Introduction to High Performance Scientific Computing

Autumn, 2018

Lecture 11

Imperial College
London

Prasun Ray

8 November 2018

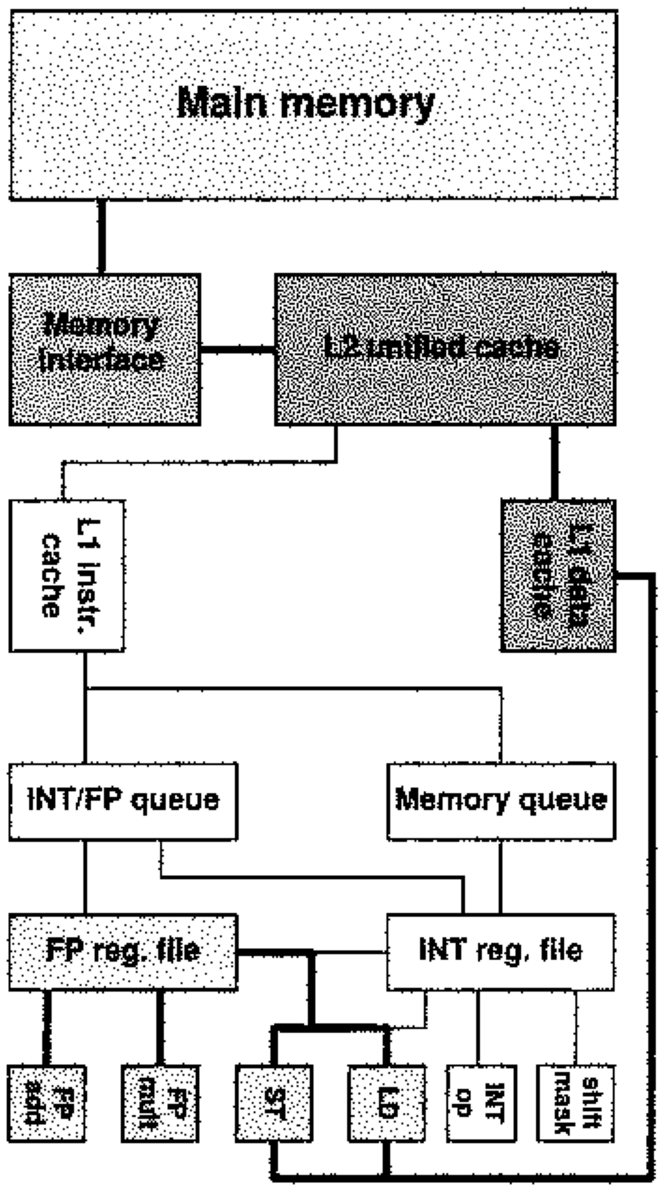
Today

Basic computer architecture

Introduction to parallel computing

Imperial College
London

Schematic of single core processor



RAM

L2 Cache

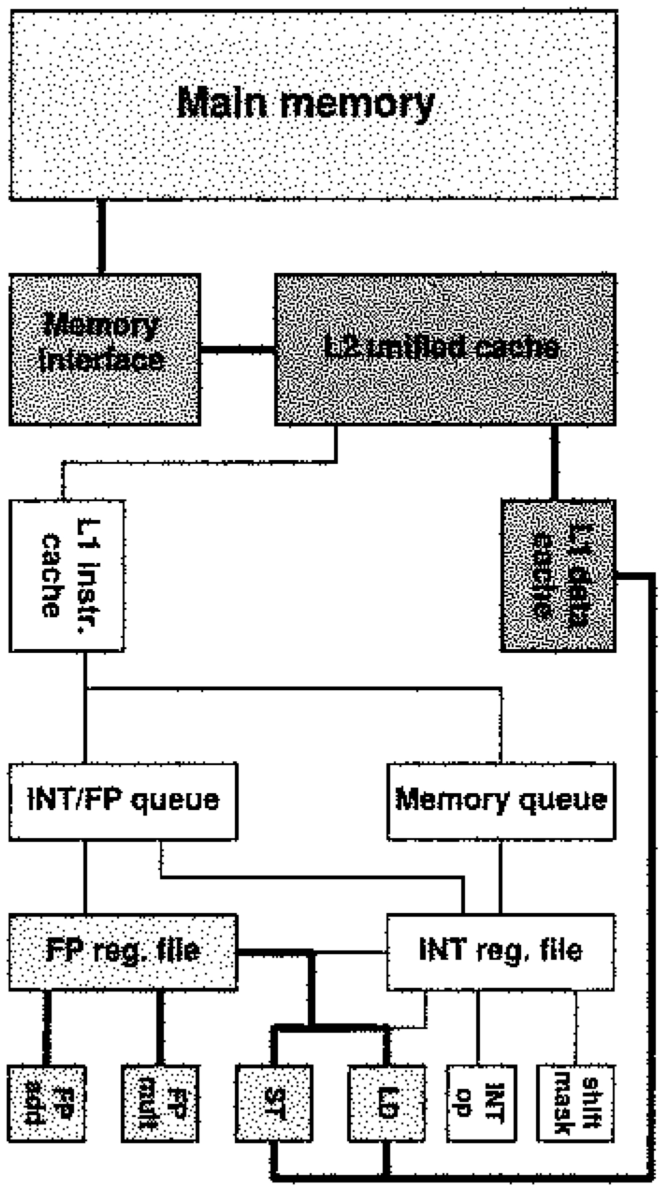
L1 Cache

Registers

Arithmetic units

(Image from Hager & Wellein)

