E7441: Scientific computing in biology and biomedicine

Introduction to parallel computing

Vlad Popovici, Ph.D.

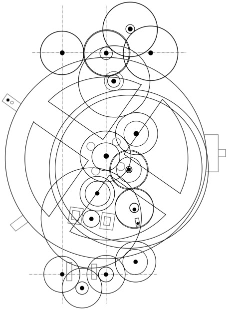
Fac. of Science - RECETOX

Outline

1. A historical perspective
2. Why parallel computing?
3. Principles of parallel computing
   - Introduction
   - Programming models
   - Implementations

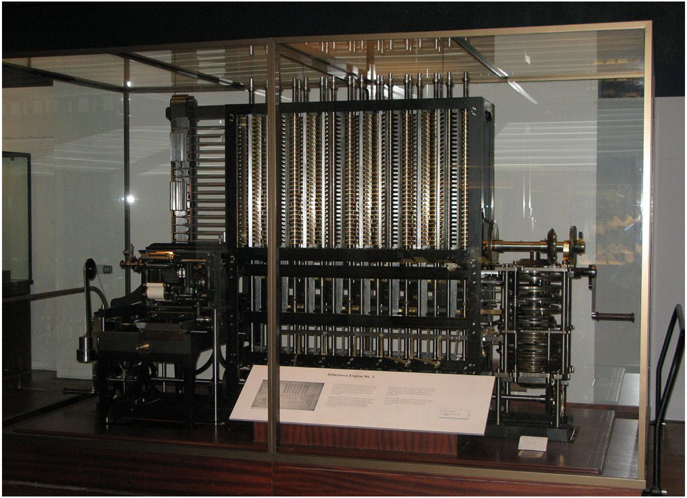
- "I think there is a world market for maybe five computers." (Thomas Watson, chairman of IBM, 1943)
- "There is no reason for any individual to have a computer in their home." (Ken Olson, founder Digital Equipment Corporation, 1977)
- "640K of memory ought to be enough for anybody." Bill Gates, chairman of Microsoft, 1981

- ~ 2500 BC: Babylon - the first abacus
- ~ 100 BC: Antikythera device - believed to be the first mechanical computer
- first half of the 19th century: Charles Babbage's differential machine (to tabulate polynomials) and analytical machine (only design)
- 1941: Z3 computer by Konrad Zuse: first programmable, fully automatic computing machine



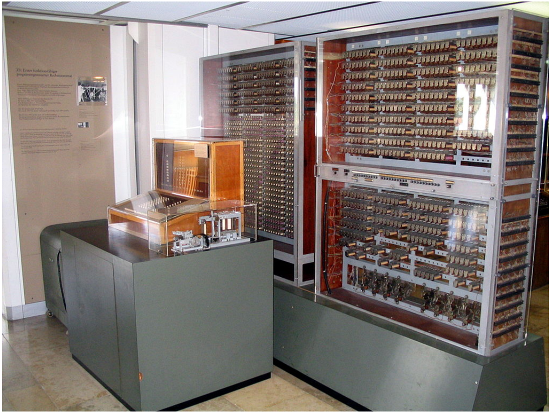
source: Wikipedia

$\sim$ 1840 Charles Babbage produces the differential machine, a mechanical computer.



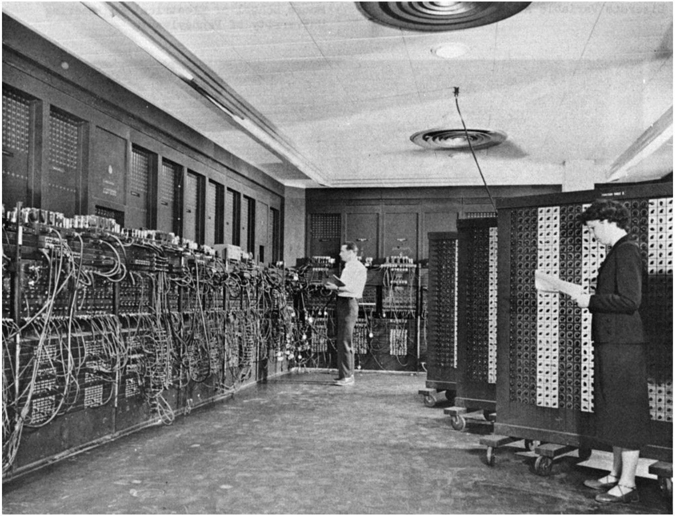
source: Wikipedia

1941: Z3 computer: electro-mechanical computer, $\sim$ 2000 relays, 22-bit words, operating at 5-10 Hz.



