Introduction to High Performance Scientific Computing

Autumn, 2018

Lecture 12

Imperial College
London

Prasun Ray

13 November 2018

Announcements

- Tutorials from this week on will be in Huxley 410
  - Except last Tuesday of last week of term
  - Will give quick demo using the new VMs at beginning of this week's lab
  - See class webpage for instructions on launching these VMs

- Tuesday office hours also in Huxley 410 (except last week of term)

- A correction was added to the homework assignment on Sunday (related to definition of testing error)

- Homework 1 marking is in-progress, hope to provide scores+feedback by Thursday afternoon

- If you haven't done so already, please fill out the short course feedback form here: https://goo.gl/forms/q0Vq81pu1tbZCNlf1

Comments

- **Fortran and f2py tips/notes**
  - **If you're running a Fortran routine from** qtconsole**, the output from the Fortran code will print to the *Unix* terminal where qtconsole is launched**
    - **If using** ipython **(or** ipython3**) terminal, the Fortran output will be displayed within the** ipython **terminal**

  - **A segmentation fault is usually associated with arrays and array sizes, *e.g.* trying to access the 11th element in a 10-element array**
    - **Compiling with the** -fcheck=bounds **flag may give a more helpful error message:**
      - gfortran –fcheck=bounds –o test.exe test.f90
      - f2py --opt='-fcheck=bounds' –c test.f90 –m t1

- **Be careful copying and pasting code from slides! Sometimes powerpoint changes text in strange ways!**

Today

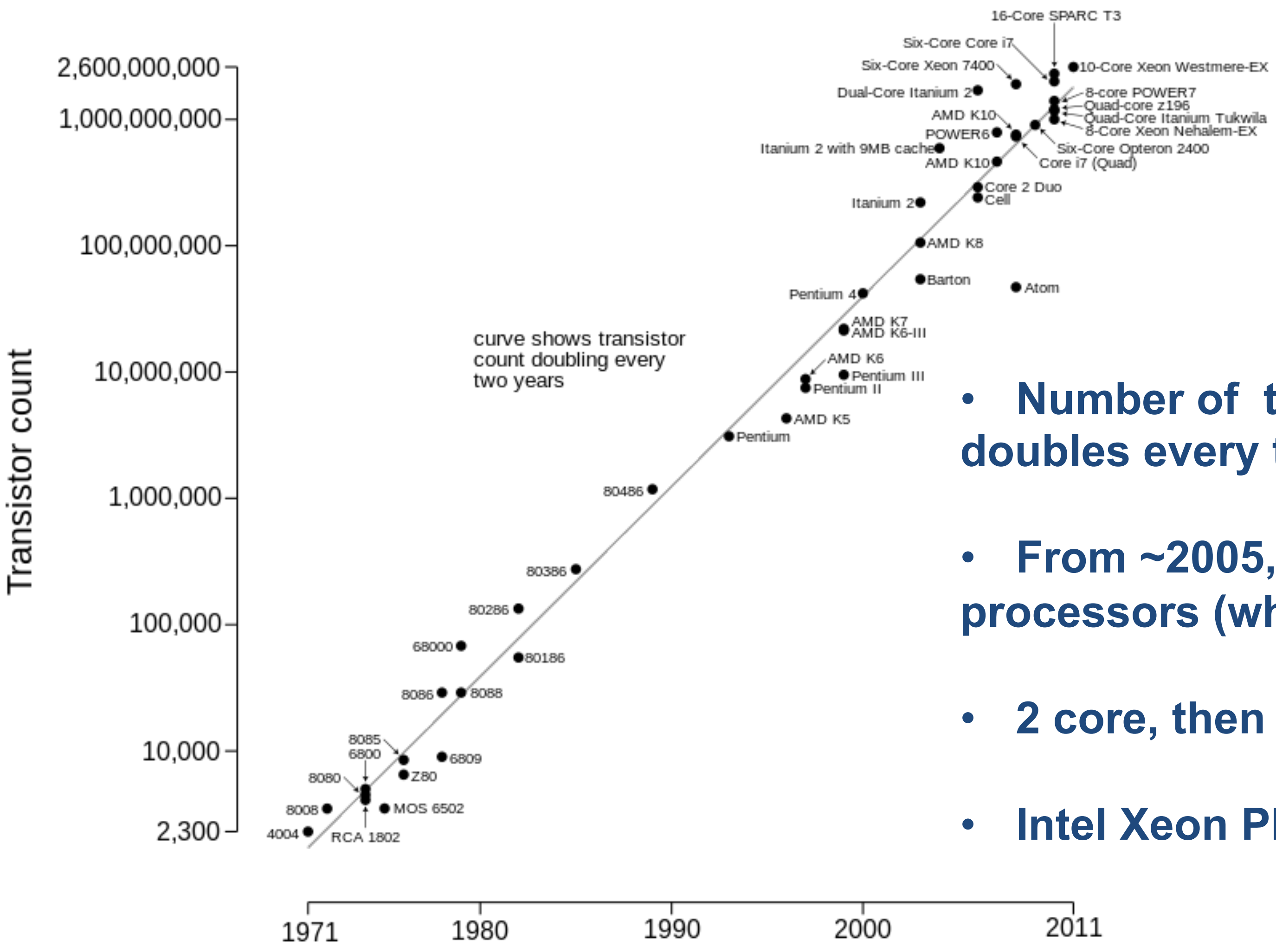
Introduction to parallel computing (carried over from lecture 11)

