Introduction to High Performance Scientific Computing

Autumn, 2016

Lecture 11

Imperial College
London

Prasun Ray

14 November 2016

Today

Further comments on f2py and timing code

Compiling and profiling fortran code

Introduction to parallel computing

Fortran timing functions

See midpoint_time.f90: timers placed around *do-loop* which computes areas of rectangles

Can use timing results to check of code is behaving "properly."

```
!timing variables
    real(kind=8) :: cpu_t1,cpu_t2,clock_time
    integer(kind=8) :: clock_t1,clock_t2,clock_rate


 call system_clock(clock_t1)
 call cpu_time(cpu_t1)
    !loop over intervals computing each interval's contribution to
integral
... Midpoint quadrature ...
 call cpu_time(cpu_t2)
 print *, 'elapsed cpu time (seconds) =',cpu_t2-cpu_t1

 call system_clock(clock_t2,clock_rate)
 print *, 'elapsed wall time (seconds)= ',
                          dble(clock_t2-clock_t1)/dble(clock_rate)
```

Fortran timing functions

- **Run midpoint_time with N=20000 and N=40000**

```
$ ./midpoint_time.exe
 elapsed cpu time (seconds) =   4.440000000000038E-004
 elapsed wall time (seconds)=    5.30000019E-04
 N=       20000
```

```
$ ./midpoint_time.exe
 elapsed cpu time (seconds) =   8.839999999999980E-004
 elapsed wall time (seconds)=    9.67999978E-04
 N=       40000
```

- **We can see the cpu time doubles (as we would hope). The wall-time shows more complicated behavior – could be related to what other background processes are running, or just the very short computation time.**

Profiling

- **Profilers give detailed information about time spent in different parts of code**
- **In python:** *run –p filename* **gives profiling info**
- **With fortran (or c), can use *gprof* utility (not available on Macs)**
- **Steps:**
  1. **Compile code with** *–pg* **flag**

```
$ gfortran –pg –o mt2.exe midpoint_time2.f90
```

Profiling

- **Profilers give detailed information about time spent in different parts of code**
- **In python:** *run –p filename* **gives profiling info**
- **With fortran (or c), can use *gprof* utility (not available on Macs)**
- **Steps:**
  1. **Compile code with** *–pg* **flag**

```
$ gfortran –pg –o mt2.exe midpoint_time2.f90
```

  2. **Run code (this will generate *gmon.out)*:**

```
$ ./mt2.exe
```

  3. **Finally, run gprof**

```
$ gprof ./mt2.exe
```

Profiling

Output looks like:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 63.67      6.25     6.25        1     6.25     9.85  MAIN__
 36.75      9.85     3.61 512000000     0.00     0.00  integrand_
```

and:

```
index % time    self  children    called     name
                6.25    3.61       1/1           main [2]
[1]    100.0    6.25    3.61       1         MAIN__ [1]
                3.61    0.00 512000000/512000000     integrand_ [3]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    9.85                  main [2]
                6.25    3.61       1/1           MAIN__ [1]
-----------------------------------------------
                3.61    0.00 512000000/512000000     MAIN__ [1]
[3]     36.6    3.61    0.00 512000000         integrand_ [3]
-----------------------------------------------
```

Profiling

- Can get line-by-line information from other tools like, *oprof*
- The more complicated the code, the more useful profiling becomes

Notes on compiling

- **Up to now:**

```
$ gfortran –c program.f90
$ gfortran –o program.exe program.o –llapack
```

or

```
$ gfortran –o program.exe program.f90 –llapack
```

- **But typically want to turn on *optimization* -O flag:**

```
$ gfortran –O3 –c program.f90
```

- **-O3 is highest level of optimization (can also use -O1, -O2)**

Notes on compiling

- Look at midpoint_time.f90 compiled with and without -O3:

```
$ ./mt2.exe
 elapsed cpu time (seconds) =  0.3251049999999998
 elapsed wall time (seconds)=   0.326231003
 N=        25600
 sum=   3.1415926537169634
 error=   1.2717027431108363E-010