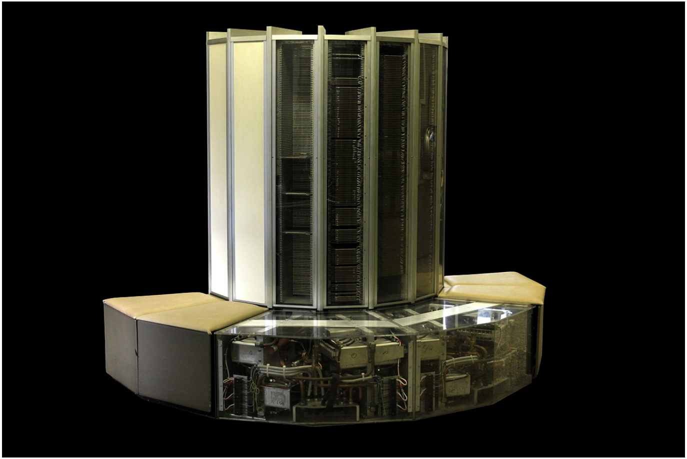
source: Wikipedia

1946: ENIAC - Electronic Numerical Integrator And Computer used initially by US Army to compute tables for artilery. Uses vacuum tubes as switches.



source: Wikipedia

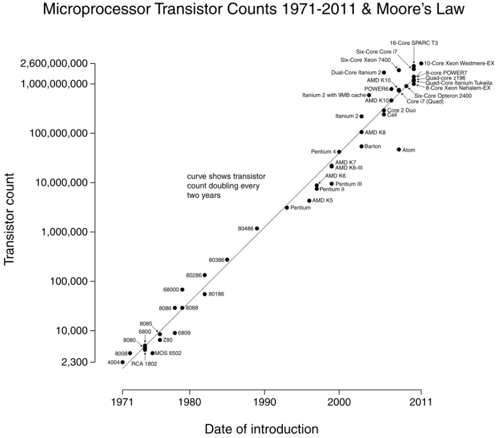
1976: Cray-1 - the first successful supercomputer



...fast forward: Tianhe-2 (top supercomputer as Nov. 2013): 33.86 PFlop/s, 3,120,000 cores; 1,024,000 GB, CPU: Intel Xeon

Moore’s law

Gordon E. Moore (co-founder Intel): "Cramming More Components onto Integrated Circuits", Electronics Magazine, 1965



source: Wikipedia

Software and hardware

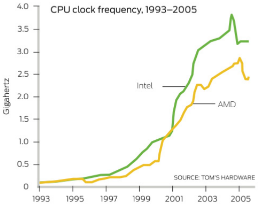
Software crises:

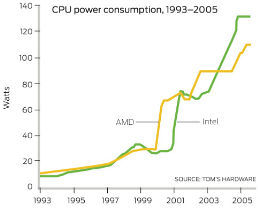
- '60s-'70s: assembly language difficult to use for large complex problems $\rightarrow$ Fortran, C: provide abstraction and portability for uniprocessors
- '80s-'90s: problems in maintaining complex systems $\rightarrow$ object-oriented programming (C++, Java)
- $\sim 2000s$: sequential performance lags behind Moore's law $\rightarrow$ programmers are oblivious to hardware better compilers, higher level languages, virtual machines

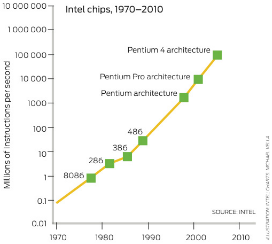
- parallel computing: using multiple execution units concurrently to solve a problem
- examples:
  - multi-core processors: several processors (cores) in a chip
  - shared memory processors (SMP): several processors interconnected through a shared memory
  - cluster computer: several computers interconnected through high-speed network