Introduction to OpenMP

- *Getting started*
- *Parallel regions*
- *Parallel loops*

Moore's law

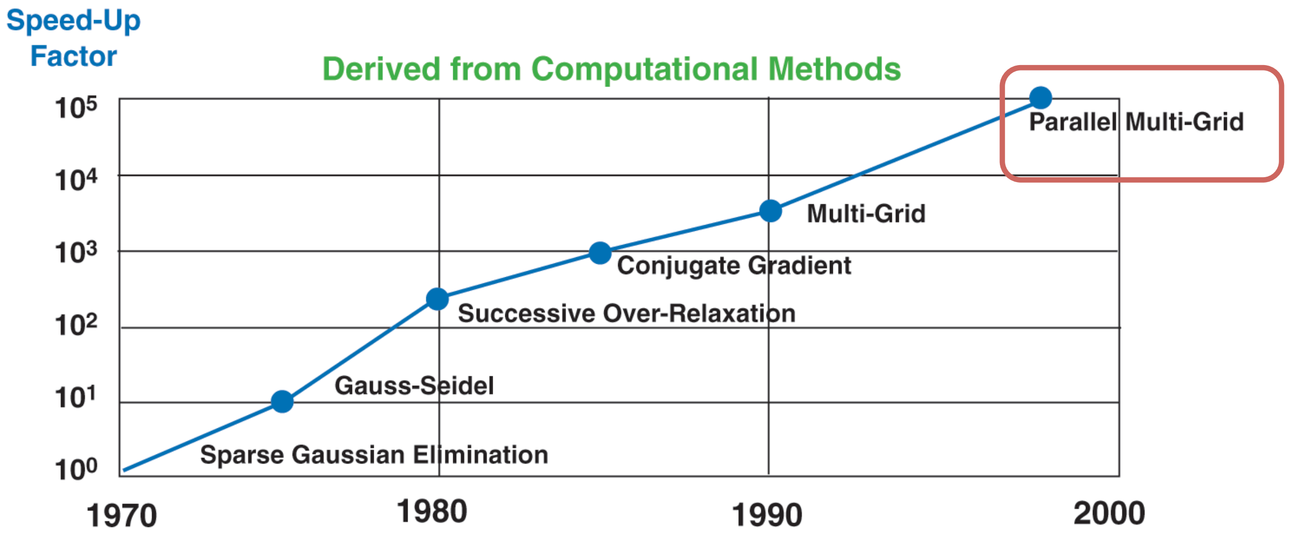
Microprocessor Transistor Counts 1971-2011 & Moore's Law

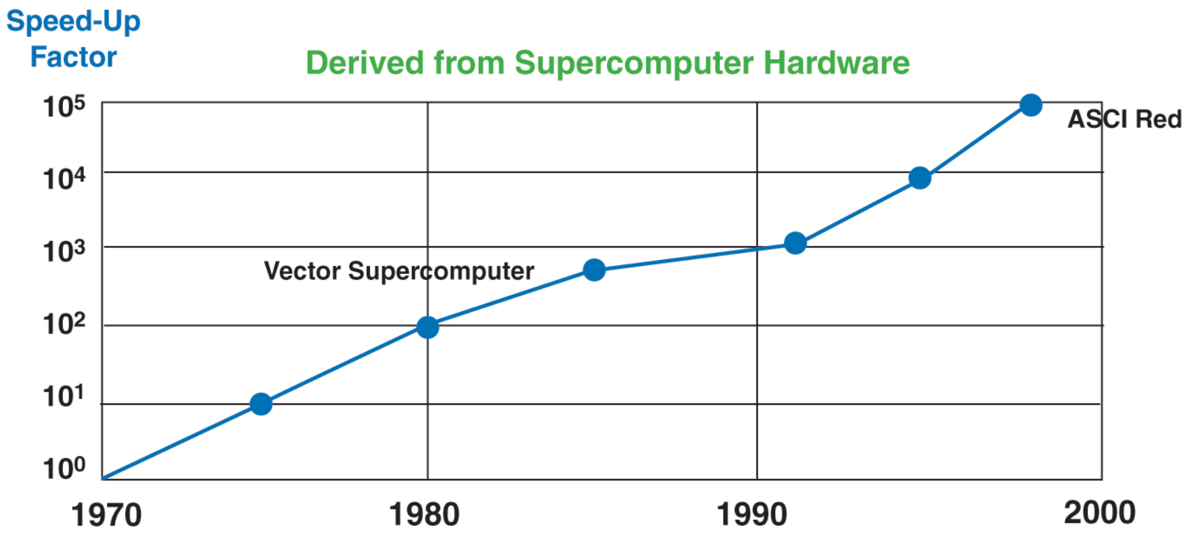


- Number of transistors on chip doubles every two years
- From ~2005, multicore processors (why?)
- 2 core, then 4, 6, 8, 16
- Intel Xeon Phi: 60+ cores!

Imperial College London

Algorithms and hardware





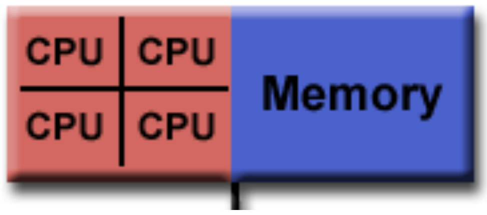
Why parallelize a code?

