“If you fail to plan, you are planning to fail!”

Benjamin Franklin, mid-eighteenth century

Lecture A.5:
Designing Parallel Programs

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2021

Lectures developed by Dr. Ignacio M. Llorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

- A.1. Parallel Processing Architectures
- A.2. Large-scale Processing on the Cloud
- A.3. Practical Aspects of Cloud Computing
- A.4. Application Parallelism
- A.5. Designing Parallel Programs

B. Parallel Computing

C. Parallel Data Processing

Wrap-Up: Advanced Topics

CS205: Contents

APPLICATION SOFTWARE

APPLICATION PARALLELISM

PARALLEL PROGRAM DESIGN

PROGRAMMING MODEL

- Optimization
- OpenACC
- OpenMP
- MPI

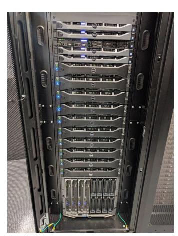
- Spark
- Map-Reduce

PLATFORM

B. BIG COMPUTE

C. BIG DATA

amazon web services



Open Nebula

FAS RC

CLOUD COMPUTING

FAS RC CANNON

HARVARD'S LARGEST CLUSTER

PARALLEL ARCHITECTURES

First Think then Code!

Context

Designing Parallel Programs

1. Sequential Version
2. Parallelization Overheads
3. Numerical Complexity
4. Efficiency and Scalability

Roadmap

Designing Parallel Programs

Code Analysis

Parallelization Overheads

Numerical Complexity

Efficiency and Scalability

Code Analysis

Understand the Program and the Problem

The first step in developing parallel software is to understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also

PARALLEL VERSION

- Develop a parallel implementation of an existing serial code
- Fine grain / compiler or directive-based parallelization
- Easier approach and faster to develop

NEW PARALLEL CODE

- Develop a completely new code from scratch
- Coarse grain / domain decomposition parallelization
- Takes longer, but better performance

CODE ANALYSIS

Code Analysis

Execution Time Components

EXECUTION_TIME = <u>CPU_TIME</u> + I/O_TIME + SYSTEM_TIME

↓

POTENTIALLY PARALLEL_TIME SECTION

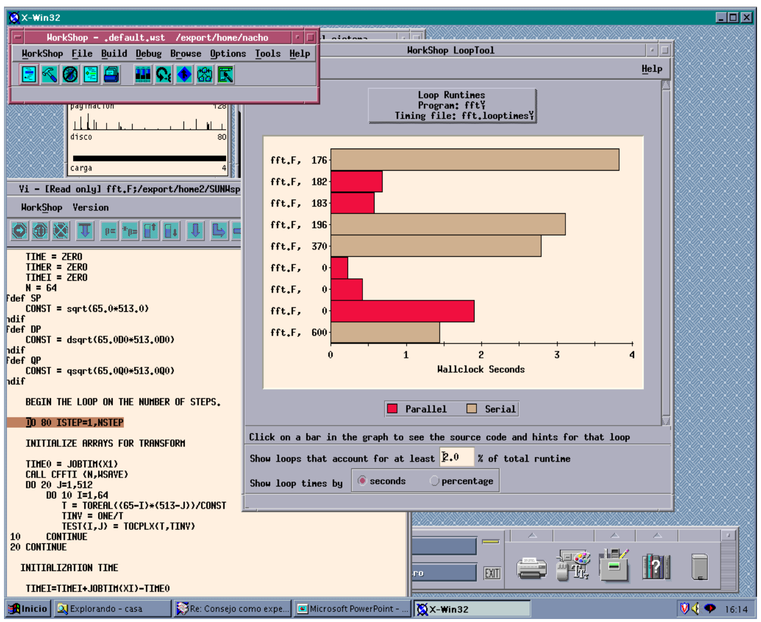
Code Analysis

Code Profiling

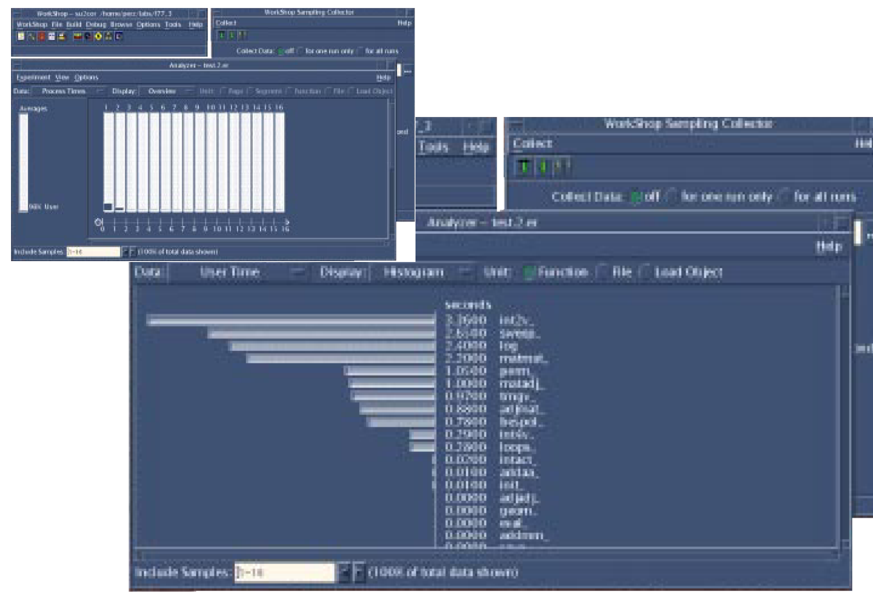
CLI Tools

gprof, tconv, dtime, etime, ...

