Introduction

HPC Fall 2010

Prof. Robert van Engelen

Syllabus

- Title: "*High Performance Computing*" (ISC5318 and CIS5930-1)
- Classes: Monday and Wednesday 12:30PM to 1:45PM in 103 LOV
- Evaluation: projects (40%), homework (20%), midterm exam (20%), and final exam (20%)
- Prerequisites: experience programming in either Java, C, C++, or Fortran
- Accounts: you need an SCS account to access machines
- Instructor: Prof. Robert van Engelen, office hour Tuesday from 12:30PM to 1:30PM in 160 LOV and upon request

http://www.cs.fsu.edu/~engelen/courses/HPC

Books

- [HPC] "*Software Optimization for High Performance Computing: Creating Faster Applications*" by K.R. Wadleigh and I.L. Crawford, Hewlett-Packard professional books, Prentice Hall.
- [OPT] "*The Software Optimization Cookbook*" (2nd ed.) by R. Gerber, A. Bik, K. Smith, and X. Tian, Intel Press.
- [PP2] "*Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*" (2nd ed.) by B. Wilkinson and M. Allen, Prentice Hall.
- [PSC] "*Parallel Scientific Computing in C++ and MPI*" by G. Karniadakis and R. Kirby II, Cambridge University Press.
- [SPC] "*Scientific Parallel Computing*" by L.R. Scott, T. Clark, and B. Bagheri, Princeton University Press.
- [SRC] "*Sourcebook of Parallel Programming*" by J. Dongara, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White (eds), Morgan Kaufmann.

Course Outline

- *Introduction*
- *Architecture and Compilers*
  - E.g. levels of parallelism, CPU and memory resources, types of (parallel) computers, compilation techniques to improve CPU and memory access
- *Performance Analysis*
  - E.g. timing code, finding hotspots, profiling, measuring message latency
- *Programming Models*
- *Programming with Shared Memory*
  - E.g. threads, openMP, locks, barriers, automatic parallelization
- *Programming with Message Passing*
  - E.g. MPI, communications, MPE and jumpshot, debugging
- *Algorithms*
  - E.g. embarrassingly parallel, synchronous, pipelined, partitioning and divide and conquer strategies, parallel numerical algorithms
- *High-Performance Libraries, Programming Languages and Tools*

Introduction

- Why parallel?
- … and why not!
- Speedup, efficiency, and scalability of parallel algorithms
- Laws
- Limitations to speedup
- The future of computing
- Lessons Learned
- Further reading

Why Parallel?

- A programmer should first ask "*why parallel?*"
- It is not always obvious that a parallel algorithm has benefits, unless we want to do things …
  - faster: doing the same amount of work in less time
  - bigger: doing more work in the same amount of time
- Both of these reasons can be argued to produce *better results*, which is the only meaningful outcome of program parallelization

Why Parallel? Faster, Bigger!

- There is an ever increasing demand for computational power to improve the speed or accuracy of solutions to real-world problems through *faster* computations and/or *bigger* simulations

- Computations must be completed in acceptable time (real-time computation), hence must be “fast enough”

Why Parallel? Faster, Bigger!

- An illustrative example: a weather prediction simulation should not take more time than the real event

- Suppose the atmosphere of the earth is divided into 5×10^8 cubes, each $1 \times 1 \times 1$ mile and stacked 10 miles high
- It takes 200 floating point operations per cube to complete one time step
- 10^4 time steps are needed for a 7 day forecast
- Then 10^{15} floating point operations must be performed
- This takes 10^6 seconds (= 10 days) on a 1 GFLOP machine

Why Parallel? Grand Challenge Problems

- Big problems
  - A "*Grand Challenge*" problem is a problem that cannot be solved in a reasonable amount of time with today's computers
  - Examples of Grand Challenge problems:
    - Applied Fluid Dynamics
    - Meso- to Macro-Scale Environmental Modeling
    - Ecosystem Simulations
    - Biomedical Imaging and Biomechanics
    - Molecular Biology
    - Molecular Design and Process Optimization
    - Fundamental Computational Sciences
    - Nuclear power and weapons simulations

Why Parallel? Physical Limits

- Which tasks are fundamentally too big to compute with one CPU?
- Suppose we have to calculate *in one second*

```
for (i = 0; i < ONE_TRILLION; i++)
  z[i] = x[i] + y[i];
```