Schematic of single core processor



RAM

L2 Cache (Can also have L3 cache shared across cores)

L1 Cache

Registers

Arithmetic units

(Image from Hager & Wellein)

CPU and memory

- CPUs typically have clock speeds of ~1-2 GHz
  - Typically, CPUs can produce 2-4 double-precision floating-point results per cycle
  - So, *peak performance is* ~4e9 FLOPS, or 4 gigaflops
- Performance often limited by movement of data in and out of the arithmetic units
- Need to consider memory hierarchy

Memory hierarchy

- Generally, the closer the memory is to the arithmetic unit, the faster and smaller the memory
- Hard drive: very large (~500 gb), *very* slow
- Main memory (RAM): large (~2 gb), sort-of fast (~1 GHz)
  - All computations, applications, etc… should fit in main memory

Memory hierarchy

- Generally, the closer to the arithmetic unit, the faster and smaller the memory

- Hard drive: very large (~500 gb), *very* slow

- Main memory (RAM): large (~2 gb), sort-of fast (~1 GHz)
  - All computations, applications, etc… should fit in main memory

- For my laptop (core i5 processor):
  - L3 Cache → 3mb (shared by two cores)
    - (2e6 bytes/8) = 250,000 double precision numbers
      = 500 x 500 matrix
  - L2 Cache → 256 kb (per core)
    - 256000/8 = 32000 double precision numbers
      = ~180 x 180 matrix
  - L1 Cache → 64 kb (per core), half for data, half for instructions
    - 4000 double precision numbers

Memory hierarchy

Memory access: data can be moved from registers to arithmetic units once each clock cycle:

- **For my laptop (core i5 processor):**
    - **Main memory → ~ 800 clock cycles**
    - **L2 Cache → ~ 11 clock cycles**
    - **L1 Cache → ~ 4 clock cycles**

Example

- **Construct three *n*-element arrays:** b,c,d
- **Within a loop, compute** a = b + c*d
- **Collect timing information as *n* varies**

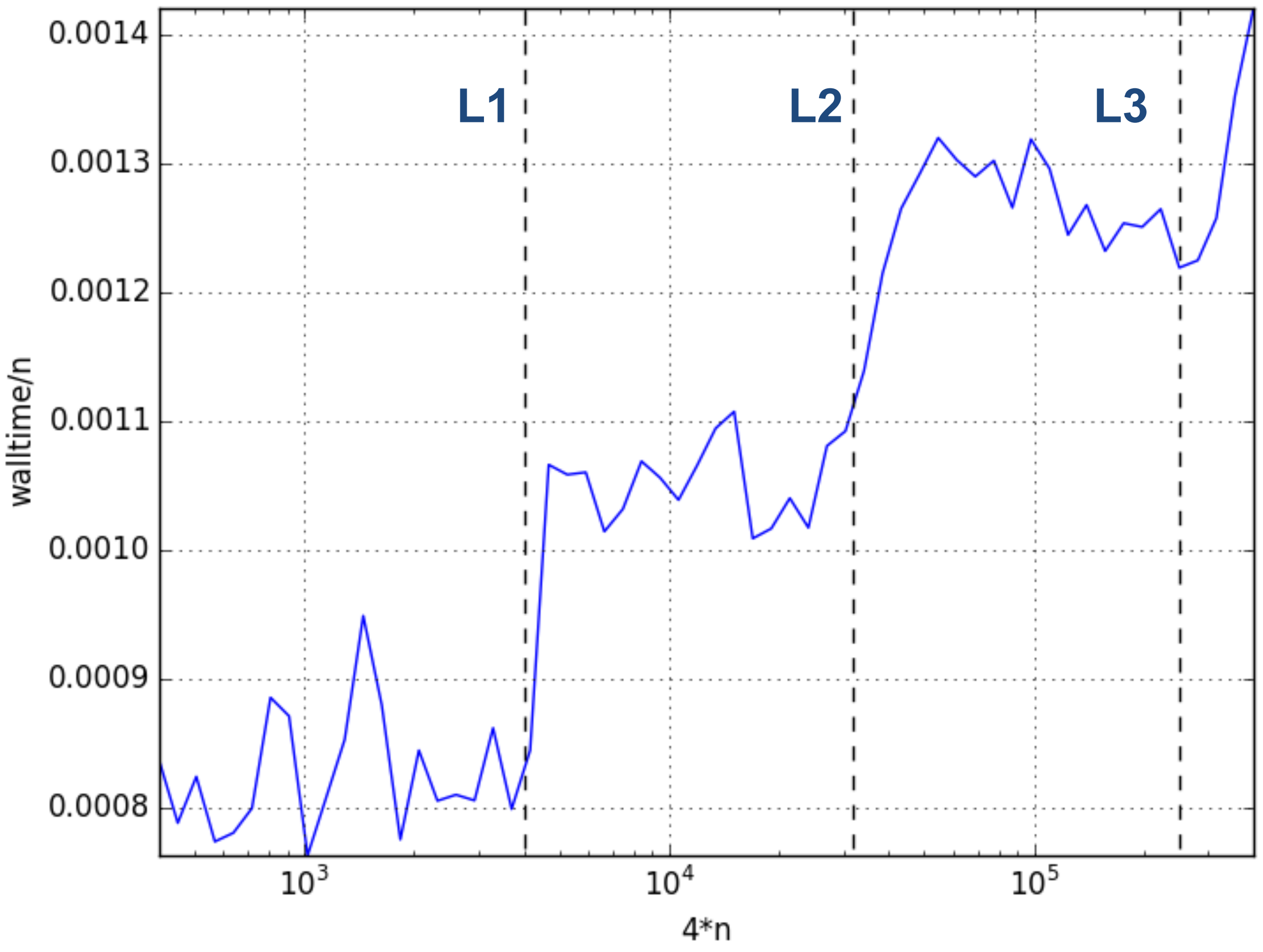
- ***triad.f90 (used by triad.py):***

```
real(kind=8), dimension(n) :: a,b,c,d

call system_clock(start)
do j1 = 1,1000000
    do i1 = 1,n
        a(i1) = b(i1) + c(i1) * d(i1)
    end do
    if (a(2) < 0) call dummy(a,b,c,d)
end do
call system_clock(stop,clockrate)
```