$ ./mt2_O3.exe
 elapsed cpu time (seconds) =  0.1437609999999997
 elapsed wall time (seconds)=   0.144617006
 N=        25600
 sum=   3.1415926537169634
 error=   1.2717027431108363E-010
```

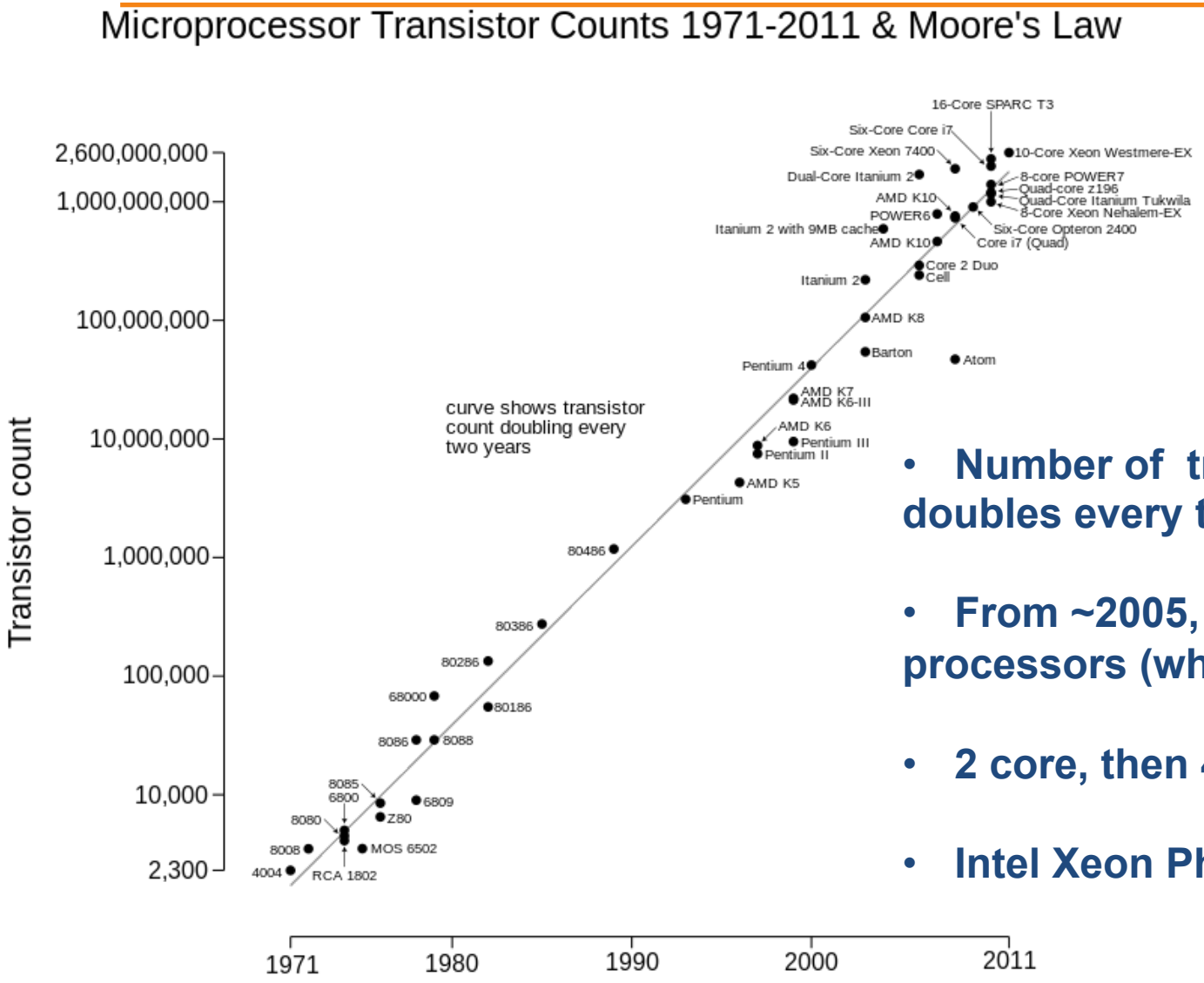
- Optimization can make a substantial difference
- f2py uses -O3 by default

Notes on compiling