- Then we have to perform 3×10^{12} memory moves per second
- If data travels at the speed of light (3×10^8 m/s) between the CPU and memory and r is the average distance between the CPU and memory, then r must satisfy

$$3 \times 10^{12}\ r = 3 \times 10^8\ \text{m/s} \times 1\ \text{s}$$

  which gives $r = 10^{-4}$ meters
- To fit the data into a square so that the average distance from the CPU in the middle is r, then the length of each memory cell will be

$$2 \times 10^{-4}\ \text{m} / (\sqrt{3} \times 10^6) = 10^{-10}\ \text{m}$$

  which is the size of a relatively small atom

Why Parallel? Important Factors

- Important considerations in parallel computing
  - *Physical limitations*: the speed of light, CPU heat dissipation
  - *Economic factors*: cheaper components can be used to achieve comparable levels of aggregate performance
  - *Scalability*: allow problem sizes to be subdivided to obtain a better match between algorithms and resources (CPU, memory) to increase performance
  - *Memory*: allow aggregate memory bandwidth to be increased together with processing power at a reasonable cost

… and Why not Parallel?

- Bad parallel programs can be worse than their sequential counterparts
  - Slower: because of communication overhead
  - Scalability: some parallel algorithms are only faster when the problem size is very large
- Understand the problem and use common sense
- Not all problems are amenable to parallelism
- In this course we will focus a significant part on non-parallel optimizations

… and Why not Parallel?

- Some algorithms are inherently sequential
- Consider for example the *Collatz conjecture*, implemented by

```
int Collatz(int n)
{ int step;
  for (step = 1; n != 1; step++)
  { if (n % 2 == 0) // is n is even?
      n = n / 2;
    else
      n = 3*n + 1;
  }
  return step;
}
```

- Given *n*, **`Collatz`** returns the number of steps to reach *n* = 1
- Conjecture: algorithm terminates for any integer *n* > 0
- This algorithm is clearly sequential
- Note: given a *vector* of *k* values, we can compute *k* Collatz numbers in parallel

Speedup

- Suppose we want to compute in parallel

```
for (i = 0; i < N; i++)
  z[i] = x[i] + y[i];
```

- Then the obvious choice is to split the iteration space in *P* equal-sized *N*/*P* chunks and let each processor share the work (*worksharing*) of the loop:

```
for each processor p from 0 to P-1 do:
  for (i = p*N/P; i < (p+1)*(N/P); i++)
    z[i] = x[i] + y[i];
```

- We would assume that this parallel version runs *P* times faster, that is, we hope for *linear speedup*
- Unfortunately, in practice this is not the case because of processor overhead, communication, and synchronization

Speedup

- **Definition**: the *speedup* of an algorithm using P processors is defined as

$$S_P = t_s / t_P$$

where t_s is the execution time of the *best available sequential algorithm* and t_P is the execution time of the parallel algorithm

- The speedup is linear (*perfect* or *ideal speedup*) if $S_P \approx P$

- The speedup is *superlinear* when $S_P > P$

Relative Speedup

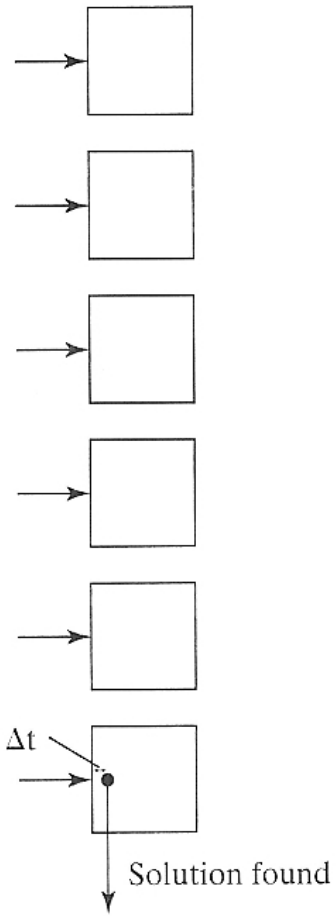
- **Definition**: The *relative speedup* is defined as
$$S^1{}_P = t_1 / t_P$$
where t_1 is the execution time of the parallel algorithm on one processor

- Similarly, $S^k{}_P = t_k / t_P$ is the relative speedup with respect to k processors, where $k < P$

