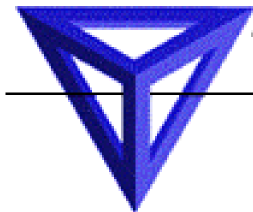
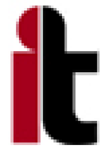
Parallel & Cluster Computing

Clusters & Distributed Parallelism

National Computational Science Institute
August 8-14 2004

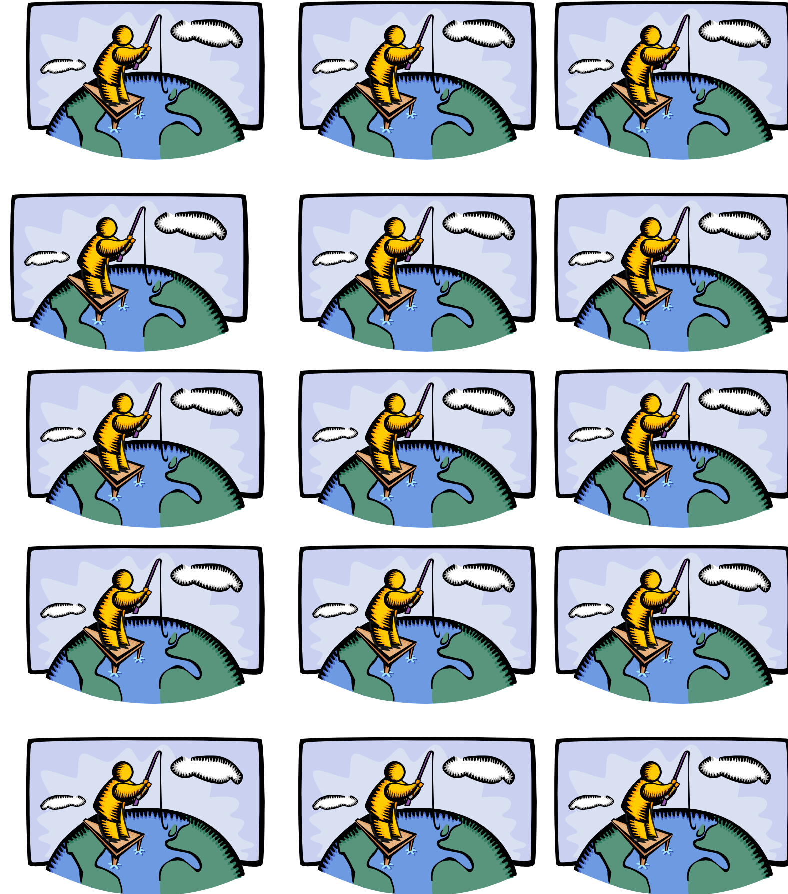
Paul Gray, University of Northern Iowa
David Joiner, Shodor Education Foundation
Tom Murphy, Contra Costa College
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College

Parallelism

Parallelism means doing multiple things at the same time: you can get more work done in the same time.

Less fish …



More fish!

What Is Parallel Computing?

Parallel computing is the use of multiple processors to solve a problem, and in particular the use of multiple processors operating **concurrently** on different parts of a problem.

The different parts could be different tasks, or the same task on different pieces of the problem's data.

Kinds of Parallelism

- Shared Memory Multithreading
- Distributed Memory Multiprocessing ← this workshop!
- Hybrid Shared/Distributed Parallelism

Why Parallelism Is Good

- The Trees: We like parallelism because, as the number of processors working on a problem grows, we can solve the same problem in less time.
- The Forest: We like parallelism because, as the number of processors working on a problem grows, we can solve bigger problems.

Parallelism Jargon

- Threads: execution sequences that share a single memory area ("address space") and can see each other's memory
- Processes: execution sequences with their own independent, private memory areas ← this workshop!

… and thus:

- Multithreading: parallelism via multiple threads
- Multiprocessing: parallelism via multiple processes

As a general rule, Shared Memory Parallelism is concerned with threads, and Distributed Parallelism (this workshop!) is concerned with processes.

Jargon Alert

In principle:

- "shared memory parallelism" ➔ "multithreading"
- "distributed parallelism" ➔ "multiprocessing"

In practice, these terms are often used interchangeably:

- Parallelism
- Concurrency (not as popular these days)
- Multithreading
- Multiprocessing

Typically, you have to figure out what is meant based on the context.