- **Many other useful compiler flags, e.g.:**
  - -Wall: **“warn about all” – generates warnings about common sources of bugs**
  - -fbounds-check: **checks that array index is within bounds of an array (common problem)**

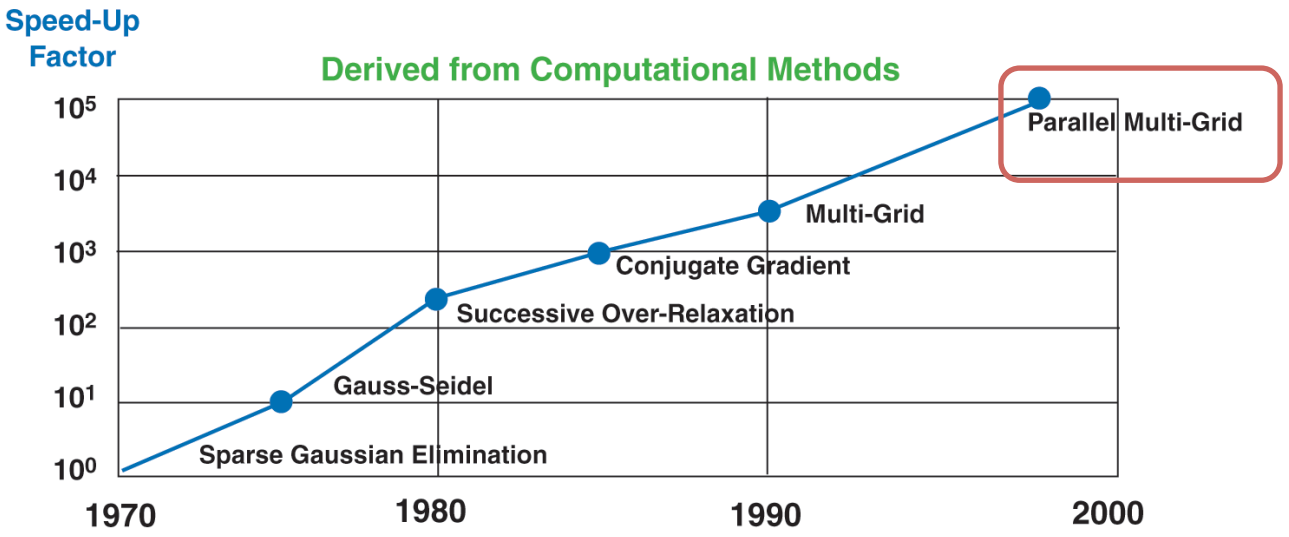
- **Comprehensive list at: https://gcc.gnu.org/onlinedocs/gfortran/Invoking-GNU-Fortran.html**