- The relative speedup $S^k{}_P$ is used when k is the smallest number of processors on which the problem will run

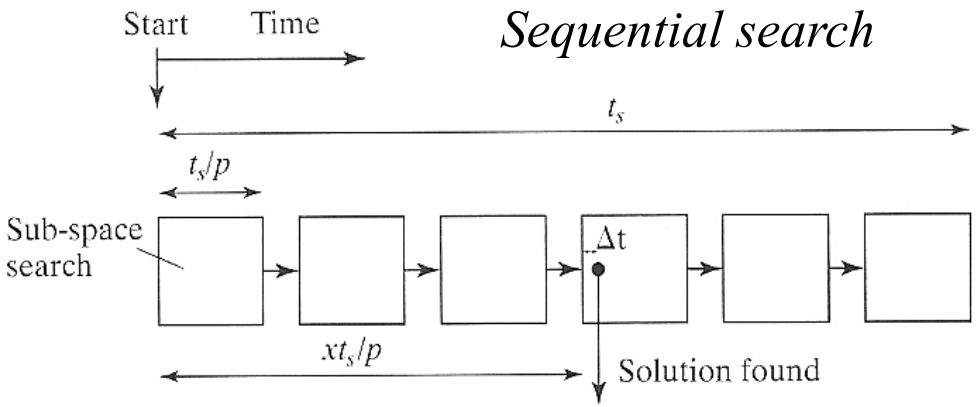
An Example

Parallel search



- Search in parallel by partitioning the search space into P chunks
- $S_P = ((x \times t_s/P) + \Delta t) / \Delta t$
- Worst case for sequential search (item in last chunk): $S_P \rightarrow \infty$ as Δt tends to zero
- Best case for sequential search (item in first chunk): $S_P = 1$

Sequential search



Effects that can Cause Superlinear Speedup

- *Cache effects*: when data is partitioned and distributed over *P* processors, then the individual data items are (much) smaller and may fit entirely in the data cache of each processor

- For an algorithm with linear speedup, the extra reduction in cache misses may lead to superlinear speedup

Efficiency

- **Definition**: the *efficiency* of an algorithm using P processors is

$$E_P = S_P / P$$

- Efficiency estimates how well-utilized the processors are in solving the problem, compared to how much effort is lost in communication and synchronization

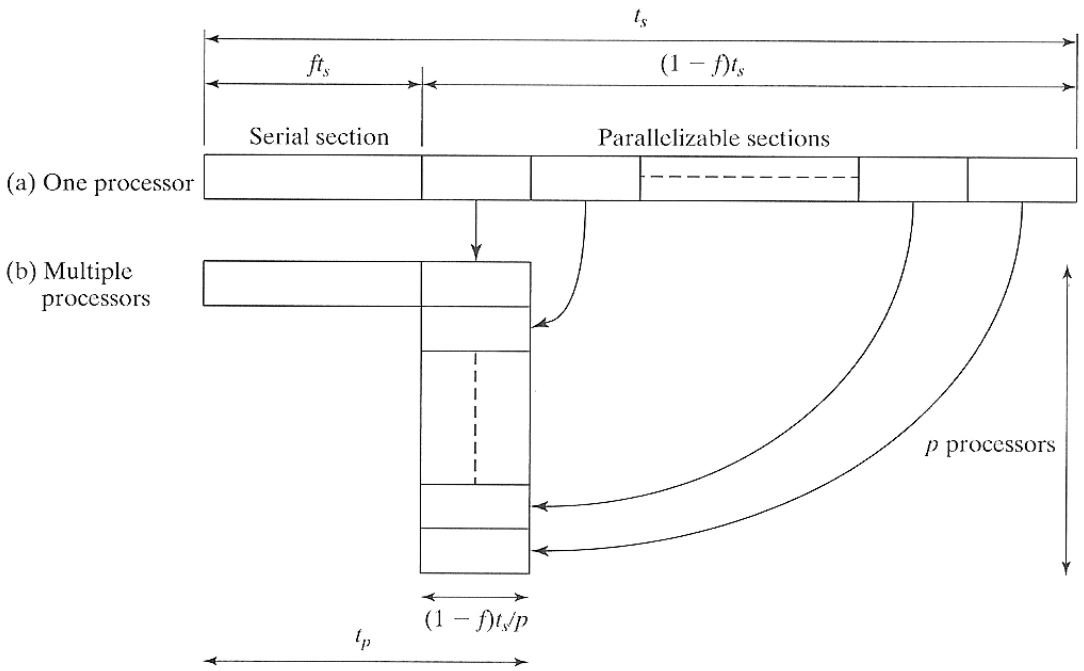
- Algorithms with ideal speedup and algorithms running on a single processor have $E_P = 1$