GUI Tools



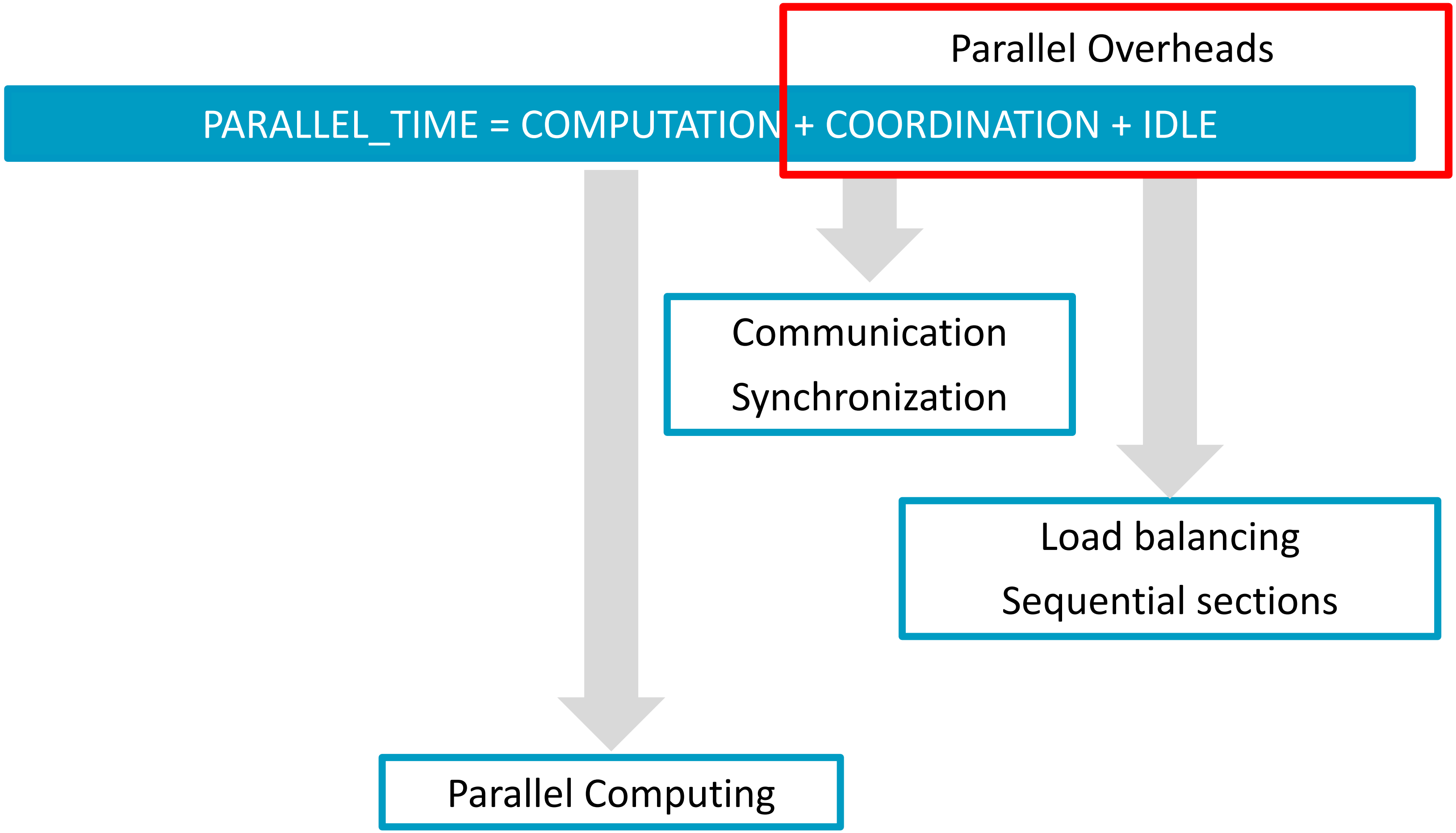
Looptool (solaris)



cvd (SGI)