Example

Results:



- **Vertical lines indicate cache sizes**
- **Clear performance loss when a cache level becomes full**

Temporal locality

Due to the influence of cache size, important to think about where data is stored:

- If possible, data in cache should be re-used as much as possible (temporal locality)
  - “cache hit”: needed data is found in cache,
  - “cache miss”: data is not in (nearest cache)
    - old data needs to be moved out, new data moved in
    - data is moved in cache lines (typically 8 or 16 floats)
- Re-using data in cache reduces cache misses and associated performance loss.

Spatial locality

- Since data is passed in “lines,” there can be performance gain when using data which is adjacent in memory
- There can also be a penalty when using data that is scattered in memory
- Consider how matrices are stored in memory

$$A = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 16 \\ 2 & 6 & 10 & 20 \\ 3 & 7 & 11 & 24 \end{bmatrix}$$

Fortran:

- 0,1,2,3 occupy consecutive locations in memory
- “column-major” ordering

Imperial College London

Spatial locality

- Since data is passed in “lines,” there can be performance gain when using data which is adjacent in memory
- There can also be a penalty when using data that is scattered in memory
- Consider how matrices are stored in memory

$$A = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 16 \\ 2 & 6 & 10 & 20 \\ 3 & 7 & 11 & 24 \end{bmatrix}$$

Python (and c):

- **0,4,8,12 occupy consecutive locations in memory**
- **“row-major” ordering**
- **When using** np.array**, can force ‘Fortran ordering’**

Spatial locality: simple example

- Really only important for very large matrices:
  - Create 20000 x 20000 random matrix
  - Compute statistics by manually looping along rows and columns
  - Use timer to calculate wall time

```
for i in range(0,n):
  av += np.average(H[i,:])
  sum += np.sum(H[i,:])
  var += np.var(H[i,:])
  std += np.std(H[i,:])
```

```
time: 4.8690969944
```

```
for i in range(0,n):
  av += np.average(H[:,i])
  sum += np.sum(H[:,i])
  var += np.var(H[:,i])
  std += np.std(H[:,i])
```

```
time: 132.157668114
```

- Better to first compute transpose, then loop across a row.

Code optimization

- Can use these ideas about cache to improve/optimize code
  - With compiled languages, the compiler will do much of the optimization for you
  - Working with large matrices in interpreted languages requires greater care

Code optimization

- Can use these ideas about cache to improve/optimize code
    - With compiled languages, the compiler can do much of the optimization for you
    - Working with large matrices in interpreted languages requires greater care
    - Most important point: first develop a code that works, then optimize it.