- Many difficult-to-parallelize algorithms have efficiency that approaches zero as P increases

Scalability

- Speedup describes how the parallel algorithm's performance changes with increasing P
- *Scalability* concerns the efficiency of the algorithm with changing problem size N by choosing P dependent on N so that the efficiency of the algorithm is bounded below
- **Definition**: an algorithm is *scalable* if there is minimal efficiency $\varepsilon > 0$ such that given any problem size N there is a number of processors $P(N)$ which tends to infinity as N tends to infinity, such that the efficiency $E_{P(N)} \geq \varepsilon > 0$ as N is made arbitrarily large

Amdahl's Law



- Several factors can limit the speedup
  - Processors may be idle
  - Extra computations are performed in the parallel version
  - Communication and synchronization overhead
- Let f be the fraction of the computation that is sequential and cannot be divided into concurrent tasks

Amdahl's Law

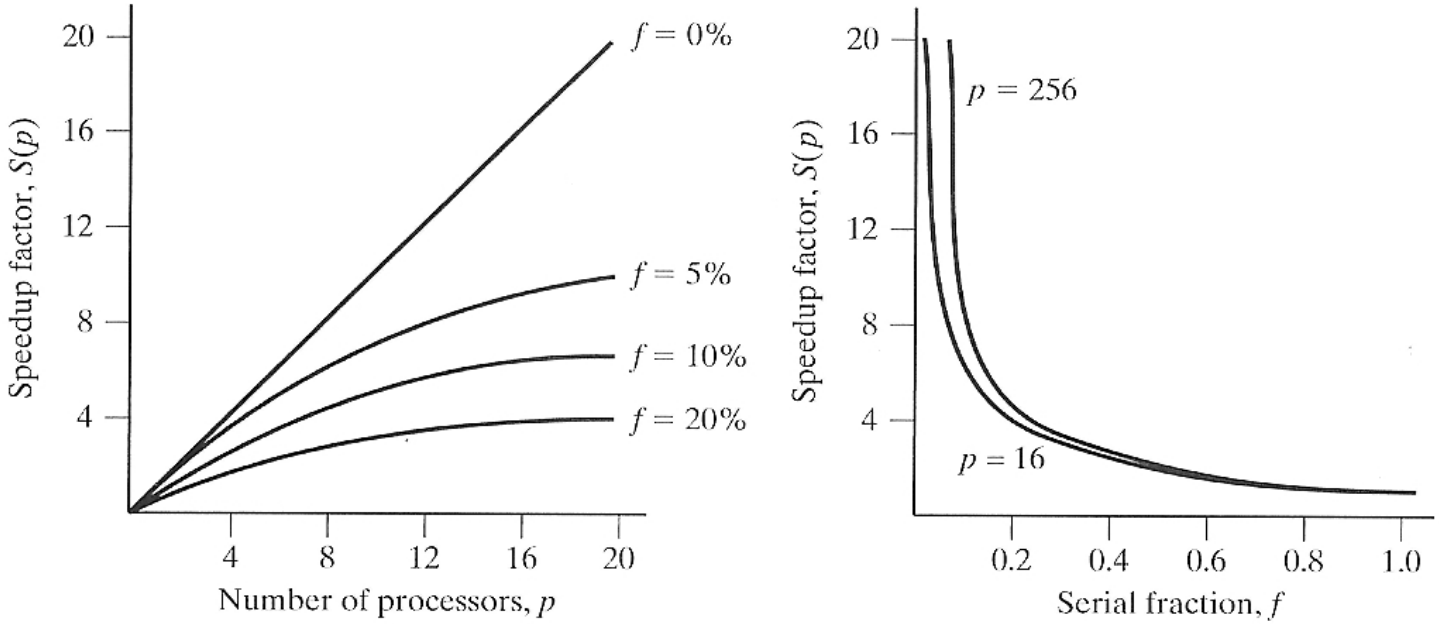
- *Amdahl's law* states that the speedup given P processors is