Parallelization Overheads

Inefficiencies in Parallel Processing



Parallelization Overheads

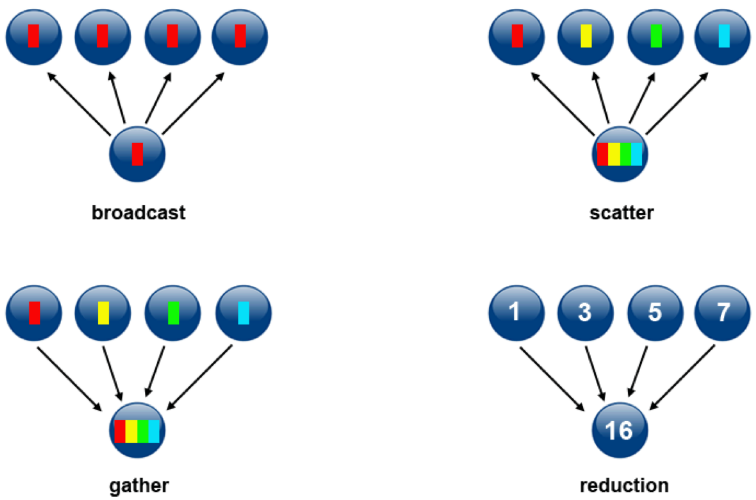
Communication

Types of Communication

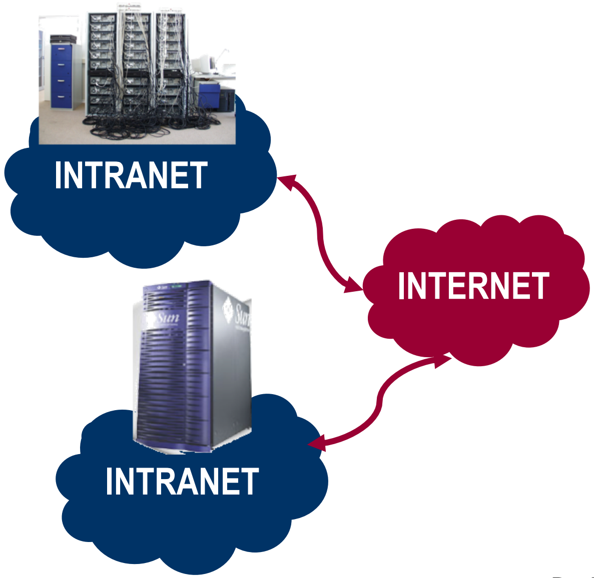
- Memory sharing (implicit): Access to a shared memory space
- Message passing (explicit): Point-to-point, vector reductions, broadcasts, global collective operations (all-to-all operations, gather, scatter…)…

Scales of Communication

- Internal: Within a core (in-cache), a chip (between caches) and a machine (across sockets)
- External: Within a switch, across switches within a DC, and across internet between DCs



Source: https://computing.llnl.gov/tutorials/parallel_comp



Parallelization Overheads

Minimizing Communication Overhead

Overlapping with Computation

- Memory sharing: Overlap memory requests with other instructions if there is enough work to do
- Message passing: Send a message and do computation while the message is being sent or initiate a recv, do work and then poll to see if it is done

Latency vs. Bandwidth

- Latency: Time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- Bandwidth: Amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.

Parallelization Overheads

Synchronization

Synchronization

- Managing the sequence of work and the tasks performing it
- It is a critical design consideration for most parallel programs

Types of Synchronization

- **Memory sharing** (explicit): Mutual exclusion (locks, mutexes, monitors, ...), consensus (barriers...) and conditions (flags, condition variables, signals...)
- **Message passing** (explicit): Global synchronization (barriers, scalar reductions, ...) and broadcasts with small signals

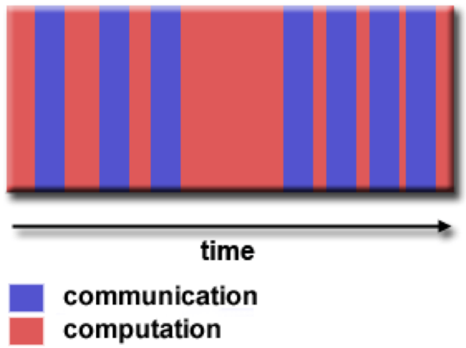
Parallelization Overheads

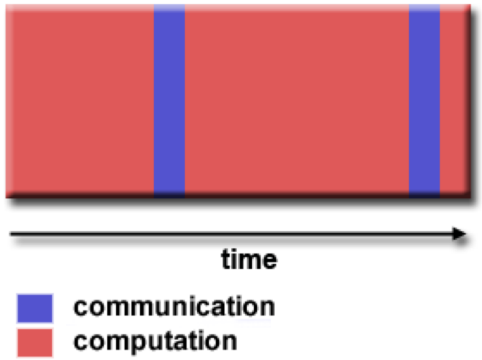
Granularity

Computation to Communication Ratio

- Periods of computation are typically separated from periods of communication by synchronization events.
- Qualitative measure of the computation grain, usually as the ratio of computation to communication based on data and machine sizes.

Fine-Grained	Coarse-Grained
Relatively small amounts of computational work are done between communication events	Relatively large amounts of computational work are done between communication/synchronization events
Low computation to communication ratio	High computation to communication ratio





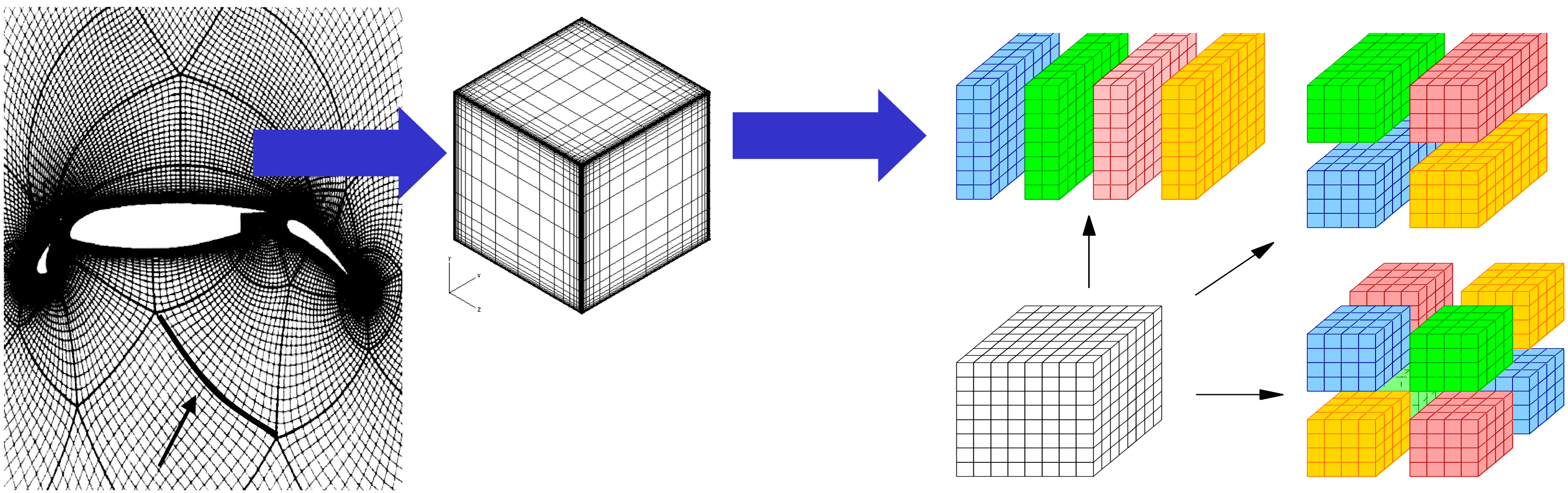
Source: https://computing.llnl.gov/tutorials/parallel_comp