Distributed Multiprocessing: The Desert Islands Analogy

An Island Hut

Imagine you’re on an island in a little hut.

Inside the hut is a desk.

On the desk is a phone, a pencil, a calculator, a piece of paper with numbers, and a piece of paper with instructions.

Instructions

The instructions are split into two kinds:

- Arithmetic/Logical: e.g.,
  - Add the 27th number to the 239th number
  - Compare the 96th number to the 118th number to see whether they are equal
- Communication: e.g.,
  - dial 555-0127 and leave a voicemail containing the 962nd number
  - call your voicemail box and collect a voicemail from 555-0063 and put that number in the 715th slot

Is There Anybody Out There?

If you’re in a hut on an island, you aren’t specifically aware of anyone else.

Especially, you don’t know whether anyone else is working on the same problem as you are, and you don’t know who’s at the other end of the phone line.

All you know is what to do with the voicemails you get, and what phone numbers to send voicemails to.

Someone Might Be Out There

Now suppose that Dave is on another island somewhere, in the same kind of hut, with the same kind of equipment.

Suppose that he has the same list of instructions as you, but a different set of numbers (both data and phone numbers).

Like you, he doesn't know whether there's anyone else working on his problem.

Now suppose that Paul and Tom are also in huts on islands.

Suppose that each of the four has the exact same list of instructions, but different lists of numbers.

And suppose that the phone numbers that people call are each others'. That is, your instructions have you call Dave, Paul and Tom, Dave's has him call Paul, Tom and you, and so on.

Then you might all be working together on the same problem.

All Data Are Private

Notice that you can't see Dave's or Paul's or Tom's numbers, nor can they see yours or each other's.

Thus, everyone's numbers are private: there's no way for anyone to share numbers, except by leaving them in voicemails.

Long Distance Calls: 2 Costs

When you make a long distance phone call, you typically have to pay two costs:

- Connection charge: the fixed cost of connecting your phone to someone else's, even if you're only connected for a second
- Per-minute charge: the cost per minute of talking, once you're connected

If the connection charge is large, then you want to make as few calls as possible.

Like Desert Islands

Distributed parallelism is very much like the Desert Islands analogy:

- Processors are independent of each other.
- All data are private.
- Processes communicate by passing messages (like voicemails).
- The cost of passing a message is split into:
  - latency (connection time: time to send a message of length 0 bytes);
  - bandwidth (time per byte).

What is a Cluster?

What is a Cluster?

"… [W]hat a ship is … It's not just a keel and hull and a deck and sails. That's what a ship needs. But what a ship is ... is freedom."

– Captain Jack Sparrow

"Pirates of the Caribbean"

[1]

What a Cluster is ….

A cluster needs of a collection of small computers, called nodes, hooked together by an interconnection network (or interconnect for short).

It also needs software that allows the nodes to communicate over the interconnect.

But what a cluster is … is the ***relationships*** among these components.

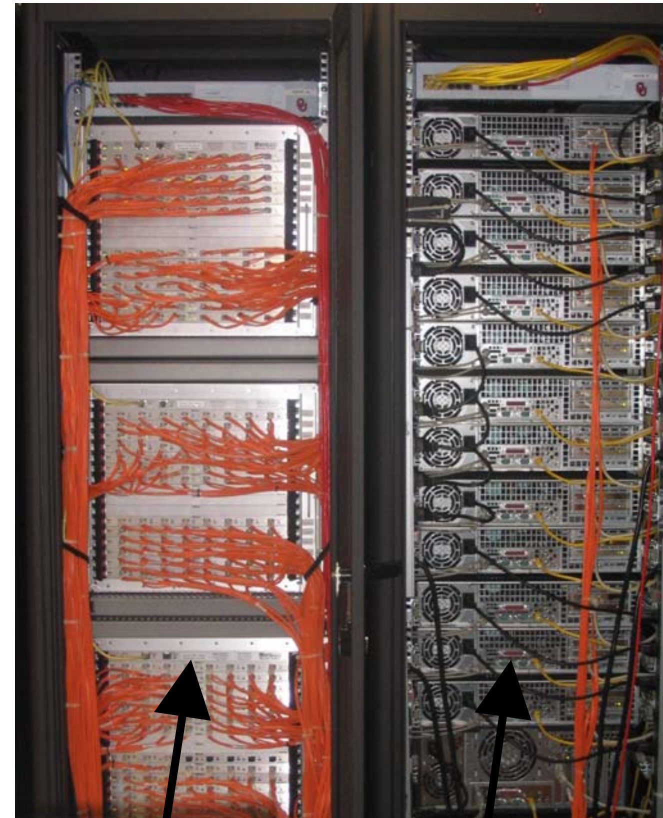
What's in a Cluster?