$$S_P = t_s / (f \times t_s + (1\text{-}f)t_s / P) = P / (1 + (P\text{-}1)f)$$

- As a consequence, the *maximum speedup* is limited by

$$S_P = f^{-1}$$

as $P \rightarrow \infty$



Gustafson's Law

- Amdahl's law is based on a fixed workload or fixed problem size

- *Gustafson's law* defines the *scaled speedup* by keeping the parallel execution time constant by adjusting P as the problem size N changes

$$S_{P,N} = P + (1\text{-}P)\alpha(N)$$

where $\alpha(N)$ is the non-parallelizable fraction of the normalized parallel time $t_{P,N} = 1$ given problem size N

- To see this, let $\beta(N) = 1\text{-}\ \alpha(N)$ be the parallelizable fraction

$$t_{P,N} = \alpha(N) + \beta(N) = 1$$

then, the scaled sequential time is

$$t_{s,N} = \alpha(N) + P\ \beta(N)$$

giving

$$S_{P,N} = \alpha(N) + P\ (1\text{-}\ \alpha(N)) = P + (1\text{-}P)\alpha(N)$$

Limitations to Speedup: Data Dependences

- The Collatz iteration loop has a *loop-carried dependence*
  - The value of *n* is carried over to the next iteration
  - Therefore, the algorithm is inherently sequential
- Loops with loop-carried dependences cannot be parallelized
- To find parallelism in an application
  - Change the loops to remove dependences (if possible!)
  - Apply algorithmic changes by rewriting the algorithm (this may change the result of the output)

Limitations to Speedup: Data Dependences

- Consider for example the update step in a *Gauss-Seidel iteration* for solving a two-point boundary-value problem:

```
do i=1,n
  soln(i)=(f(i)-soln(i+1)*offdiag(i)
                -soln(i-1)*offdiag(i-1))/diag(i)
enddo
```

- By contrast, the *Jacobi iteration* for solving a two-point boundary-value problem does not exhibit loop-carried dependences:

```
do i=1,n
  snew(i)=(f(i)-soln(i+1)*offdiag(i)
                -soln(i-1)*offdiag(i-1))/diag(i)
enddo
do i=1,n
  soln(i)=snew(i)
enddo
```

- In this case the iteration space of the loops can be partitioned and each processor given a chunk of the iteration space

Limitations to Speedup: Data Dependences

1.
```
do i=1,n
  diag(i)=(1.0/h(i))+(1.0/h(i+1))
  offdiag(i)=-(1.0/h(i+1))
enddo
```

2.
```
do i=1,n
  dxo=1.0/h(i)
  dxi=1.0/h(i+1)
  diag(i)=dxo+dxi
  offdiag(i)=-dxi
enddo
```

3.
```
dxi=1.0/h(1)
do i=1,n
  dxo=dxi
  dxi=1.0/h(i+1)
  diag(i)=dxo+dxi
  offdiag(i)=-dxi
enddo
```

- Three example loops to initialize a finite difference matrix
- Which loop(s) can be parallelized?
- Which loop probably runs more efficient on a sequential machine?

Efficient Parallel Execution

- Trying to construct a parallel version of an algorithm is not the end-all do-all of high-performance computing
  - Recall Amdahl's law: the maximum speedup is bounded by
    $$S_P = f^{-1} \text{ as } P \rightarrow \infty$$
  - Thus, efficient execution of the non-parallel fraction f is extremely important
  - We can reduce f by improving the sequential code execution (e.g. algorithm initialization parts), I/O, communication, and synchronization

- To achieve high performance, we should highly optimize the per-node sequential code and use profiling techniques to analyze the performance of our code to investigate the causes of overhead