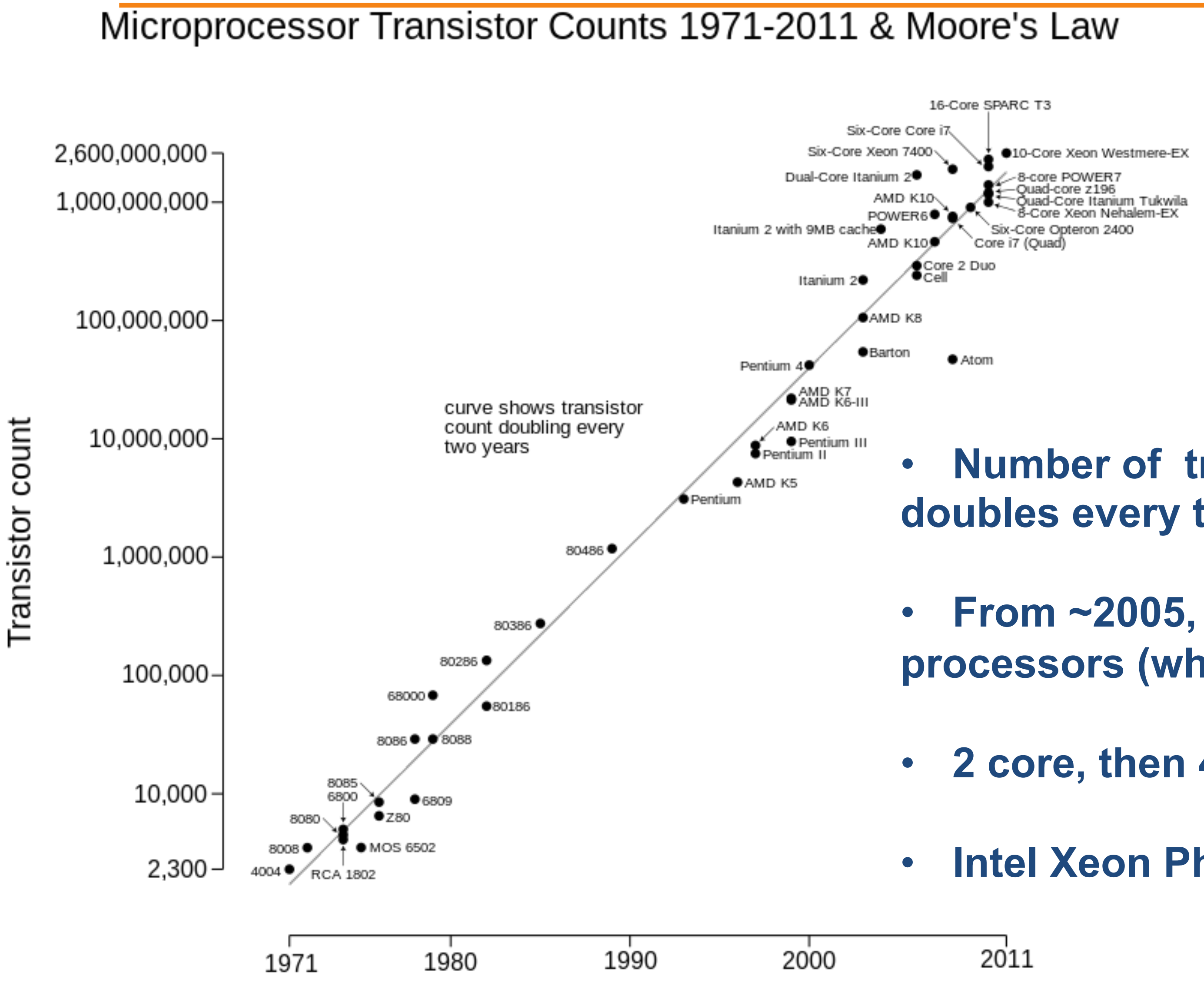
Does anyone have any questions?

Does anyone have any questions?

What does "high performance" mean?