Parallelization Overheads

Granularity

Example:

- Numerical resolution of PDE using an explicit discretization method

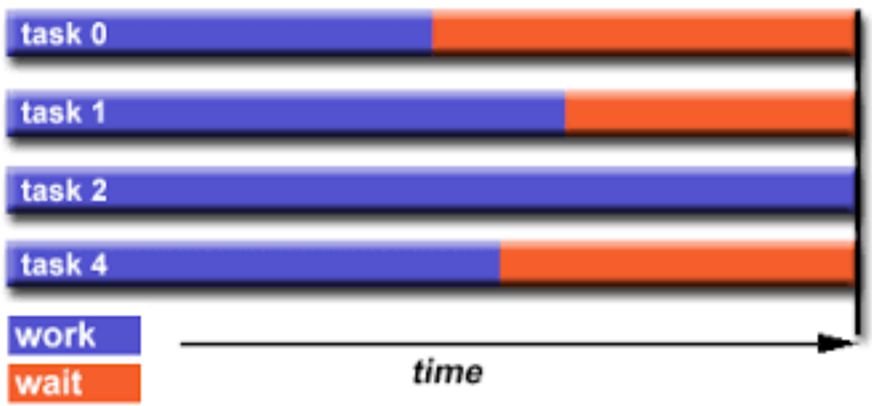


	1D Parallelization	2D Parallelization
Computation	$n/p*n^2$	n^3/p
Communication	n^2	$n^2/p^{1/2}$
Granularity	n/p	$n/p^{1/2}$

Parallelization Overheads

Load Balancing

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time
- It can be considered a minimization of task idle time



Source: https://computing.llnl.gov/tutorials/parallel_comp

Parallelization Overheads

Data Dependencies (Sequential)

- A dependence exists between program statements when the order of statement execution affects the results of the program
- A data dependence results from multiple use of the same location(s) in storage by different tasks
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism

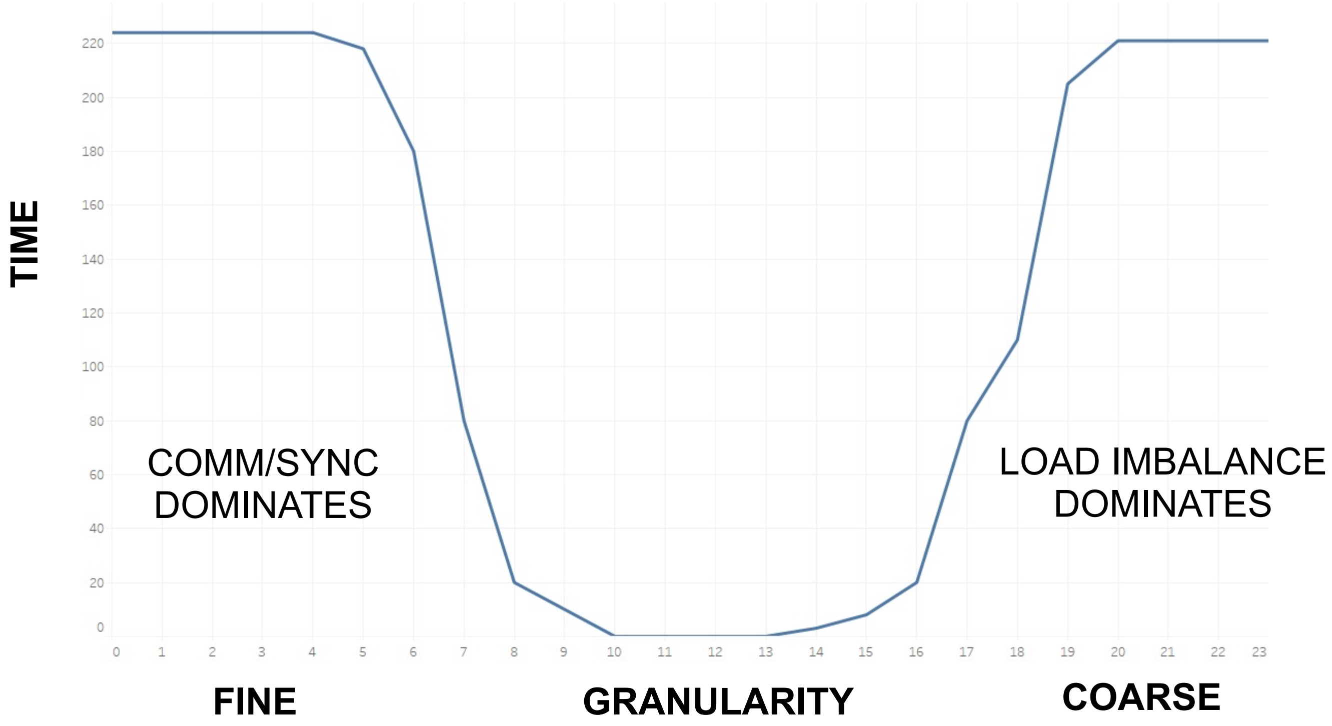
```
DO I = 2,N
       A(I) = B(I) - A(I-1)
END DO
```