1. Serial (single-processor) code is too slow

or

2 Serial code is too big

Parallel computing paradigms

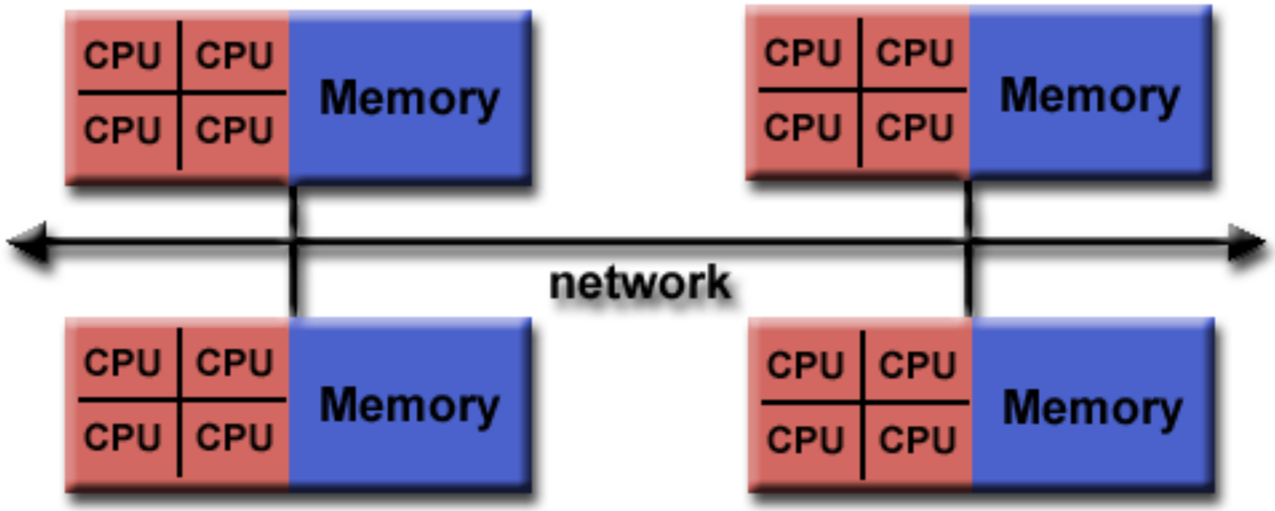


Shared memory

- One 4-core chip with shared memory (RAM)
- MPI can coordinate communication between cores
- OpenMP generally easier to use for shared-memory systems

- MPI = *Message Passing Interface*
- OpenMP = *Open Multi-Processing*

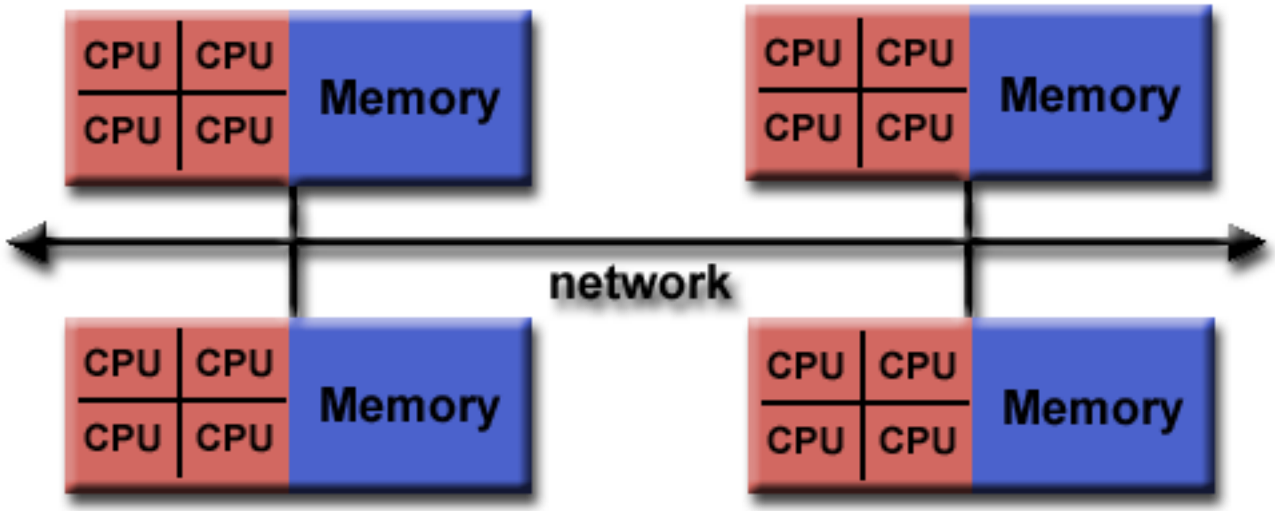
Parallel computing paradigms



Distributed memory

- Each (4-core) chip has its own memory
- The chips are connected by network ‘cables’
- MPI coordinates communication between two or more CPUs