Efficient Sequential Execution

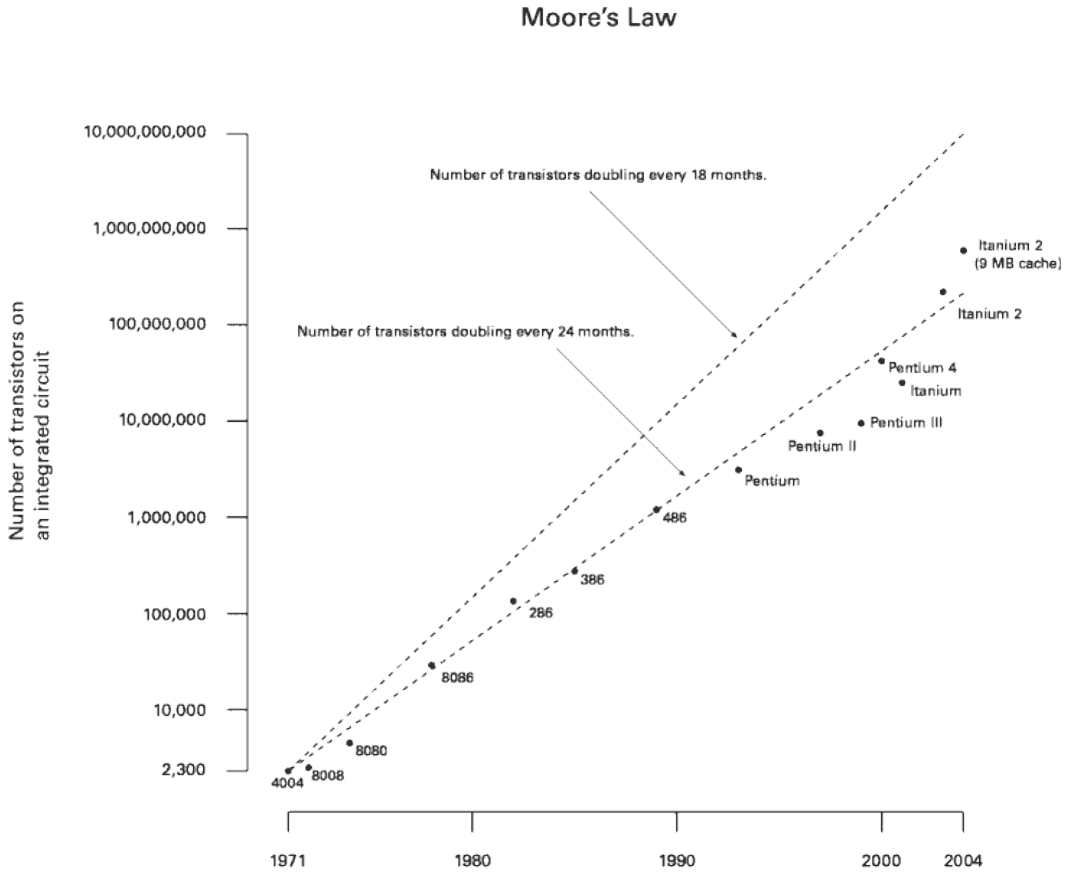
- Memory effects are the greatest concern for optimal sequential execution
  - Store-load dependences, where data has to flow through memory
  - Cache misses
  - TLB misses
  - Page faults
- CPU resource effects can limit performance
  - Limited number of floating point units
  - Unpredictable branching (if-then-else, loops, etc) in the program
- Use common sense when allocating and accessing data
- Use compiler optimizations effectively
- Execution best analyzed with performance analyzers

Lessons Learned from the Past

- Applications
  - Parallel computing can transform science and engineering and answer challenges in society
  - To port or not to port is NOT the question: a complete redesign of an application may be necessary
  - The problem is not the hardware: hardware can be significantly underutilized when software and applications are suboptimal
- Software and algorithms
  - Portability remains elusive
  - Parallelism isn't everything
  - Community acceptance is essential to the success of software
  - Good commercial software is rare at the high end