Parallelization Overheads

Interrelation Between the Different Overheads

OVERHEAD = COMM + SYNC + LOAD IMBALANCE

Graph of execution time using p processors

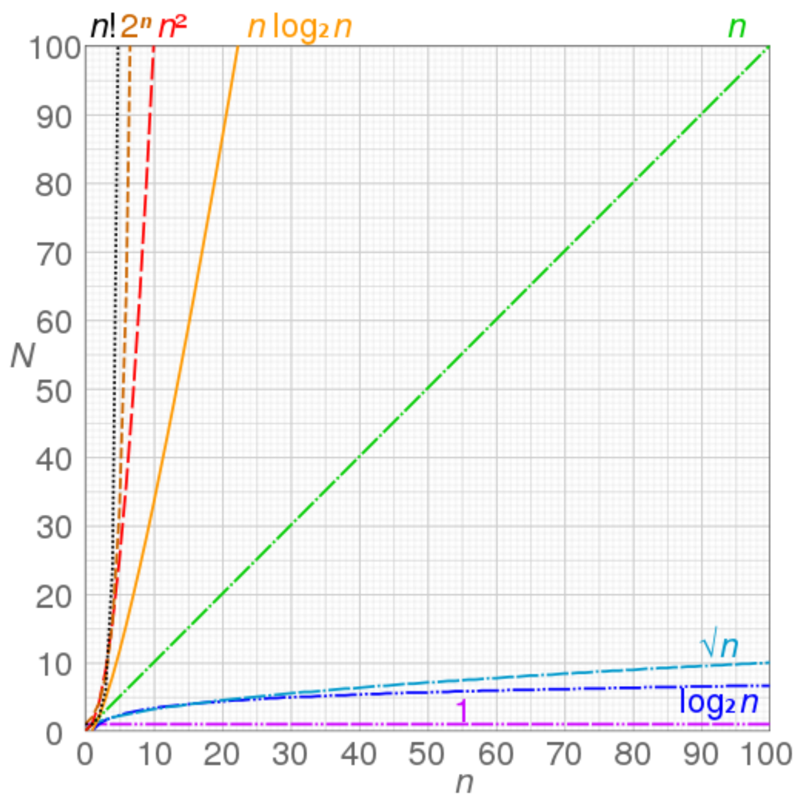


Numerical Complexity

Time Complexity

- How fast or slow an algorithm performs
- Numerical function that depends on the data size of the problem

Type	Complexity
Constant	O(1)
Linear	O(n)
Logarithmic	O(log(n))
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$2^{O(n)}$



Numerical Complexity

Time Complexity

Example: N-body Problem

P	$O(N^2)$ MOLMEC 7,000	$O(NlogN)$ MEGADYN 550,000
1	8152 sec	
2	4481 sec	6305 sec
3	3956 sec	
4	2427 sec	3295 sec
6	1769 sec	
8		1849 sec

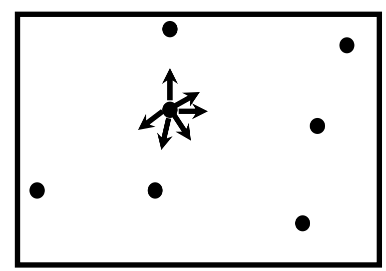
FMM (Fast Multipole)

Greengard, Rokhlin

Separate short & long range forces:

• Short-range forces are updated in each time step

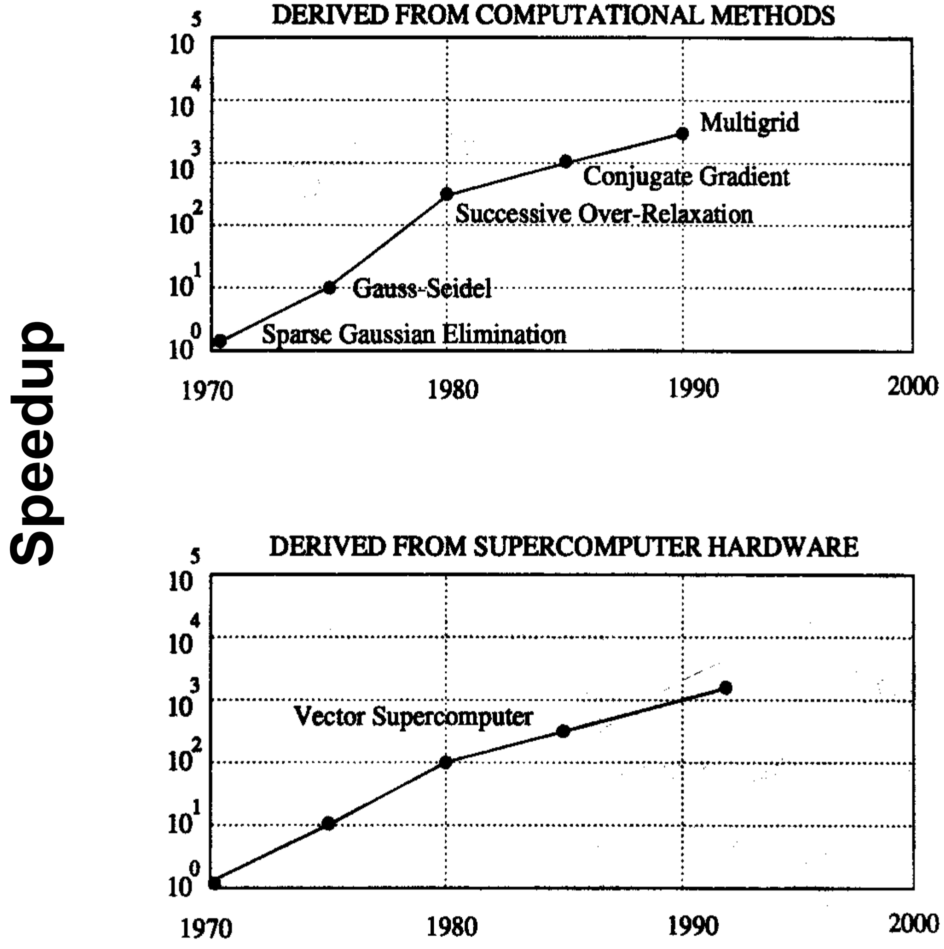
• Long-range forces are treated on "coarser scales"



- Both exhibit similar speed-up
- 550,000 particles would require 18,000 processors with MOLMEC

Numerical Complexity

Algorithms vs. Computer Improvements



Algorithm	Complexity
GE	$O(n^2)$
GS	$O(n^2 \log(n))$
SOR	$O(n^{3/2} \log(n))$
CG	$O(n^{3/2} \log(n))$
MG	$O(n \log(n))$
Full MG	$O(n)$

Grand Challenge: High Performance Computing and Communications (NSF) [1992]

Efficiency and Scalability

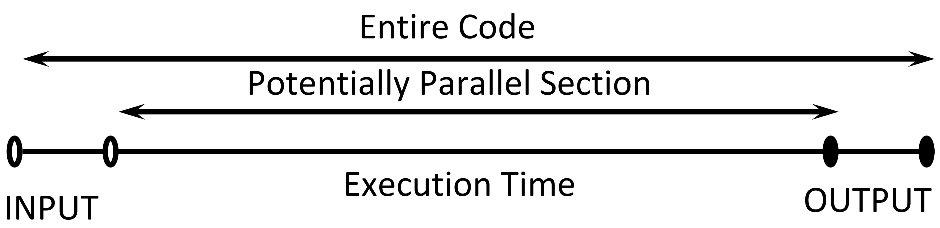
Speed-up

Parallel execution Speed-up and Efficiency for a given problem size and a number of processors