Imperial College
London

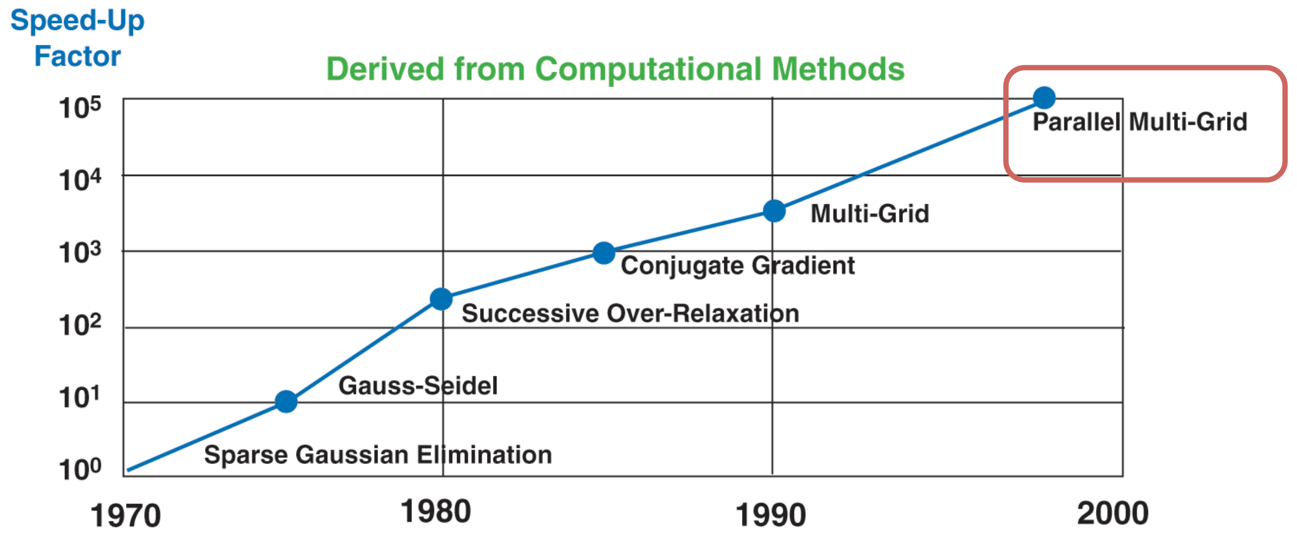
Moore's law

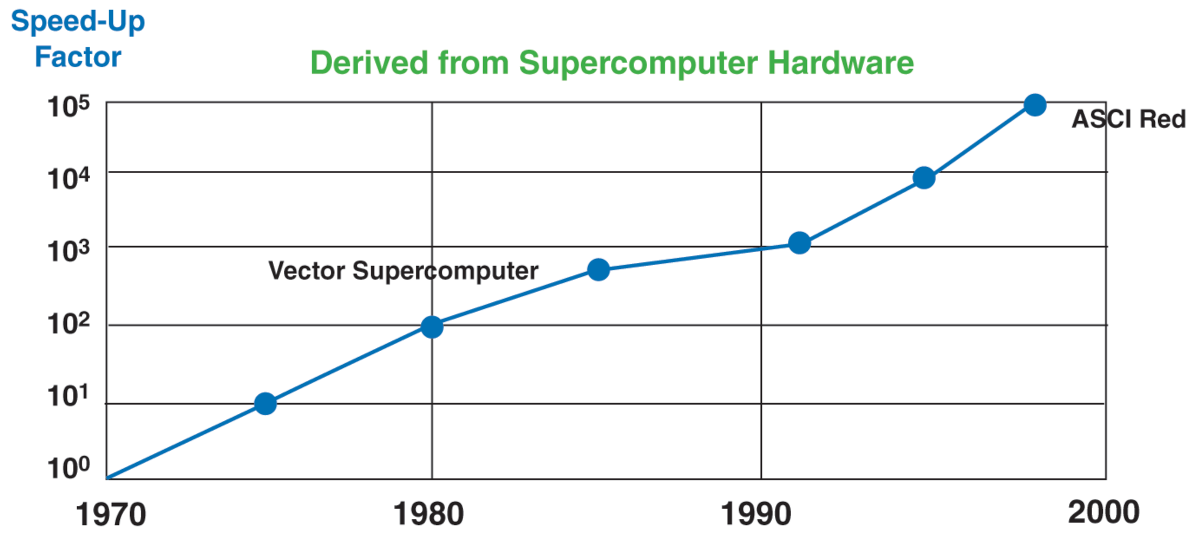


- Number of transistors on chip doubles every two years
- From ~2005, multicore processors (why?)
- 2 core, then 4, 6, 8, 16
- Intel Xeon Phi: 60+ cores!

Imperial College London

Algorithms and hardware





Imperial College London

SIAM Rev (2001)

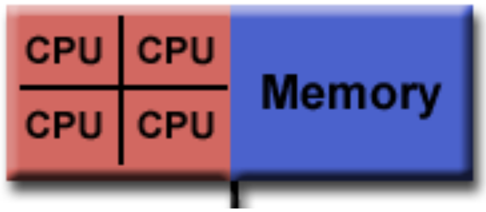
Why parallelize a code?