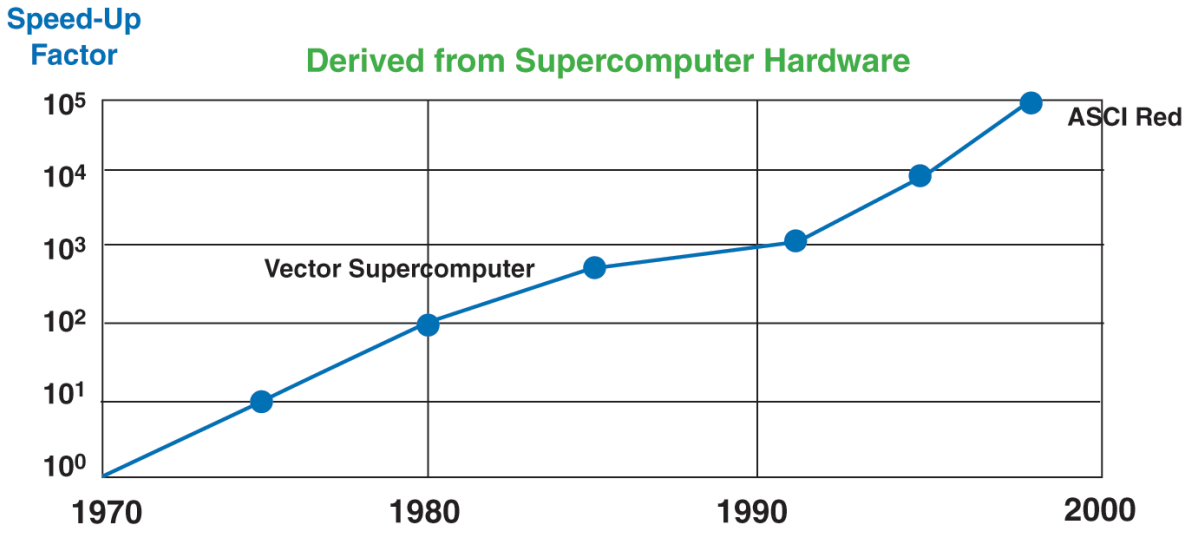
Moore's law



- **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!**

Algorithms and hardware





SIAM Rev (2001)

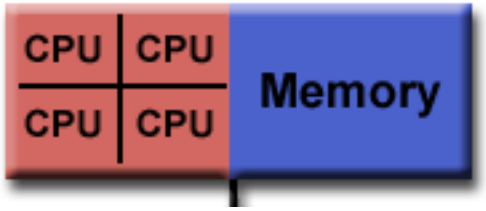
Why parallelize a code?

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

   or

2 Serial code is too big