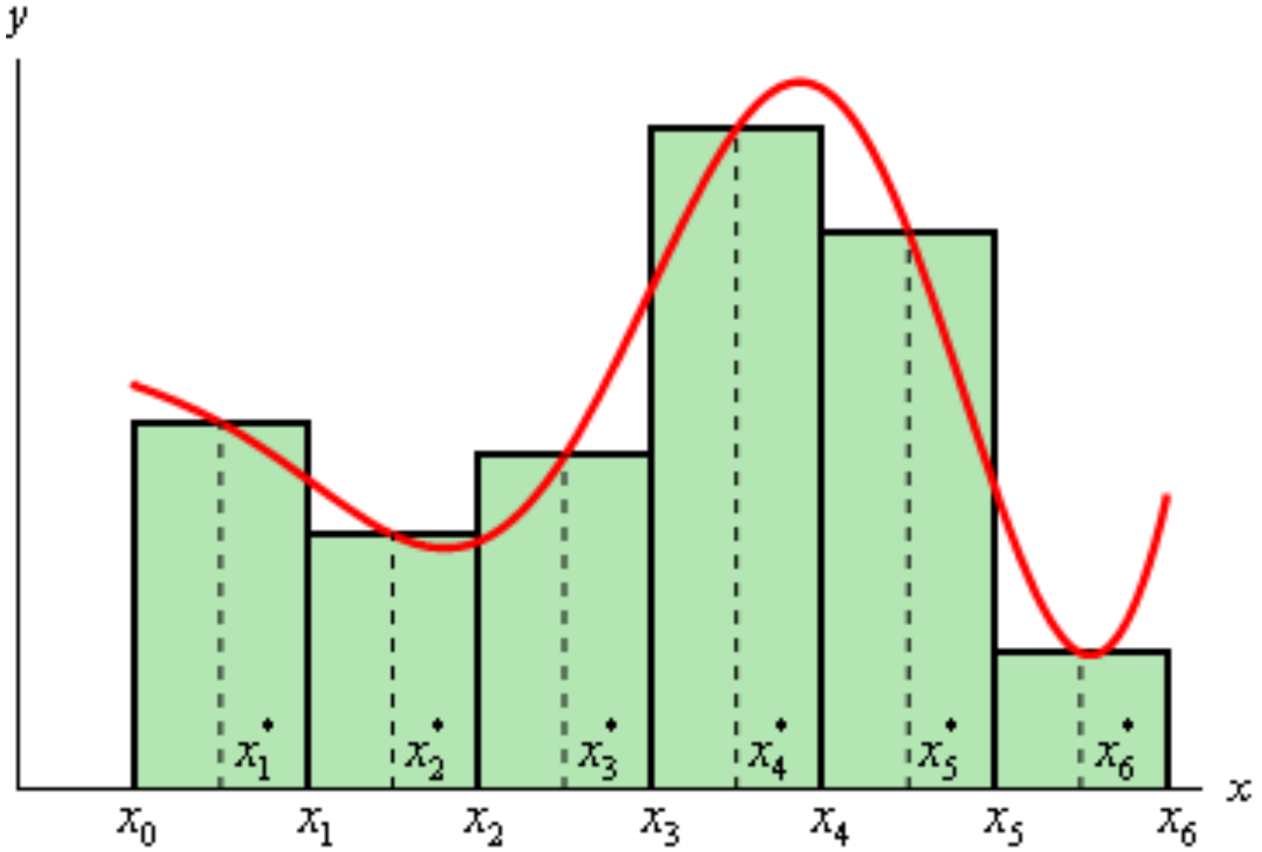
Parallel computing paradigms



Related approaches:

- Hybrid programming: mix of shared-memory (OpenMP) and distributed-memory (MPI) programming
- GPU’s: Shared memory programming (CUDA or OpenCL)
- Coprocessors and co-array programming