These days, many clusters are made of nodes that are basically Linux PCs, made out of the same hardware components as you'd find in your own PC.

As for interconnects, it's quite common for clusters these days to use regular old Ethernet (typically Gigabit), but there are also several high performance interconnects on the market:

- Dolphin Wulfkit (SCI)
- Infiniband
- Myrinet
- Quadrics
- 10 Gigabit Ethernet

An Actual Cluster



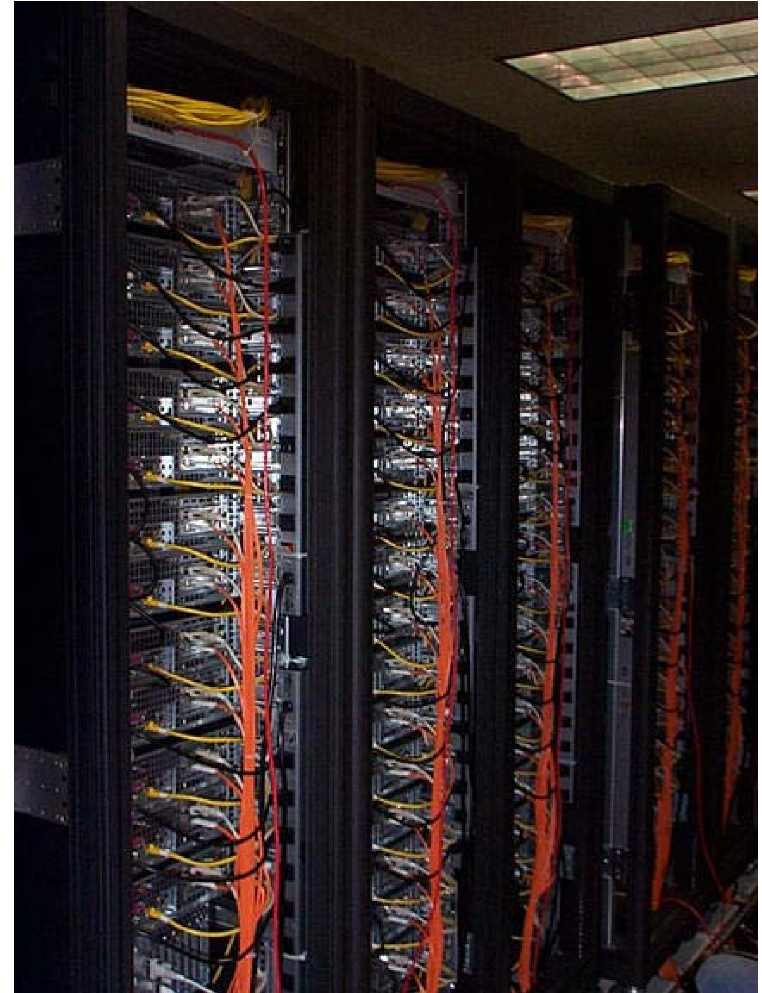
Interconnect

Nodes

`boomer.oscer.ou.edu`

Details about boomer

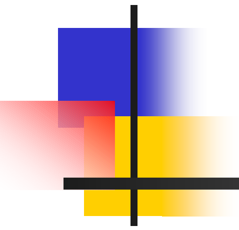
270 Pentium4 Xeon 2 GHz CPUs
270 GB RAM
8700 GB disk
OS: Red Hat Linux Enterprise 3.0
Peak speed: 1.08 TFLOP/s*
Programming model:
distributed multiprocessing



*TFLOP/s: trillion floating point calculations per second

More boomer Details

- Each node has
  - 2 Pentium4 XeonDP CPUs (2.0 GHz)
  - 2 GB RAM
  - 1 or more hard drives (mostly EIDE, a few SCSI)
  - 100 Mbps Ethernet (cheap backup interconnect)
  - Myrinet2000 (high performance interconnect)
- Breakdown of nodes
  - 135 compute nodes
  - 8 storage nodes
  - 2 “head” nodes (where you log in, compile, etc)
  - 1 management node (batch system, LDAP, etc)