$$S(n,p) = \frac{T(n,1)}{T(n,p)} \qquad E(n,p) = \frac{S(n,p)}{p}$$

Theoretical Speed-up

- $S_T(n,p)$ only considers overheads due to sequential parts



Parallel Fraction of Code

$$c = \frac{T_{\text{parallel section}}}{T_{\text{entire code}}}$$

$$S_T(n,p) = \frac{T(n,1)}{T(n,p)} = \frac{1}{(1-c) + c/p}$$

$$c = 1 \Rightarrow S_T(n,p) = p \quad \text{(linear speed up)}$$

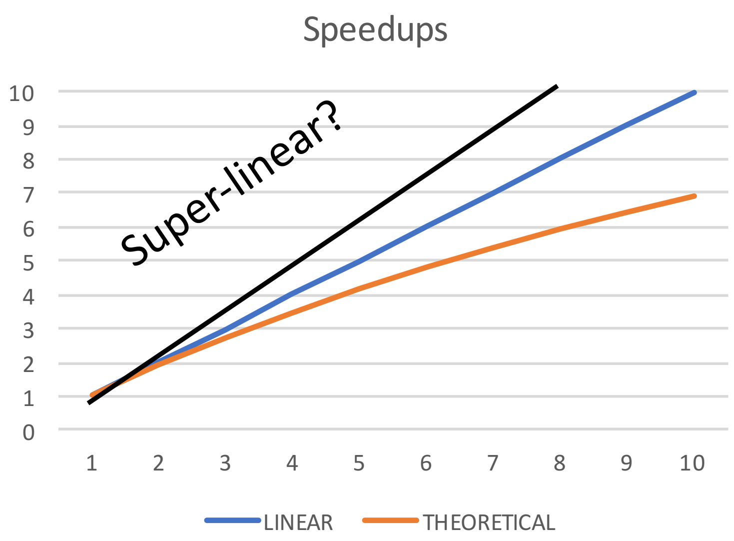
Efficiency and Scalability

Speed-up

Example (fixed n): c=0.95

$$S_T(n,p) = \frac{1}{0.05 + 0.95/p}$$

SPEEDUP		
p	LINEAR	THEORETICAL
1	1	1.0
2	2	1.9
3	3	2.7
4	4	3.5
5	5	4.2
6	6	4.8
7	7	5.4
8	8	5.9
9	9	6.4
10	10	6.9



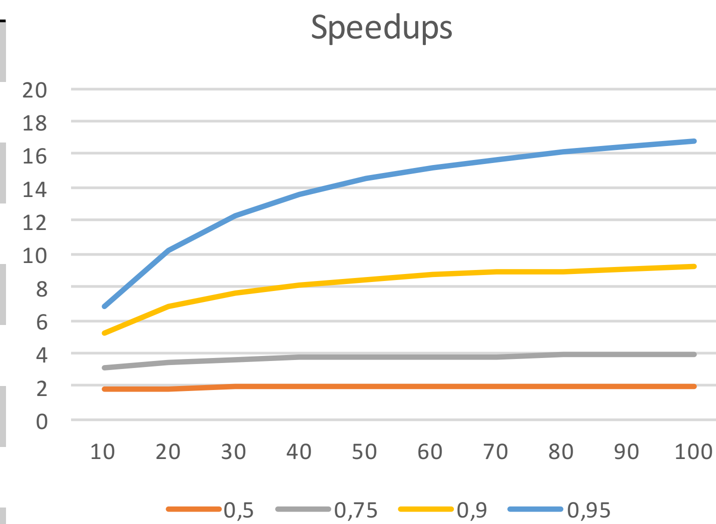
Efficiency and Scalability

Amdahl Law (1967)

- Amdahl's Law states that potential program speedup is defined by the fraction of code (c) that can be parallelized
- Speedup is limited by sequential code, even a small percentage of sequential code can greatly limit potential speedup

SPEEDUPS FOR DIFFERENT C's

p	0.5	0.75	0.9	0.95
10	1.8	3.1	5.3	6.9
20	1.9	3.5	6.9	10.3
30	1.9	3.6	7.7	12.2
40	2.0	3.7	8.2	13.6
50	2.0	3.8	8.5	14.5
60	2.0	3.8	8.7	15.2
70	2.0	3.8	8.9	15.7
80	2.0	3.9	9.0	16.2
90	2.0	3.9	9.1	16.5
100	2.0	3.9	9.2	16.8



$$\text{Asymptotic } S_T : \lim_{p\to\infty} S_T = \frac{1}{1-c}$$

Efficiency and Scalability

Speed-up

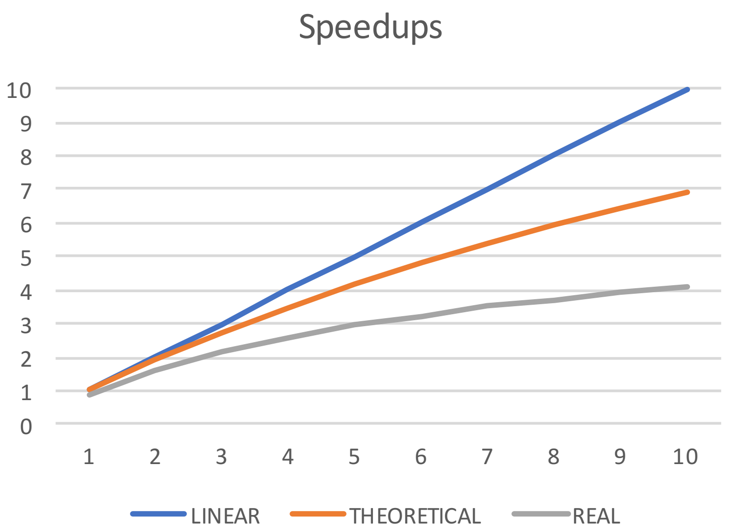
In reality, the situation is even worse than predicted by Amdahl's Law due to the parallelization overheads

Real Speed-up $$S_R(n,p) = \frac{1}{0.05 + 0.95/p + 0.1}$$

SPEEDUP

p	LINEAR	THEORETICAL	REAL
1	1	1.0	0.9
2	2	1.9	1.6
3	3	2.7	2.1
4	4	3.5	2.6
5	5	4.2	2.9
6	6	4.8	3.2
7	7	5.4	3.5
8	8	5.9	3.7
9	9	6.4	3.9
10	10	6.9	4.1

OVERHEAD = COMM + SYNC + LOAD IMBALANCE



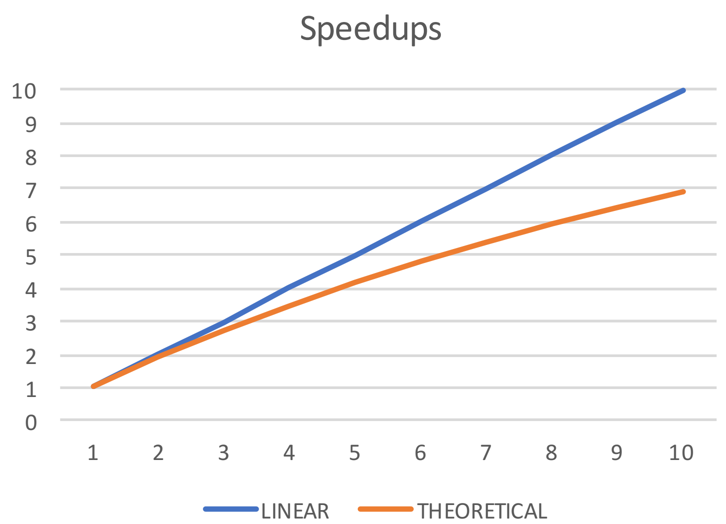
Efficiency and Scalability

Gustafson Law (1988)

- Amdahl's law keeps the problem size fixed.
- Larger systems should be used to solve larger problems. ideally there should be a fixed amount of parallel work per processor. (SCALED PROBLEM SIZE)

$$S'_T(n,p) = 1 - c + cp$$

SPEEDUP		
p	LINEAR	THEORETICAL
1	1	1.0
2	2	2.0
3	3	2.9
4	4	3.9
5	5	4.8
6	6	5.8
7	7	6.7
8	8	7.7
9	9	8.6
10	10	9.6



Breakout Room

Try to derive Gustafson's Law

Hints:

- Decompose the workload for a constant time into parallel and serial parts. This will involve the parameter c.
- Next, increase the number of processors to p. How does this affect the parallel workload?
- Finally, put the modified parallel workload back into the speedup.