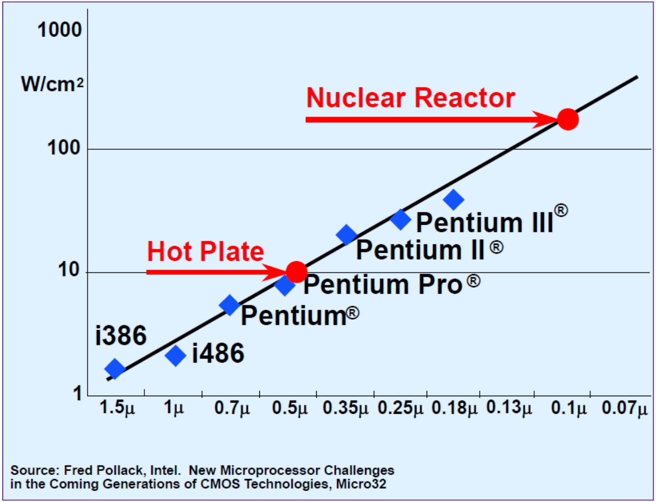
Issues with the traditional model: power density

(Ross: Why CPU Frequency Stalled, IEEE Spectrum Magazine, 2008)



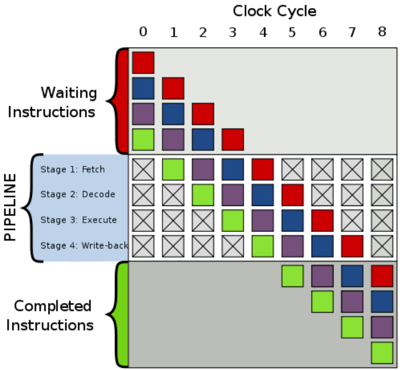





1000
W/cm2
100
10
1
Nuclear Reactor
Hot Plate
Pentium III®
Pentium II®
Pentium Pro®
Pentium®
i386
i486
1.5μ
1μ
0.7μ
0.5μ
0.35μ
0.25μ
0.18μ
0.13μ
0.1μ
0.07μ
Source: Fred Pollack, Intel. New Microprocessor Challenges in the Coming Generations of CMOS Technologies, Micro32

Issues cont'd: gains from implicit parallelism tapped out

Example: instruction-level parallelism. Machine instruction: decomposed into 4-stages: fetch, decode, execute and write-back



Issues cont'd

Other issues:

- increase in production costs (decrease in "chip yield")
- increase in amount of data to be processed

Solution: explicit parallelism

- multi-core
- multi-processor
- multi-machine

Principles

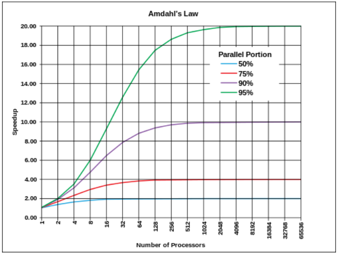
- identifying parallelism
- granularity: more smaller or fewer larger tasks?
- locality: data and instruction location
- load balance: aim: no lost CPU cycles
- synchronization
- overhead

Identifying parallelism

Amdahl's law:

$$S_n = \frac{T_1}{T_n} \leq \frac{1}{\alpha + (1-\alpha)/n} \leq \frac{1}{\alpha}$$

where α is the fraction of the program that is strictly sequential, T_i is the execution time on i processors and S_i is the speed-up obtained by using i processors instead of 1.

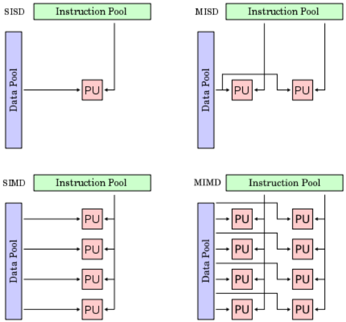


Identifying parallelism

- *implicit parallelism*
  - hardware level: superscalar processors, multi-core, cluster computing
  - compiler level: parallelizing compilers
- *explicit parallelism*
  - programming language level
  - library level

Processing architectures

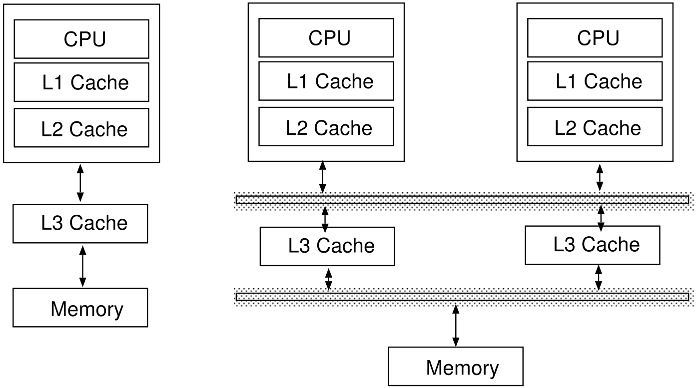
Flynn's taxonomy ("old way"): *S*ingle/*M*ultiple Instruction × *S*ingle/*M*ultiple Data



Source: Wikipedia

Examples: SISD: mainframes; SIMD: GPUs; MISD: fault tolerant systems; MIMD: most computers nowadays

Locality: a box in a box in a box...



