do j=1,m
   !$omp parallel do
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

But have to make sure loop can be parallelized!

Incorrect code for replicating first column:

```
!$omp parallel do private(j)
do i=2,n
   do j=1,m
      a(i,j) = a(i-1,j)
   enddo
enddo
```

Corrected: (j's can be done in any order, i's cannot)

```
!$omp parallel do private(i)
do j=1,m
   do i=2,n
      a(i,j) = a(i-1,j)
   enddo
enddo
```

Incorrect code for computing $\|x\|_1 = \sum_i |x_i|$:

```
norm = 0.d0
!$omp parallel do
do i=1,n
    norm = norm + abs(x(i))
    enddo
```

There is a race condition: each thread is updating same shared variable `norm`.

Correct code:

```
!$omp parallel do reduction(+ : norm)
do i=1,n
    norm = norm + abs(x(i))
    enddo
```

A reduction reduces an array of numbers to a single value.