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

or

2 Serial code is too big

Parallel computing paradigms

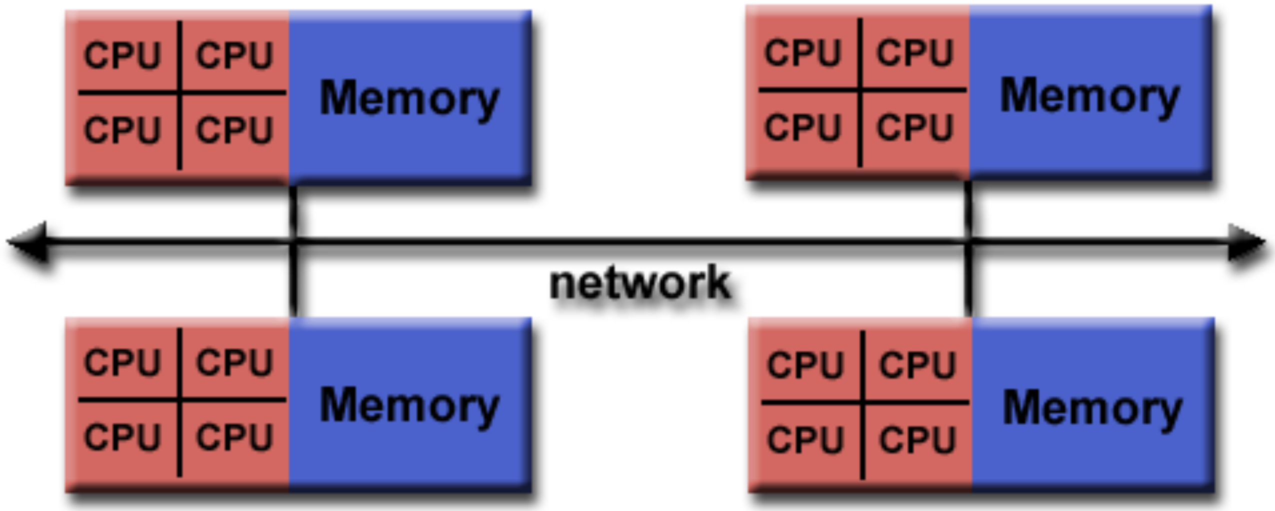


Shared memory

- 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*

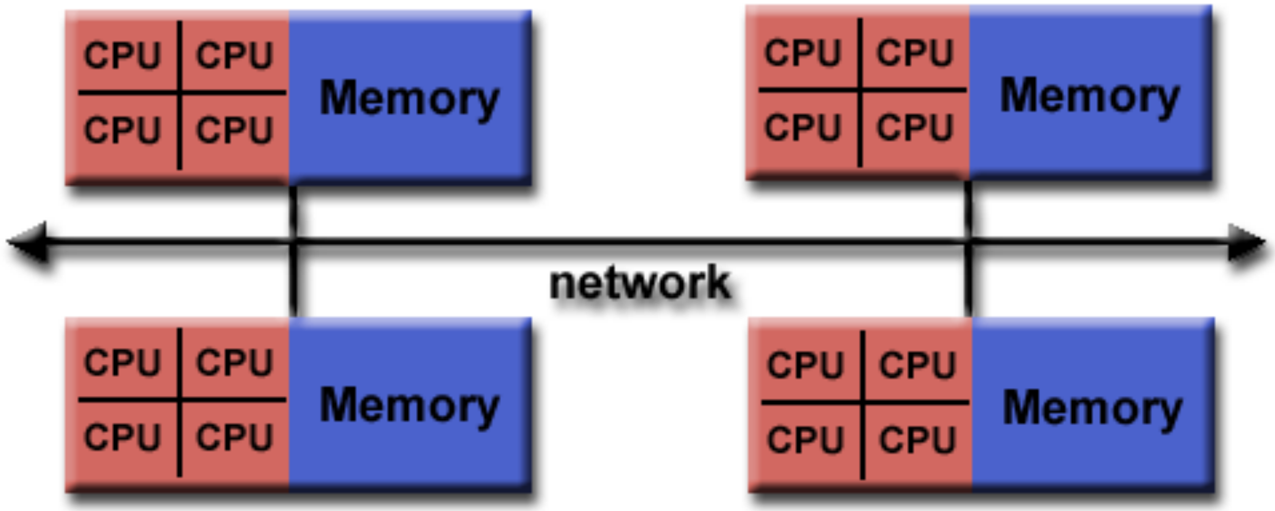
Parallel computing paradigms