Computing topologies

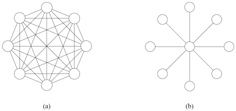


Figure 2.14 (a) A completely-connected network of eight nodes; (b) a Star connected network of nine nodes.

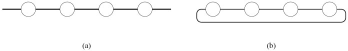


Figure 2.15 Linear arrays: (a) with no wraparound links; (b) with wraparound link.

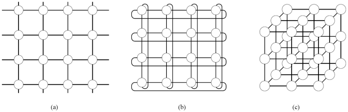
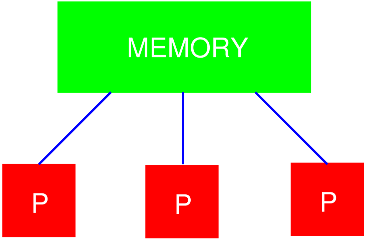


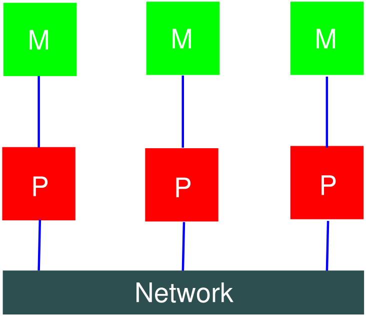
Figure 2.16 Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

Source: Grama - Introduction to Parallel Computing

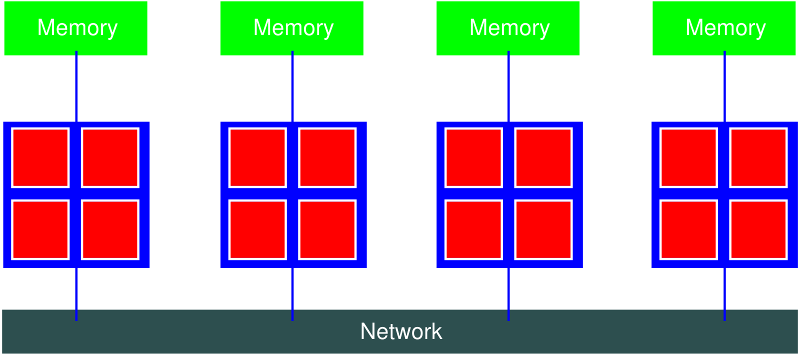
Shared memory: multicore or multi-CPU machines



Distributed memory: clusters with single CPUs nodes

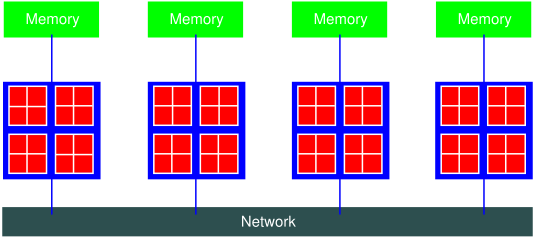


Hybrid systems



- a limited number of CPUs have access to a pooled memory
- using more CPUs implies communication over network through *message-passing*

Hybrid systems with multicore CPUs



- extension of the hybrid model
- communication becomes increasingly complex
- many levels in the memory hierachy: cache(s), local main memory, other node's memory, etc
- you can add accelerators: e.g, GPUs
- requires a new programming model, and different communication protocols

Load balancing

- aim: distribute evenly the *load* (work) on all available resources...
- ...and thus minimize the time a resource is idle
- causes of imbalanced load:
  - insufficient paralelism
  - unequal task size (poor design?)

Types of parallelism

- *data parallelism*: each processor performs the same task on different data (h/w: SIMD, MIMD)
- *task parallelism*: each processor performs a different task on the same data (h/w: MISD, MIMD)
- usually, both types of parallelism are present

Example: re-annotation of a microarray chip

(embarrassingly parallel problem)
Problem: map (BLAST) each probe from a microarray against the latest version of the human genome (RefSeq).
Naive implementation on 2 CPUs:

```
program:
...
if CPU == 'CPU1' then
  idx = 1,..,Np/2
elseif CPU == 'CPU2' then
  idx = Np/2 + 1,...,N
endif

BLAST(Probes[idx])
...
```

```
program:
...
idx = 1,..,Np/2

BLAST(Probes[idx])
...
```

```
program:
...
idx = Np/2 + 1,...,N

BLAST(Probes[idx])
...
```

Better ways of distributing the data exists for this problem! Ex: distribute also the RefSeq...