Don't worry if you don't get it right away! The goal is to start thinking it through.

Efficiency and Scalability

Scalability

The Program should scale up to use a large number of processors – But what does that really mean?

FIXED PROBLEM SIZE
(strong scaling)

- Problem size stays fixed as more processors are added.
- Goal: Run same problem faster.
    - Reduce execution time.
- Perfect scaling: Problem solved in 1/p time.
    - Another way of seeing this is: S=p with n constant

FIXED SIZE PER PROCESSOR
(weak scaling)

- Problem size *per processor* stays the same as more processors are added.
- Goal: Run larger problem in same amount of time.
- Perfect scaling is S=p with n/p constant.

Efficiency and Scalability

Strong vs. Weak Scaling

Strong Scaling

- Speed-up on the same size problem
- Perfect strong scaling: Speedup of p on p processors
- Typically, small data but computationally intense
- At some point it breaks down

Weak Scaling

- Problem grows “proportionally” to processors
- What does proportionally mean (for example NxN matrix multiply)?
    - 2N x 2N - double N
    - 1.4N x 1.4N - double entries
    - 1.26N x 1.26N - double operations

Efficiency and Scalability

Scalability

ISOEFFICIENCY

What is the rate at which the problem size must increase with p to keep E(n,p) fixed?

A parallel algorithm is called scalable if E(n,p) can be kept constant by increasing the problem size as n grows.

This rate determines the scalability of the system. The slower this rate, the better.

I.M. Llorente et al. / Parallel Computing 22 (1996) 1169–1195

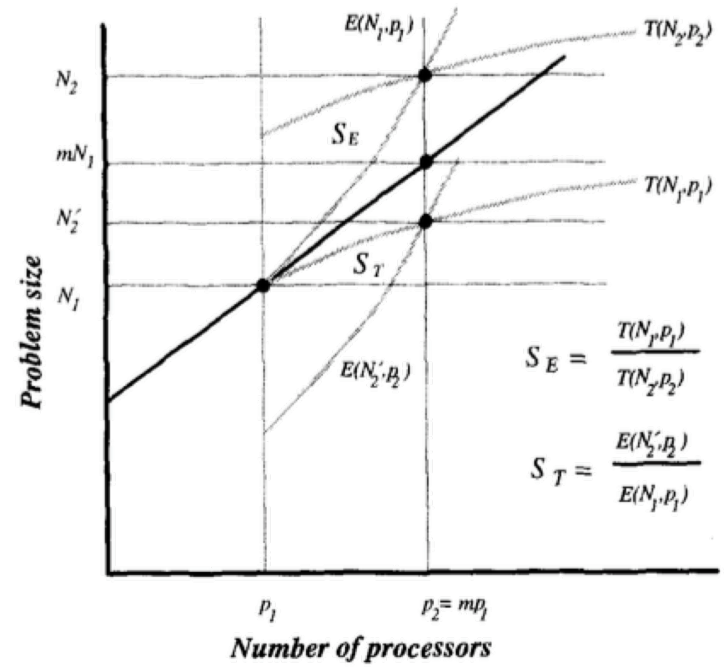


Fig. 8. Isoefficiency and isotime scalability metrics.

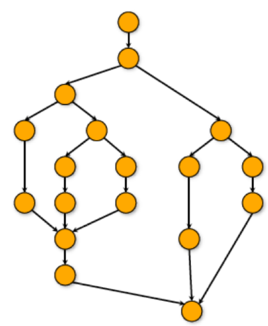
Efficiency and Scalability

Work Span

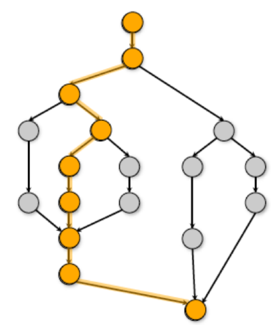
COMPUTATIONS REPRESENTED AS A GRAPH OF DEPENDENCIES

Amdahl is too simple, only talks about serial nodes

WORK = All Computations
Proportional to T_s
(time to run on single node)



SPAN= Critical Path Compute
Proportional to T_∞
(time to run on infinite nodes)



UPPER BOUNDS ON SPEEDUP
Speedup <= p
Speedup <= T_s/T_∞

Reading Assignments / Open Discussion

Relations between Efficiency and Executing Time at Scaling

I. M. Llorente, F. Tirado, L. Vázquez
"Some aspects about the scalability of scientific applications on parallel architectures" Parallel Computing, 1996, Vol.22(9), pp.1169-1195

What is isomemory scaling?

What is isotime scaling?

What is isoefficiency scaling?

What is naive scaling?

What is realistic scaling?

Next Steps

- HWA due on Tuesday!
  Linpack compilation (Performance Competition!)

- Get ready for next **lecture** (Part B!):
  B.1. Foundations of Parallel Computing

- Get ready for first **hands-on:**
  H1. Python Multiprocessing

- **Reading assignments**:

Questions

Designing Parallel Programs