Parallel computing paradigms

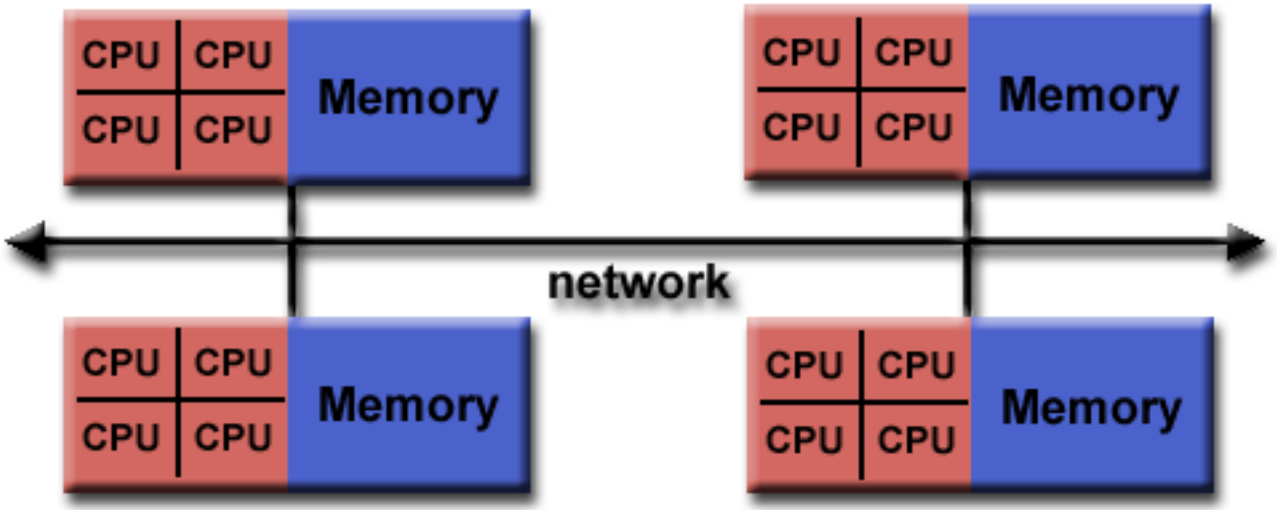


Shared memory

- 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*

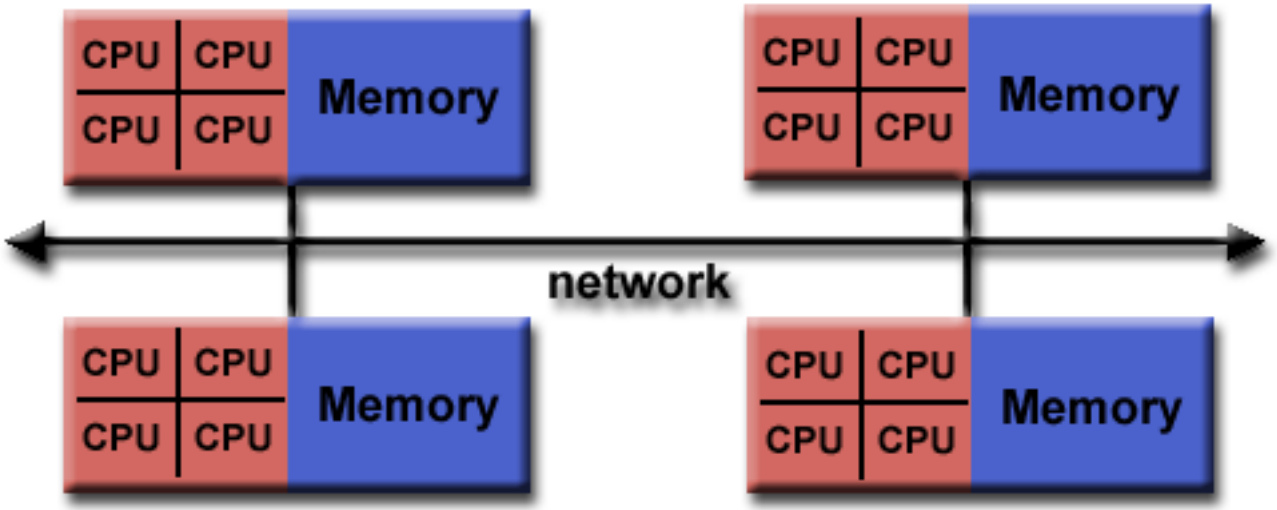
Parallel computing paradigms



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

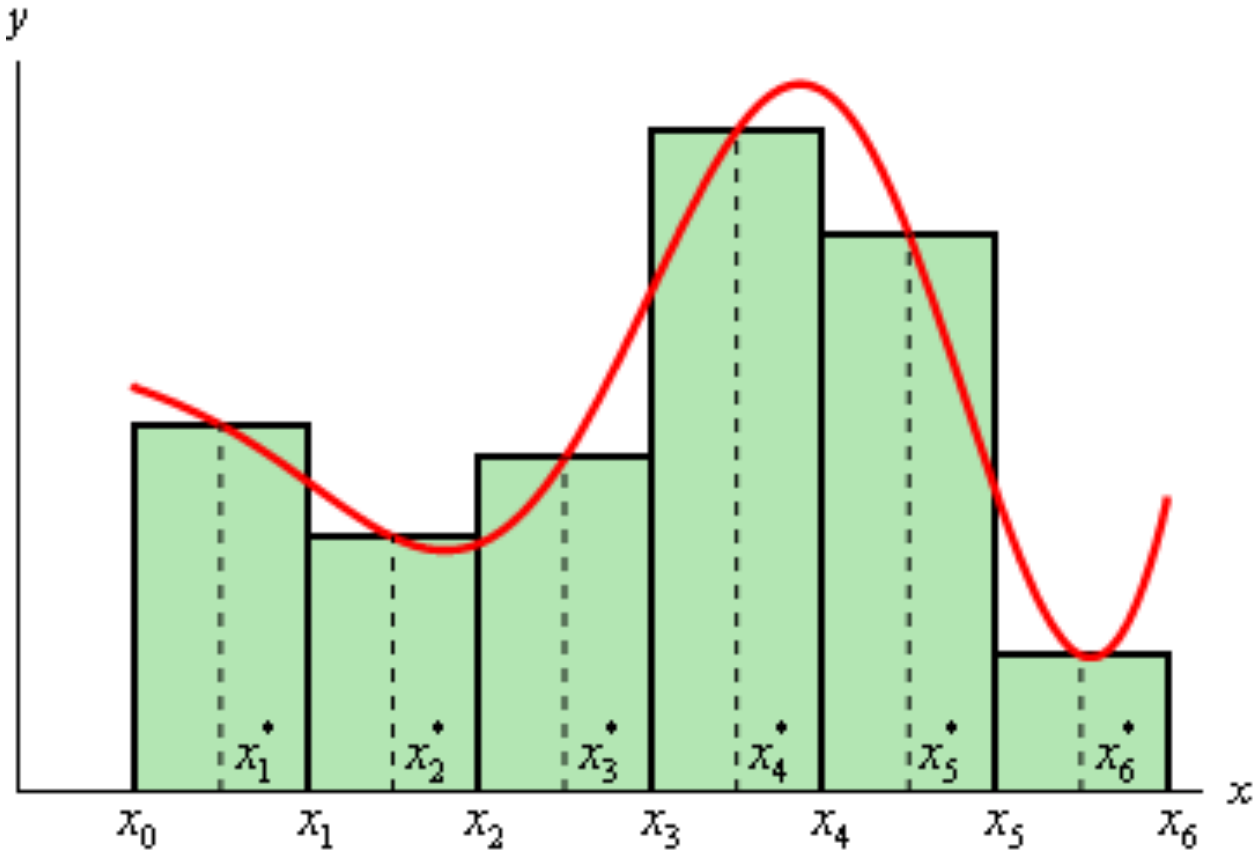