Example: computing an integral



- **Estimate integral with midpoint rule,**

$$I = \int_{x_0}^{x_6} f(x)dx$$

1. **Compute:**

$$f(x_1^*), f(x_2^*), \ldots$$

2. **Compute areas of rectangles:**

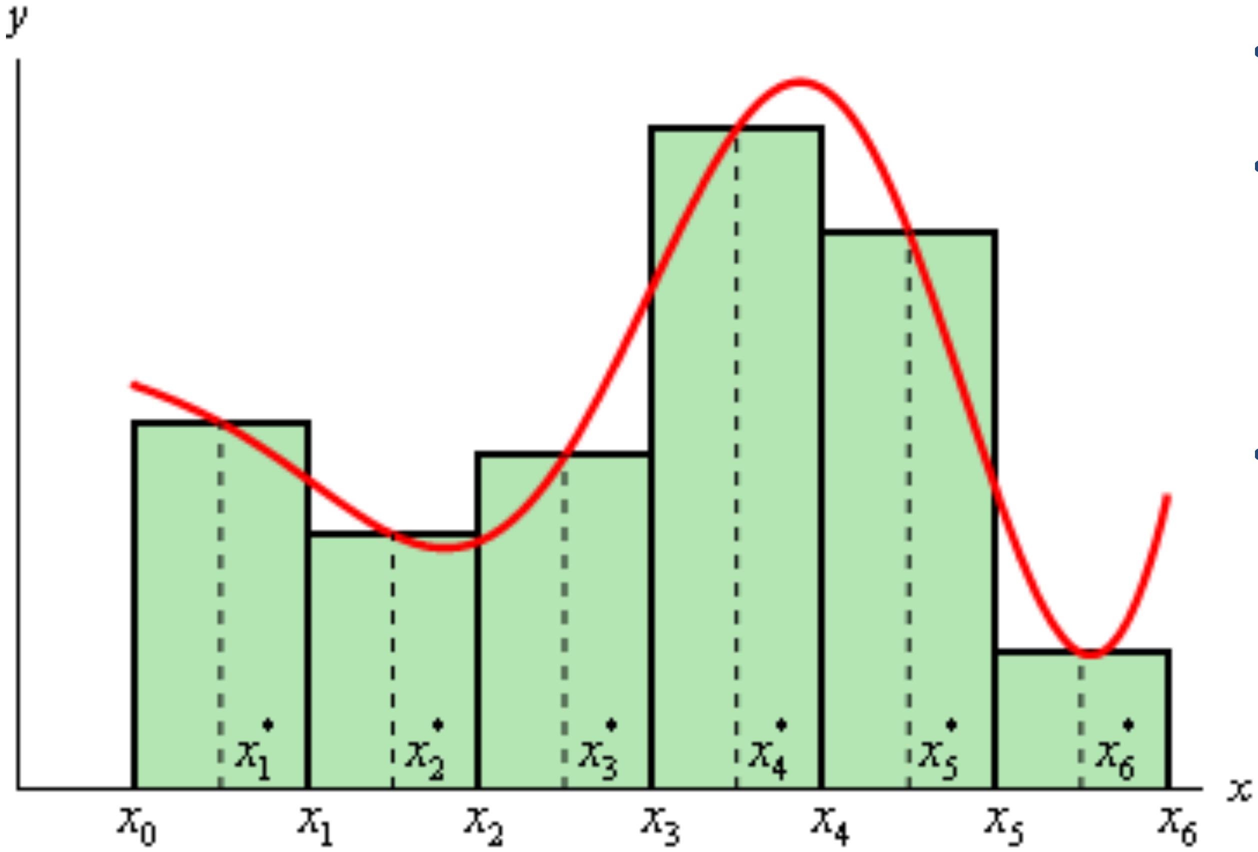
$$I_1 = (x_1 - x_0) * f(x_1^*)$$

3. **Sum areas:**

$$I \approx I_1 + I_2 + I_3 + \ldots$$

Imperial College
London

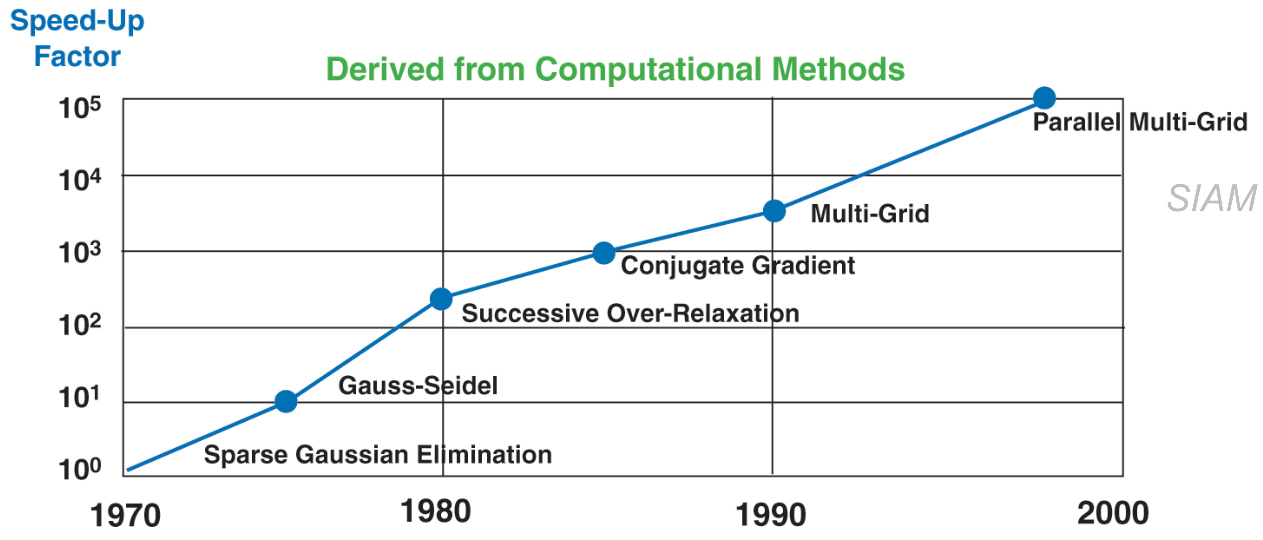
Example: computing an integral



- **How to parallelize?**
- **With three processors, can compute areas of two rectangles on each processor**
- **Not practical for small calculations, but could split 1e7 rectangles across, say, 10 processors**

Imperial College London

Scaling and performance

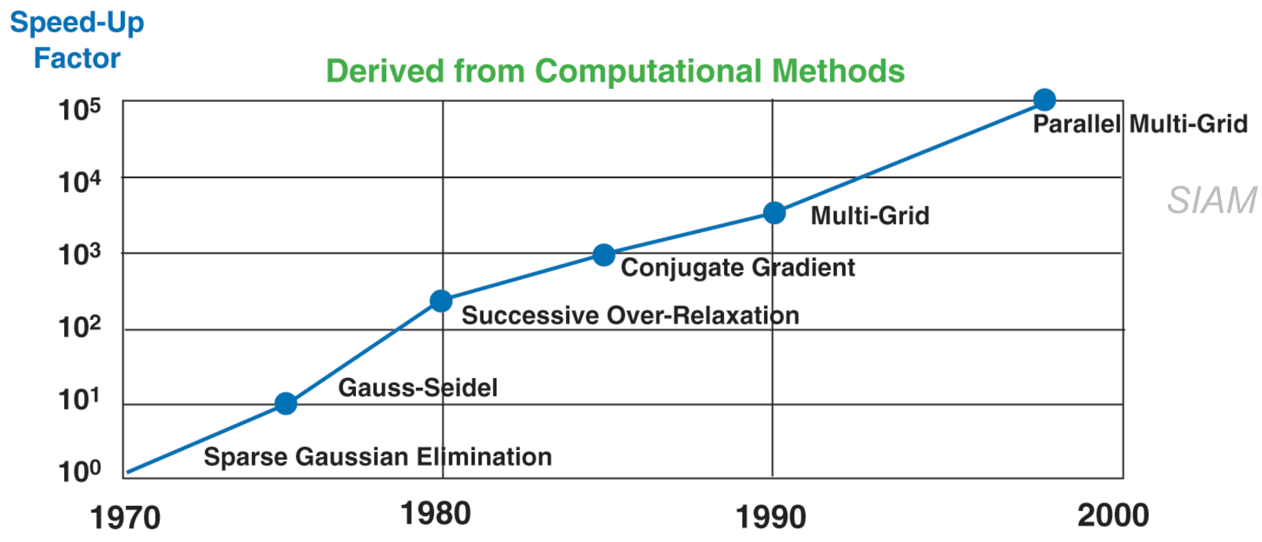


SIAM Rev (2001)

- **How do we measure performance of a parallel code?**
- **Serial code: Optimize the *efficiency* → cost required to obtain a certain level of accuracy**

Imperial College London

Scaling and performance



SIAM Rev (2001)

- **How do we measure performance of a parallel code?**
- **Serial code: Optimize the *efficiency* → cost required to obtain a certain level of accuracy**
- **Parallel code: Also optimize *scaling* or *speedup*: how much faster is the calculation when the number of procs is increased?**