Problem decomposition

- split the computations into concurrent tasks
- build the task-dependency graph
- there is no one-size-fits-all technique
- some methods: *recursive decomposition*, *data-decomposition*, *exploratory decomposition* and *speculative decomposition*

Recursive decomposition: example

Problem: find the minimum of a vector

```
proc serial_min(A, n)
  min = A[1]
  for i = 2 to n do
    if A[i] < min
    then min = A[i]
  end for
  return min
end serial_min
```

```
proc rec_min(A, i, j)
  if i == j
  then min = A[i]
  else
    lmin = rec_min(A, i, j/2)
    rmin = rec_min(A, j/2+1, j)
    if lmin < rmin
    then min = lmin
    else min = rmin
    end if
  end if
  return min
end rec_min
```

Data decomposition: example

Matrix multiplication: $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$. Write it as

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix}$$

and distribute the four tasks:

$$\begin{aligned}
\text{Task 1: } \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\
\text{Task 2: } \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\
\text{Task 3: } \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\
\text{Task 4: } \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22}
\end{aligned}$$

Other decompositions

- *exploratory decomposition*: decompose the search space for the solution and search for a solution in each subspace; then choose among the solutions
- *speculative decomposition*: launch alternative computation branches in parallel while waiting for input for deciding which branch to use
- hybrid decompositions

Mapping techniques

- problem decomposition → tasks
- the tasks need to be allocated (mapped) to processors/processes
- objective: minimize the execution time
- *overheads*: time spent for everything else but actually solving the problem:
  - inter-process interaction - synchronization and control
  - time spent being idle - poor load balancing
- reduce the process inter-dependencies and communication: e.g. maximize data locality
- improve load balancing
- reduce blocking operations

Implementations on multihtread/multicore machines

- POSIX threads (*pthreads*): OS-level paralelism.
  - threads: lightweight processes
  - the same program runs on single- or multi-core machines
  - OS has the responsibility of mapping the threads
  - needs low-level programming, dedicated library
- *OpenMP*: built on top of pthreads for SIMD-kind of parallelism
  - implemented through compiler directives
  - easier to use than pthreads
  - performance depends on compiler's 'intelligence'

OpenMP: how does it look like? ($\sum_i a_i b_i$)

```
  double a[N];
  double sum = 0.0;
  int i, n, tid;

#pragma omp parallel shared(a) private(i)
  {
    tid = omp_get_thread_num();

    /* Only one of the threads do this */
#pragma omp single
    {
      n = omp_get_num_threads(); printf("Number of threads = %d\n", n);
    }
    /* Initialize a */
#pragma omp for
    for (i=0; i < N; i++) {
      a[i] = 1.0;
    }

    /* Parallel for loop computing the sum of a[i] */
#pragma omp for reduction(+:sum)
    for (i=0; i < N; i++) {
      sum = sum + (a[i]);
    }

  } /* End of parallel region */
```

Implementations on distributed-memory systems

- *MPI: Message Passing Interface*
  - de facto standard for distributed memory programming (clusters)
  - data must be manually decomposed
  - use special libraries
  - based on *sending* and *receiving* messages: data and synchronization
- *PVM*: Parallel Virtual Machine
  - previous library for cluster programming
  - based on message-passing principle
  - supplanted by MPI

MPI: how does it look like?

```
#include <mpi.h>
int main(int argc, char *argv[])
{
  int numprocs, myid;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* print out my rank and this run's PE size */
  printf("Hello from %d of %d\n", myid, numprocs);

  MPI_Finalize();
}
```

Implementations in R

- parallelism came as an after thought
- target: massive data applications
- tries to bring to R some of the libraries existing to other languages
- *snow*: for traditional clusters, supports PVM, MPI,...; is portable (UNIX, Windows)
- *multicore*: targets multi-core/-CPU machines; simple; does not run on Windows; does not handle parallel RNGs
- *parallel*: snow+multicore in new R (>=2.14); strange interactions with OS
- *R+Hadoop*: based on Hadoop cluster
- *RHIPE*: based on Hadoop, targets map-reduce operations
- *Segue*: APPLY-like calculations on Hadoop clusters, using Amazon's Elastic MapReduce

Questions?