Parallel computing paradigms



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:**

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

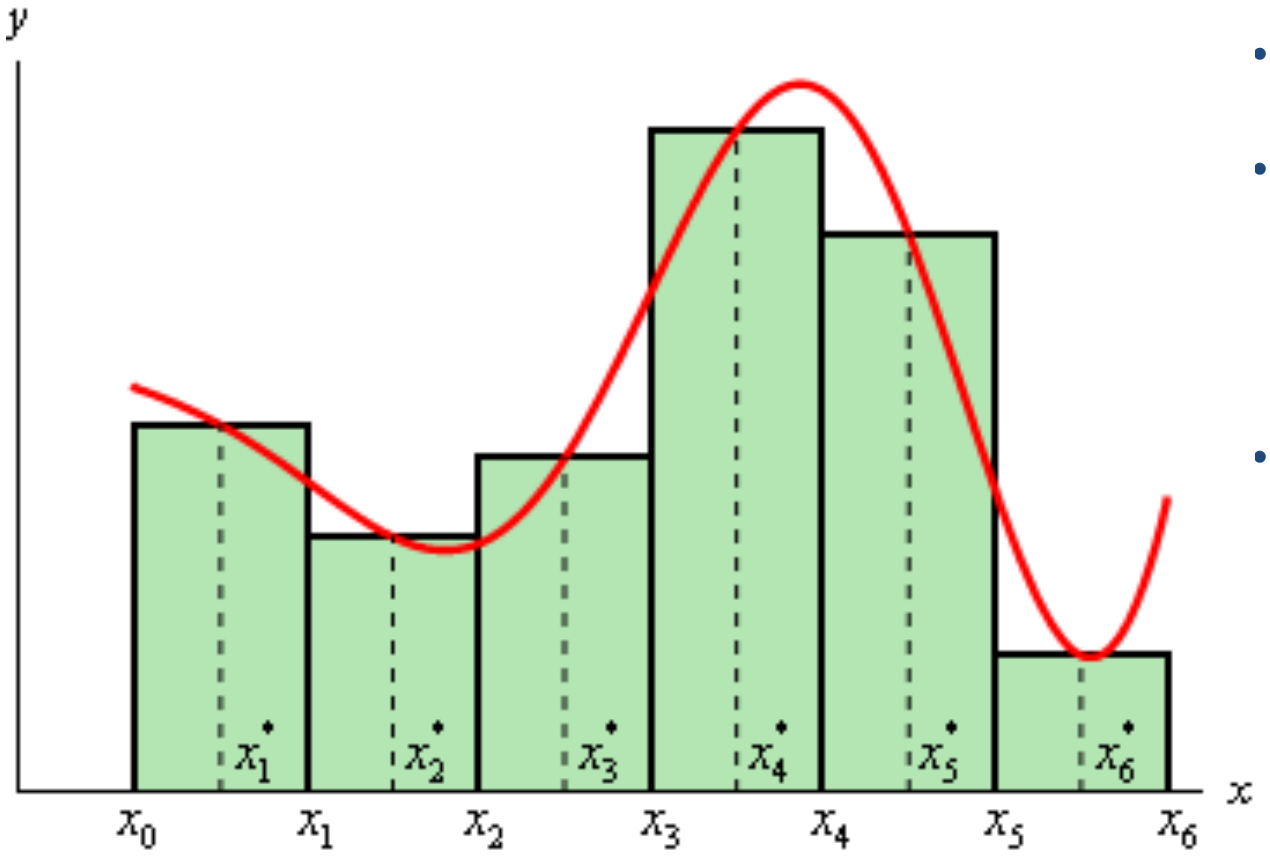
2. **Compute areas of rectangles:**

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

3. **Sum areas:**

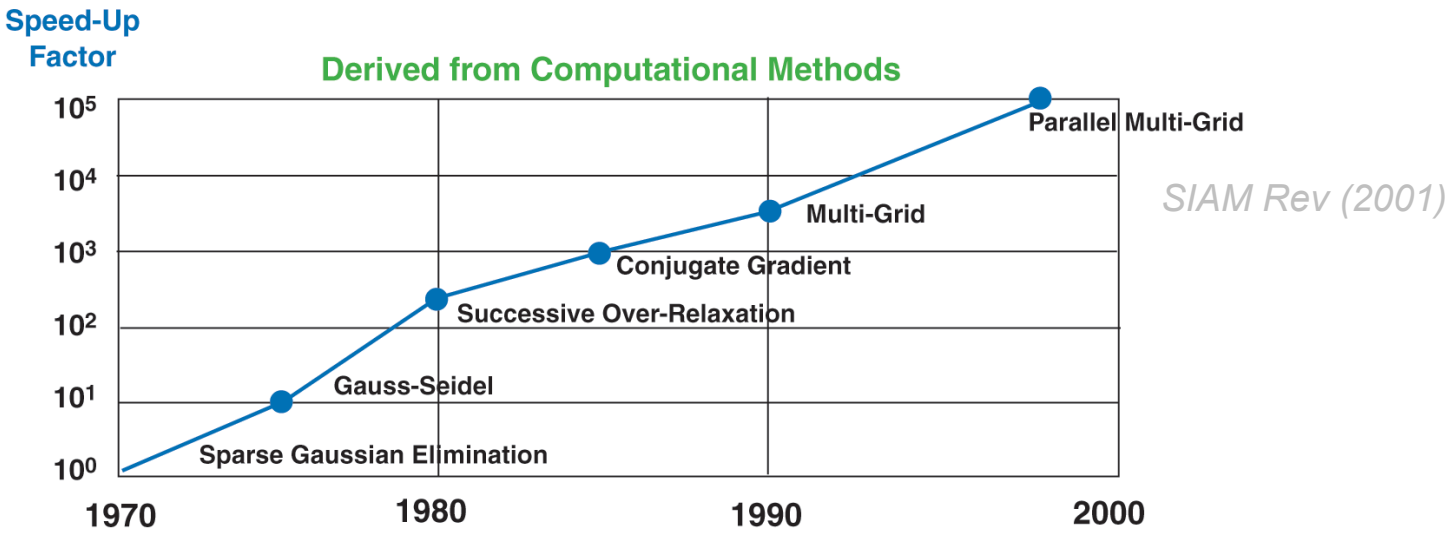