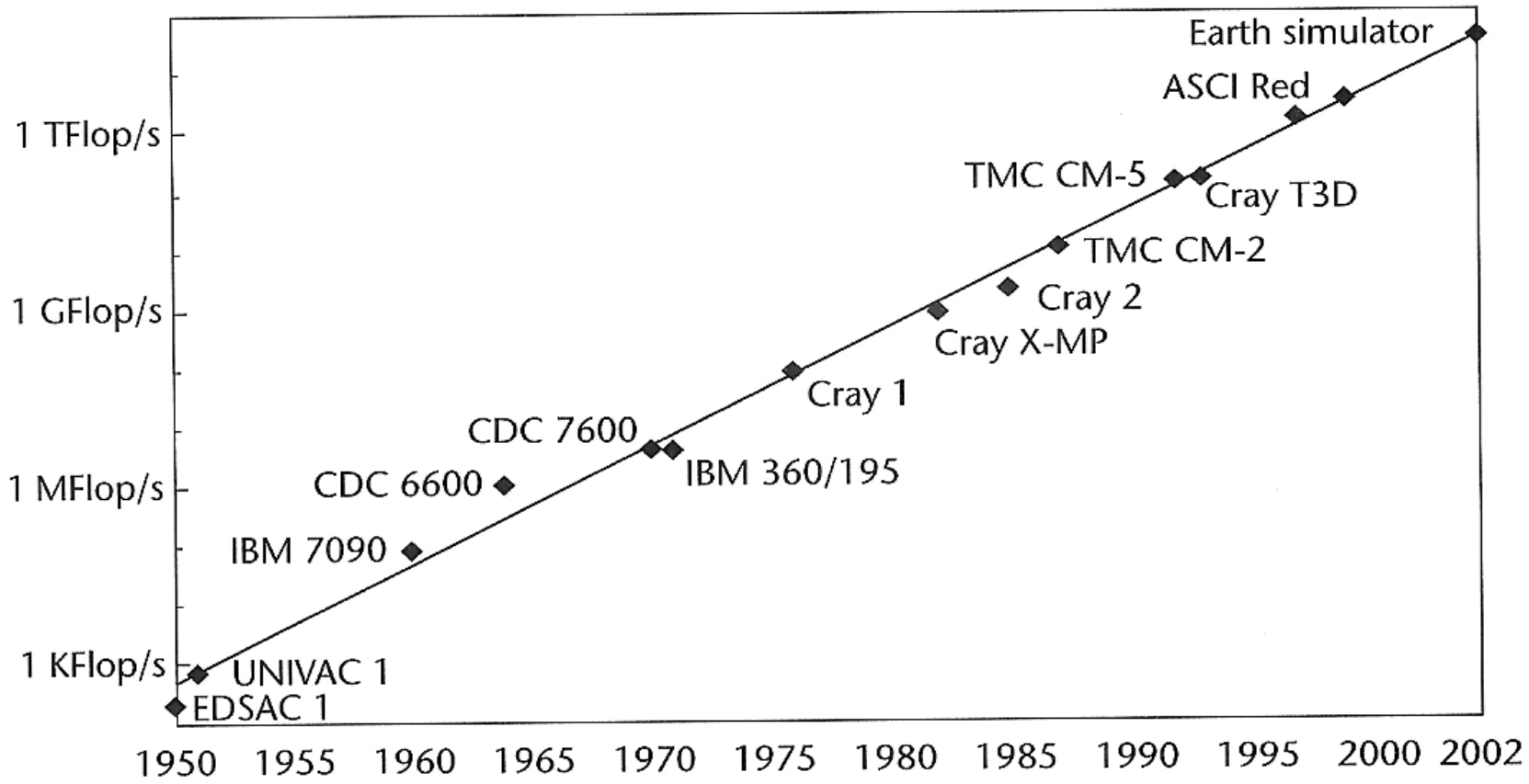
Future of Computing

- *Moore's law* tells us that we will continue to enjoy improvements of transistor cost and speed (but not CPU clock frequency!) for another decade