Imperial College London

Speedup

- Speedup = Computation time on one proc/time on N procs = Ts/Tp
- Ideal: N = 10 processors, speedup = N = 10

Speedup

- Speedup = Computation time on one proc/time on N procs = Ts/Tp
- Ideal: N = 10 processors, speedup = N = 10
- Real life: Speedup will be less than N (possibly much less) Why?
  - Startup costs
  - Communication
  - Only part of the algorithm parallelizes
- Typically interested in performance of large problems running on large number of processors
  - Workstation: N= 16, 32
  - Imperial HPC (cx2): N = 256+
  - UK HPC (Archer): N = 1e3, 1e4, …
- Ahmdal’s law provides guidance

Ahmdal's law

- Usually only part of a computation can be parallelized
  - One processor: $T(1) = s + p$
  - Two processors: $T(2) = s + p/2$
  - N processors: $T(N) = s + p/N$

    p is the part of the code that can be parallelized

Ahmdal's law

- Usually only part of a computation can be parallelized
  - One processor: T(1) = s + p
  - Two processors: T(2) = s + p/2
  - N processors: T(N) = s + p/N

    p is the part of the code that can be parallelized

So, if only half the code can be parallelized (s = p = 0.5), Then the maximum speedup T(1)/T(N → inf) = (s+p)/(s) = 2

It is important for p to be much larger than s!

Ahmdal’s law

Speedup T(1)/T(N) = (s+p)/(s+p/N)

Example: s = 0.1, p = 0.9

Number of processors	Speedup
1	1
2	1.8
4	3.1
8	4.7
16	6.4
32	7.8
256	9.7

Ahmdal's law

Speedup $T(1)/T(N) = (s+p)/(s+p/N)$

Example: $s = 0.1$, $p = 0.9$

Number of processors	Speedup
1	1
2	1.8
4	3.1
8	4.7
16	6.4
32	7.8
256	9.7

Waste of resources to use N=256!

Strong and weak scaling

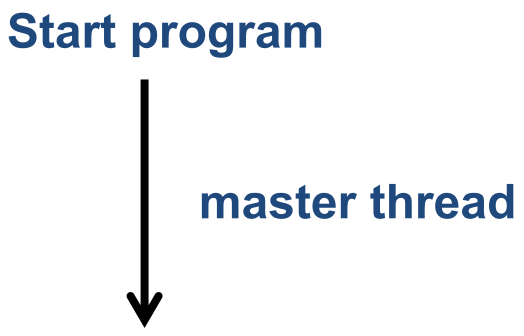
- Strong scaling: Time needed to solve a problem of fixed size as number of processors increases
- Weak scaling: Time needed for problem with *fixed size per processor*

Overview

- OpenMP provides a fairly easy approach to parallelizing c/c++ or fortran code
- Add *directives* indicating how/where the code should run in parallel
- Parallel regions have multiple threads, each of which should be assigned computational tasks
  - OpenMP is for *shared-memory* parallel programming
  - Each thread has access to all variables that existed before parallel region was created
  - This can cause problems if multiple threads try to change the same variable!
- Particularly useful for parallelizing loops
- When compiling, add *–fopenmp* flag

Overview

Program starts with single *master thread*

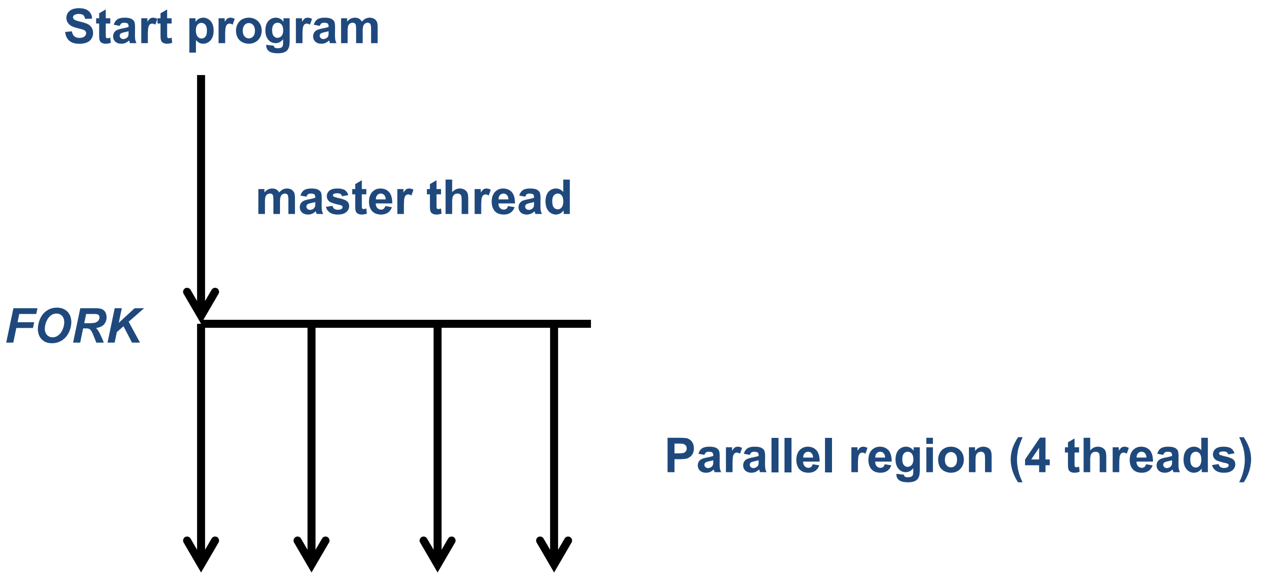


Overview

Program starts with single *master thread*

Then, launch parallel region with multiple threads.