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

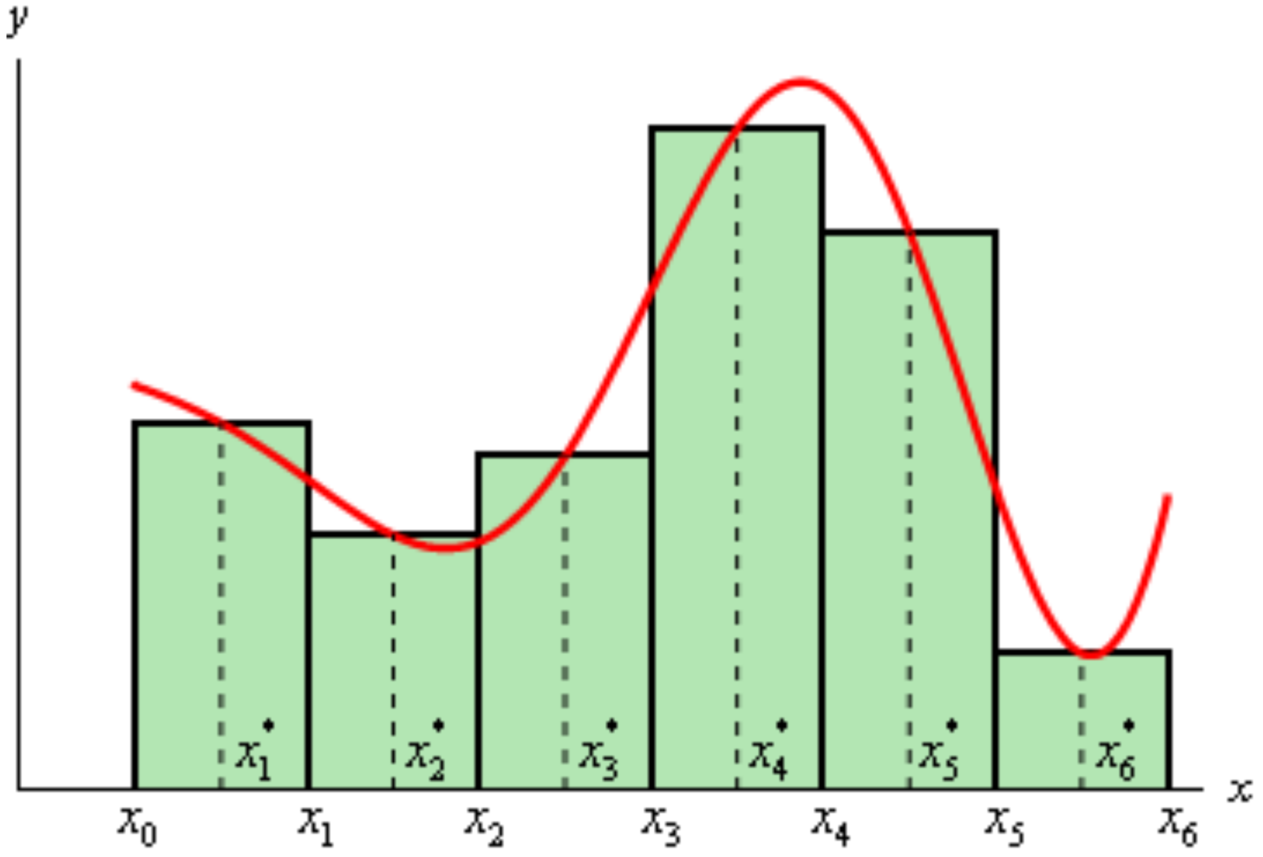
Parallel computing paradigms



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:

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

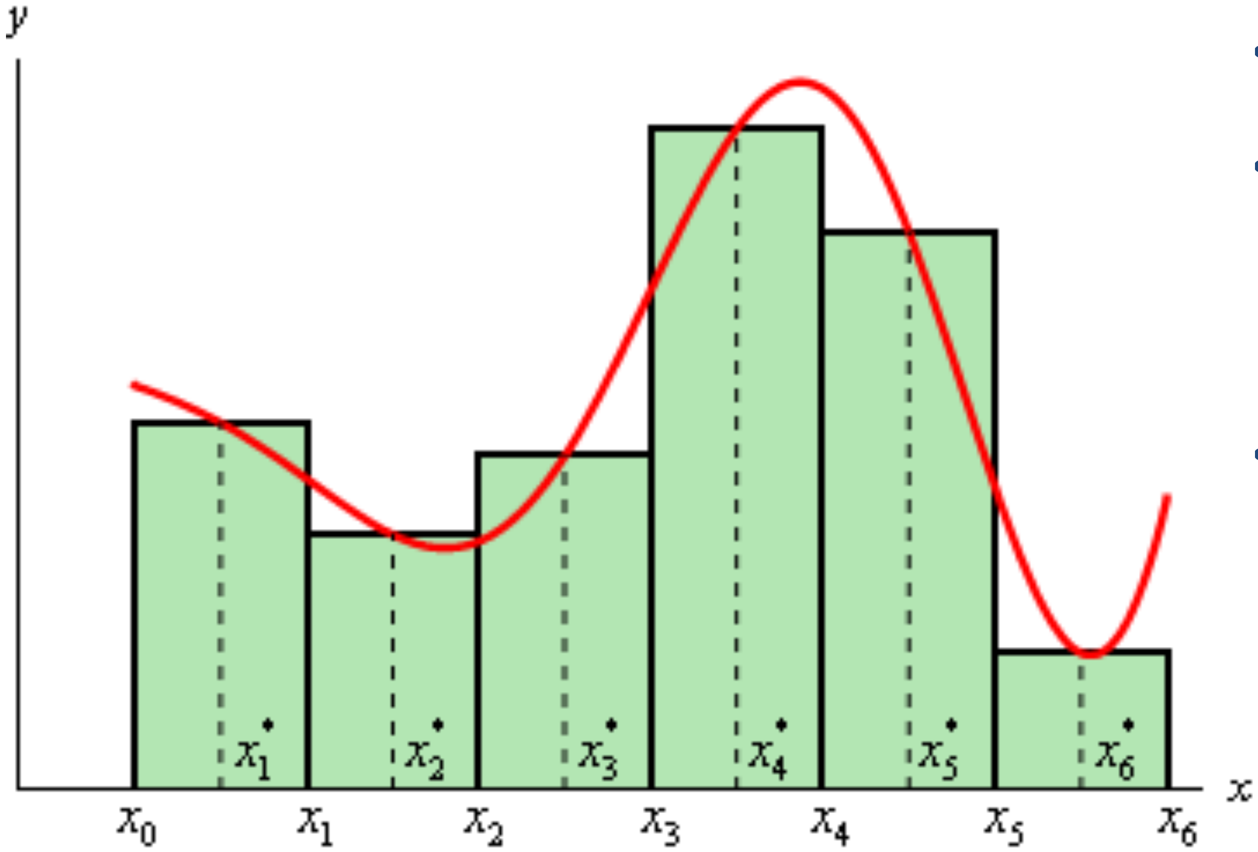
2. Compute areas of rectangles:

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

3. Sum areas:

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

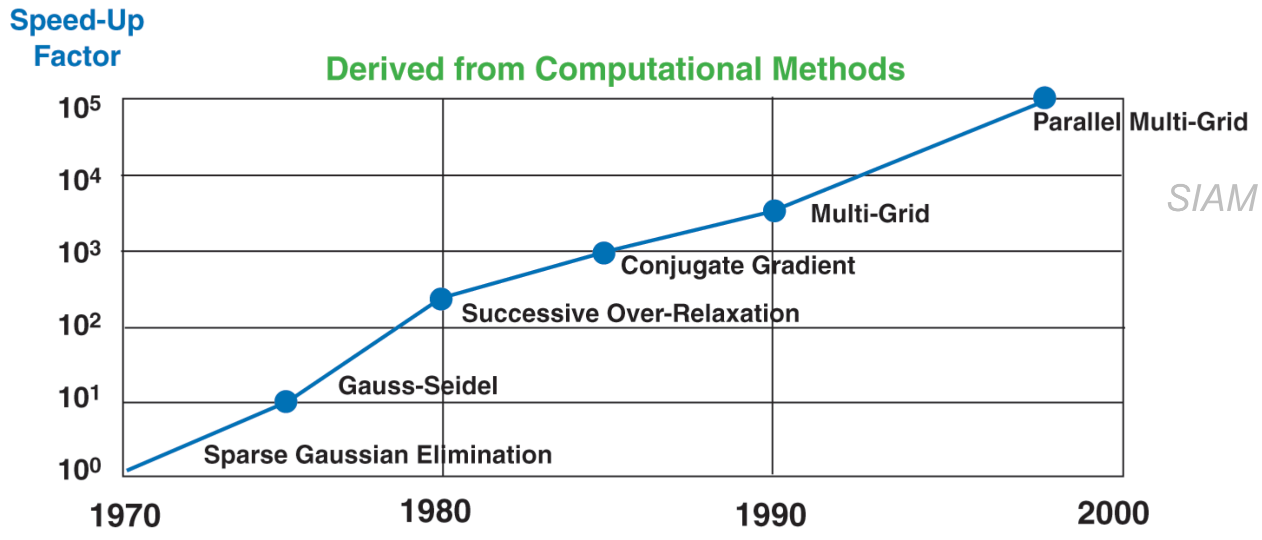
Example: computing an integral



- 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

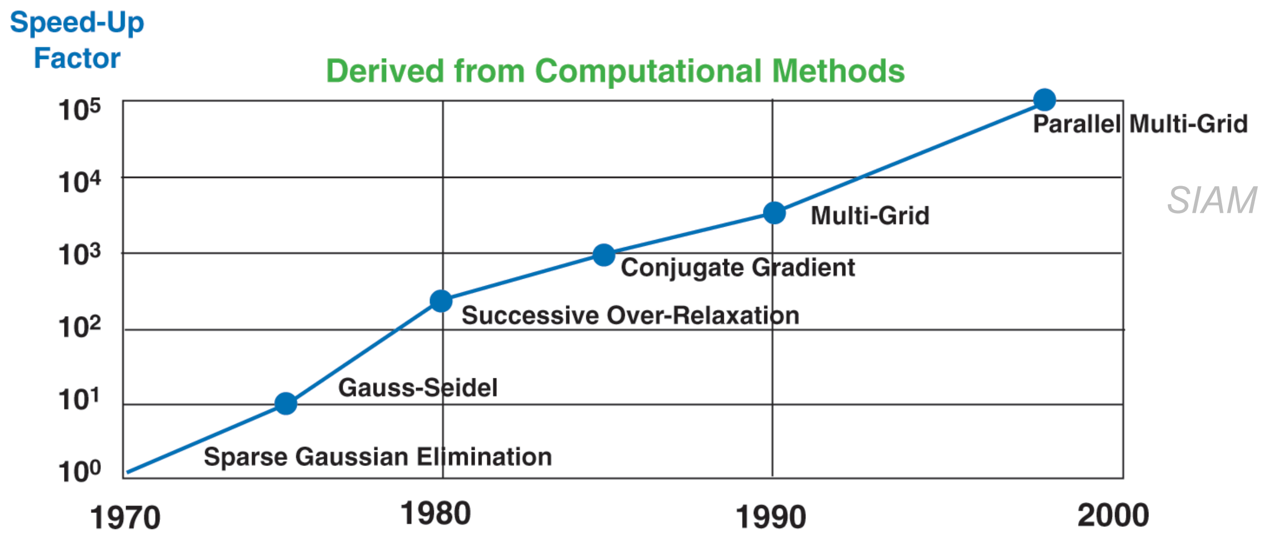


SIAM Rev (2001)

- **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***

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