$$I \approx I_1 + I_2 + I_3 + \ldots$$

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

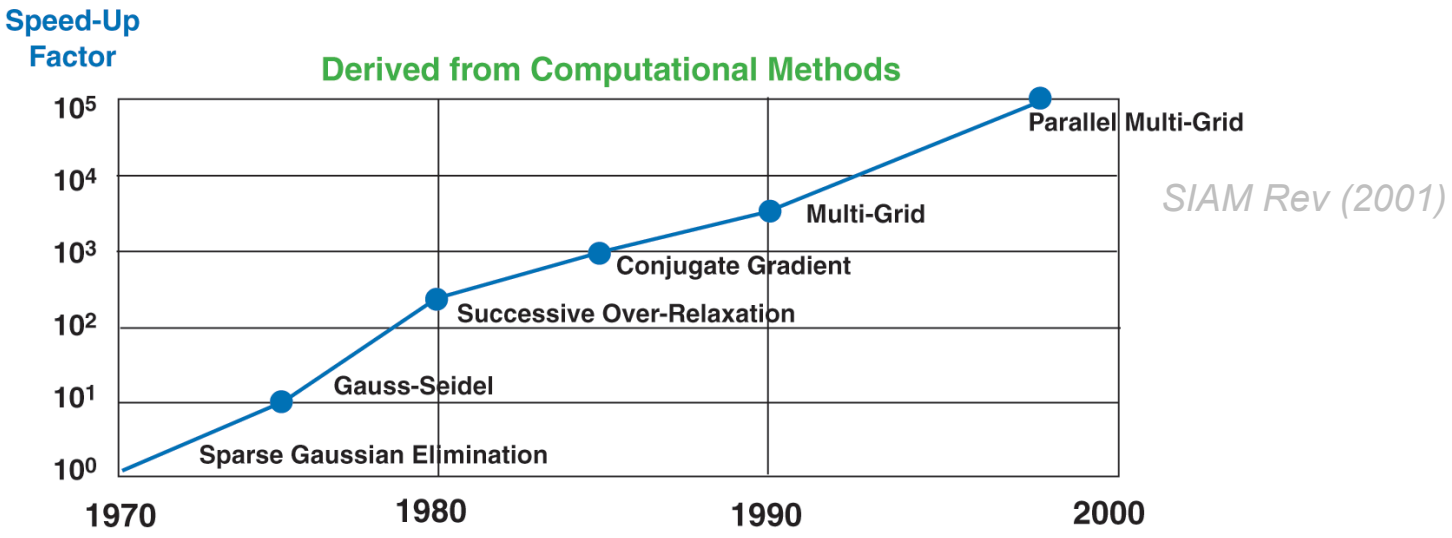
Scaling and performance



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

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?

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***