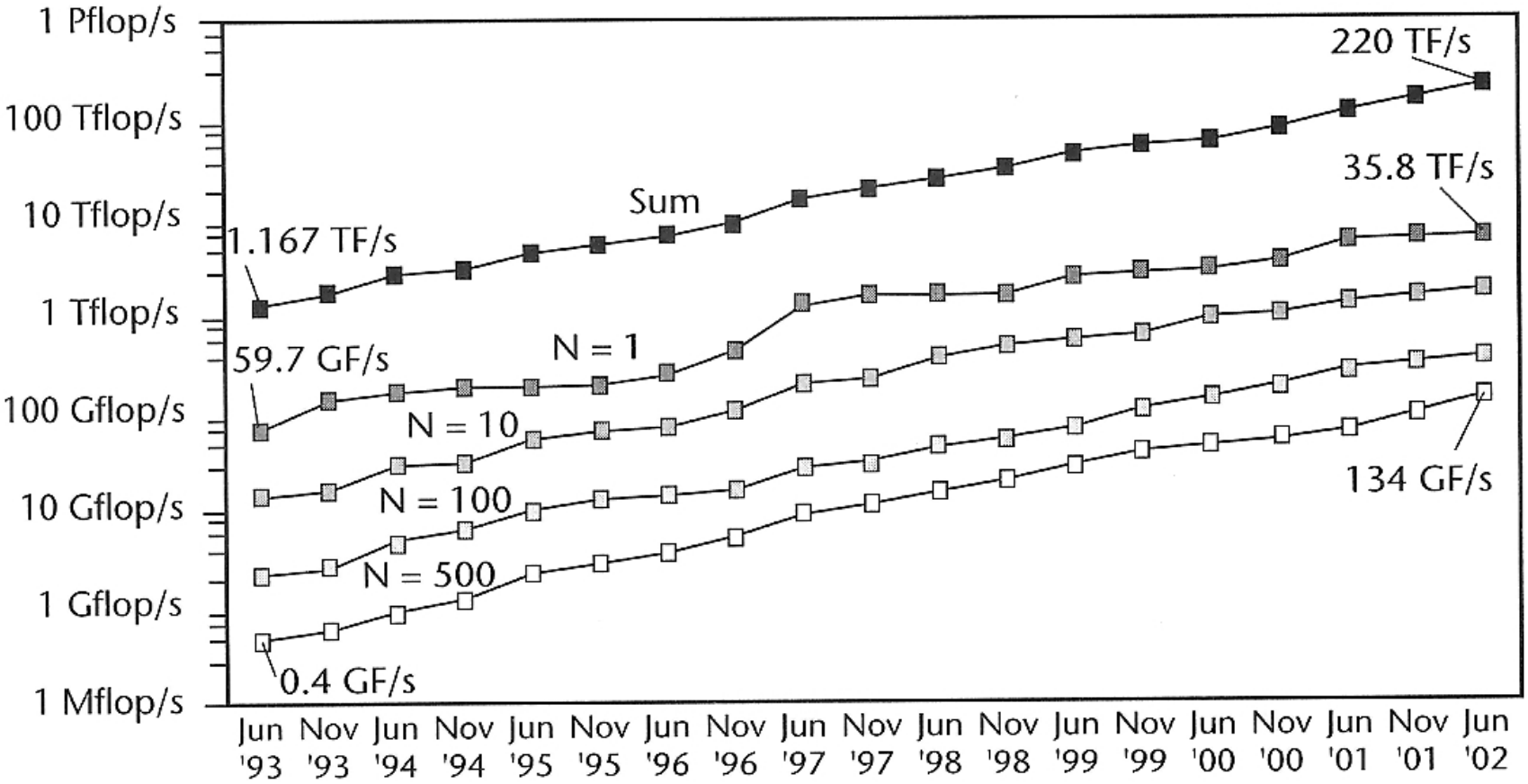
Future of Computing

- The peak performance of supercomputers follows Moore's law



Future of Computing

- Performance growth at fixed Top500 rankings



Future of Computing

- With increased transistor density we face huge CPU energy consumption and heat dissipation issues
  - This puts fundamental limits on CPU clock frequencies
  - Therefore, single CPU performance will be relatively flat
- This will mean that
  - Computers will get a lot cheaper but not faster
  - On-chip parallelism will increase with multiple cores to sustain continued performance improvement
- High-performance computing power will be available on the desktop, requiring parallel algorithms to utilize the full potential of these machines

Writing Efficient Programs

- How to program *multiprocessor* systems that employ multiple processors (often with multiple memory banks)
  - Understand **the problem** to be solved
  - Understand the **machine architecture constraints**
  - **Redesign** the algorithm when needed
  - **Partition the data** when applicable
  - Use **parallel programming languages**
  - … or programming language extensions to support parallelism
  - **Debugging** is much more complicated
  - **Performance analysis** is no longer optional

Further Reading

- [PP2] pages 3-12
- [SRC] pages 3-13
- [SPC] pages 11-15, 37-45, 48-52, 110-112
- Optional:
  - More on Moore's law
    - http://en.wikipedia.org/wiki/Moore%27s_law
  - Grand Challenge problems
    - http://en.wikipedia.org/wiki/Grand_Challenge_problem
  - Collatz conjecture and implementation on the Cell BE:
    - http://en.wikipedia.org/wiki/Collatz_conjecture http://www.ibm.com/developerworks/library/pa-tacklecell3/