Each thread has access to all variables introduced previously



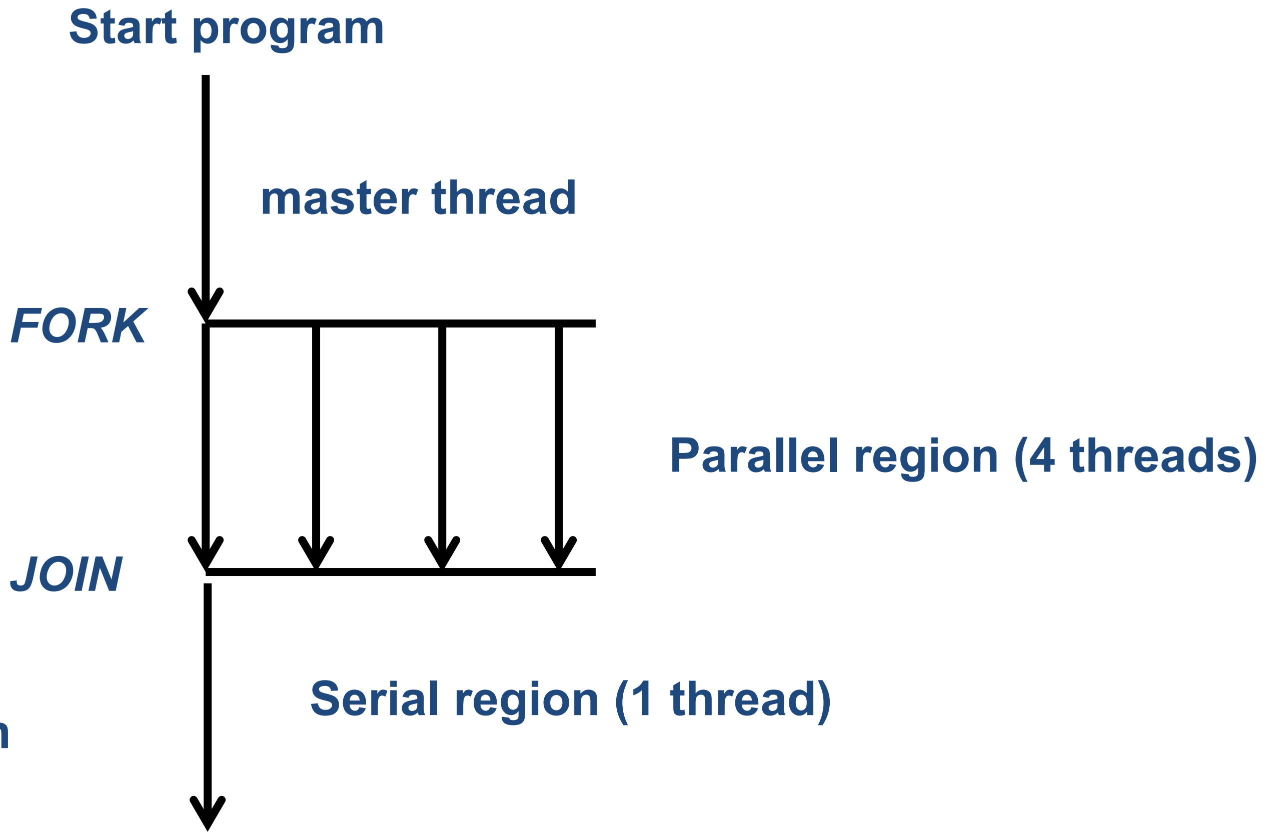
Overview

Program starts with single *master thread*

Then, launch parallel region with multiple threads.

Each thread has access to all variables introduced previously

Can end parallel region if/when desired and launch parallel regions again in future as needed



Simple OpenMP example

- **Launch parallel region, get info on threads (see** *firstomp_v0.f90)*
- **Must use openMP module,** *omp_lib*
    - **This makes functions like** *omp_get_num_threads* **available**

```
!Getting started with OpenMP

program firstomp
    use omp_lib !makes OpenMP routines, variables available
    implicit none
    integer :: NumThreads,threadID
```

Imperial College
London

Simple OpenMP example

- **Launch parallel region, get info on threads (see** *firstomp_v0.f90)*
- **Must use openMP module,** *omp_lib*
    - **This makes functions like** *omp_get_num_threads* **available**

```
!Getting started with OpenMP

program firstomp
   use omp_lib !makes OpenMP routines, variables available
   implicit none
   integer :: NumThreads,threadID
!$OMP PARALLEL
   NumThreads = omp_get_num_threads()
   threadID = omp_get_thread_num()
   print *, 'this is thread',threadID, ' of ', NumThreads
!$OMP END PARALLEL
```

- !$OMP **starts an OpenMP directive (#pragma omp in** ***c*****)**
- !$OMP PARALLEL **starts a parallel region (forks a number of threads)**
    - omp_get_num_threads **tells us how many threads are forked**
    - omp_get_thread_num **tells us which thread is being used**

Imperial College
London

Simple OpenMP example

- **Let's compile and run this:**

```
$ gfortran -fopenmp -o testv0.exe firstomp_v0.f90
$ ./testv0.exe
 this is thread           1  of           4
 this is thread           1  of           4
 this is thread           1  of           4
 this is thread           1  of           4
```

- **Total number of threads is correct, but problem getting the thread id.**

Simple OpenMP example

- **Let’s compile and run this:**

```
$ gfortran -fopenmp -o testv0.exe firstomp_v0.f90
$ ./testv0.exe
 this is thread           1  of           4
 this is thread           1  of           4
 this is thread           1  of           4
 this is thread           1  of           4
```

- **Total number of threads is correct, but problem getting the thread id.**

- **Remember:** *threadID* **is a *shared* variable.**
  - **Each thread is writing to the same variable with it’s id, so only the “last” thread has its ID displayed**

Simple OpenMP example

- **Let's compile and run this:**

```
$ gfortran -fopenmp -o testv0.exe firstomp_v0.f90
$ ./testv0.exe
 this is thread           1  of            4
 this is thread           1  of            4
 this is thread           1  of            4
 this is thread           1  of            4
```

- **Total number of threads is correct, but problem getting the thread id.**
- **Remember:** *threadID* **is a *shared* variable.**
  - **Each thread is writing to the same variable with it's id, so only the "last" thread has its ID displayed**
- **How can we fix this? First approach: define a *critical* region…**

Simple OpenMP example

- A critical region, defined with !$OMP CRITICAL, runs in serial
  - The threads carry out their tasks sequentially (*firstomp_v1.f90*)

```
!$OMP PARALLEL
    NumThreads = omp_get_num_threads()
     !$OMP CRITICAL
         threadID = omp_get_thread_num()
         print *, 'this is thread',threadID, ' of ', NumThreads
     !$OMP END CRITICAL
!$OMP END PARALLEL
```

Simple OpenMP example

- **A critical region, defined with !$OMP CRITICAL, runs in serial**
  - **The threads carry out their tasks sequentially (*firstomp_v1.f90*)**

```
!$OMP PARALLEL
   NumThreads = omp_get_num_threads()
    !$OMP CRITICAL
       threadID = omp_get_thread_num()
       print *, 'this is thread',threadID, ' of ', NumThreads
    !$OMP END CRITICAL
!$OMP END PARALLEL
```

So now, if we compile and run:

```
$ ./testv1.exe
 this is thread           0  of           4
 this is thread           2  of           4
 this is thread           1  of           4
 this is thread           3  of           4
```

Imperial College
London

Simple OpenMP example

- **Critical regions useful for: data I/O, displaying results**
- **But forcing serial execution in a parallel region, not generally desirable**
- **Better approach: set** threadID **to be a *private* variable (*firstomp.f90)***
  - **Then, each thread will have their own private copy of the variable**

Simple OpenMP example

- **Critical regions useful for: data I/O, displaying results**
- **But forcing serial execution in a parallel region, not generally desirable**
- **Better approach: set** threadID **to be a *private* variable (*firstomp.f90)***
  - **Then, each thread will have their own private copy of the variable**

```
!$OMP PARALLEL PRIVATE(threadID)
   NumThreads = omp_get_num_threads()
   threadID = omp_get_thread_num()
   print *, 'this is thread',threadID, ' of ', NumThreads
!$OMP END PARALLEL
```

```
$ ./test.exe
 this is thread           0  of            4
 this is thread           2  of            4
 this is thread           1  of            4
 this is thread           3  of            4
```

Simple parallel calculation

- **Can use *threadID* to assign tasks to threads:**

```
!$OMP PARALLEL PRIVATE(threadID)
   NumThreads = omp_get_num_threads()
   threadID = omp_get_thread_num()

    if (threadID==0) then
        call subroutine1(in1,out1)
    elseif (threadID==1) then
        call subroutine1(in2,out2)
    end if

!$OMP END PARALLEL
```

- **Important to distribute work evenly across threads (load balancing)**

Overview

- **OpenMP (primarily) consists of *directives* and *routines***
- **Directives are denoted with** !$OMP
  - !$OMP parallel**,** !$OMP critical**, …**
  - **Directives are recognized when** –fopenmp **compile-flag is used**
  - **Otherwise, they are interpreted as comments**
    - **What happens if you use:**

      !$ print *, “compiled with –fopenmp”
- **Routines are available via the** use omp_lib **command**
  - **e.g.** omp_get_thread_num **and** omp_get_num_threads

Parallel loops

- Loops form the backbone of most scientific codes
- They should be parallelized whenever possible
- They can be parallelized if the calculations of each iteration are independent of each other (no data dependencies)

```
do i1 = 1,n
    x(i1) = y(i1) + z(i1)
end do
```

Ok to parallelize

```
do i1 = 1,n
    norm = norm + abs(x(i1))
end do
```

Can’t parallelize easily: each thread updating, *norm*

Parallel loops

- OpenMP makes it very easy to parallelize loops

```
!$OMP parallel do
do i1 = 1,n
    x(i1) = y(i1) + z(i1)
end do
!$OMP end parallel do
```

- OpenMP automatically distributes iterations across threads
  - If NumThreads=2 and n=10, iterations 1,…,5 would be given to thread 0 and iterations 6,…,10 would be done by thread 1 (or vice versa)
  - The iterated variable, *i1*, is automatically set to *private*. Each thread has its own copy.

Parallel loops

- **Simple example (*loop_omp1.f90*):**

```
!$OMP parallel do private(threadID)
do i1 = 1,size(x)
    x(i1) = y(i1) + z(i1)
    threadID = omp_get_thread_num()
    print *, 'iteration ',i1,' assigned to thread ',threadID
end do
!$OMP end parallel do

print *, 'test:', maxval(abs(x-y-z))
```

- **Note: *threadID* again set to *private***

- **Compile and run…**

Parallel loops

- **Simple example (*loop_omp1.f90*):**

```
!$OMP parallel do private(threadID)
do i1 = 1,size(x)
    x(i1) = y(i1) + z(i1)
    threadID = omp_get_thread_num()
    print *, ’iteration ',i1,' assigned to thread ',threadID
end do
!$OMP end parallel do

print *, 'test:', maxval(abs(x-y-z))
```

```
$ gfortran -fopenmp -o testl1.exe loop_omp1.f90
$ ./testl1.exe
 interation            1  assigned to thread           0
 interation            3  assigned to thread           2
 interation            2  assigned to thread           1
 interation            4  assigned to thread           3
 test:   2.2204460492503131E-016
```