Parallelism Issues

Load Balancing

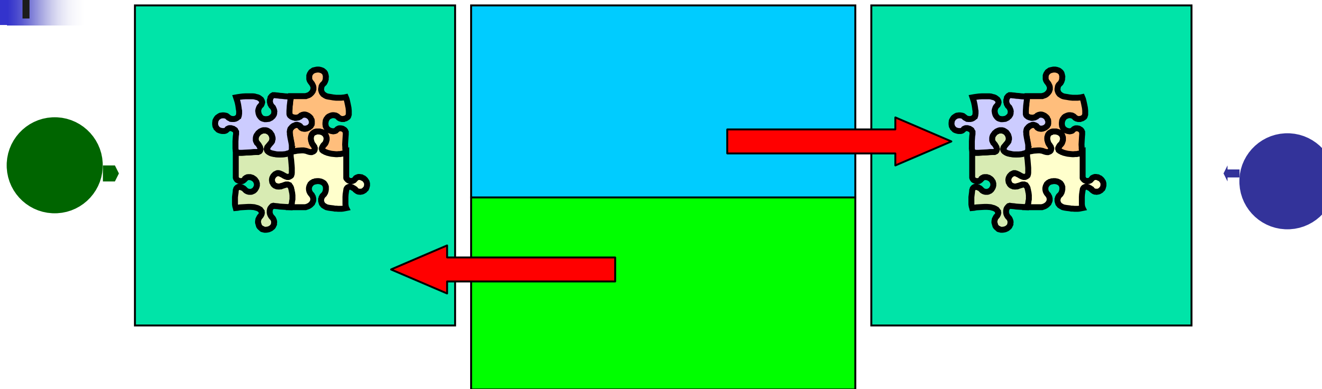
Suppose you have a distributed parallel code, but one processor does 90% of the work, and all the other processors share 10% of the work.

Is it a big win to run on 1000 processors?

Now suppose that each processor gets exactly $1/N_p$ of the work, where N_p is the number of processors.

Now is it a big win to run on 1000 processors?

Load Balancing

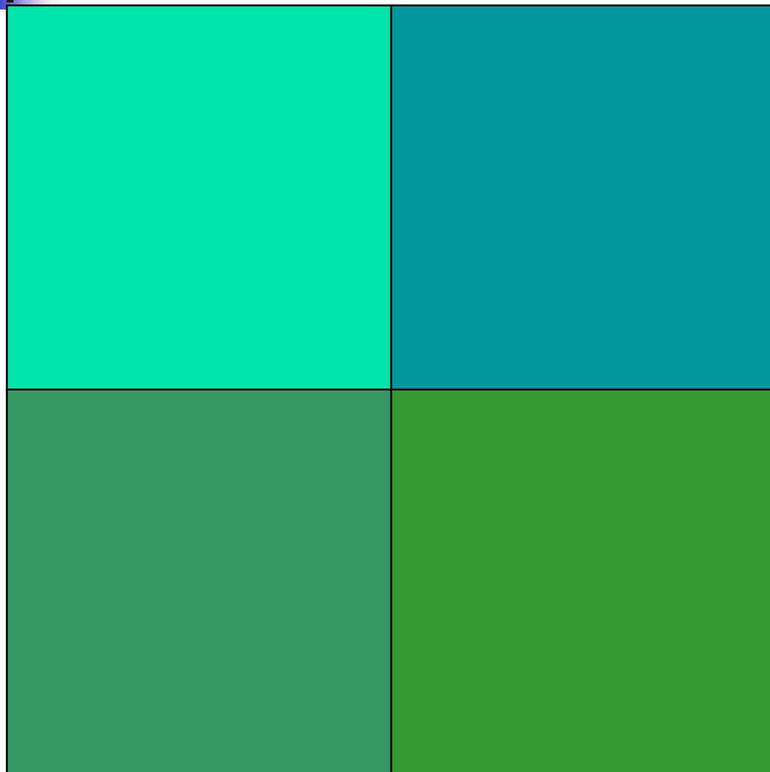


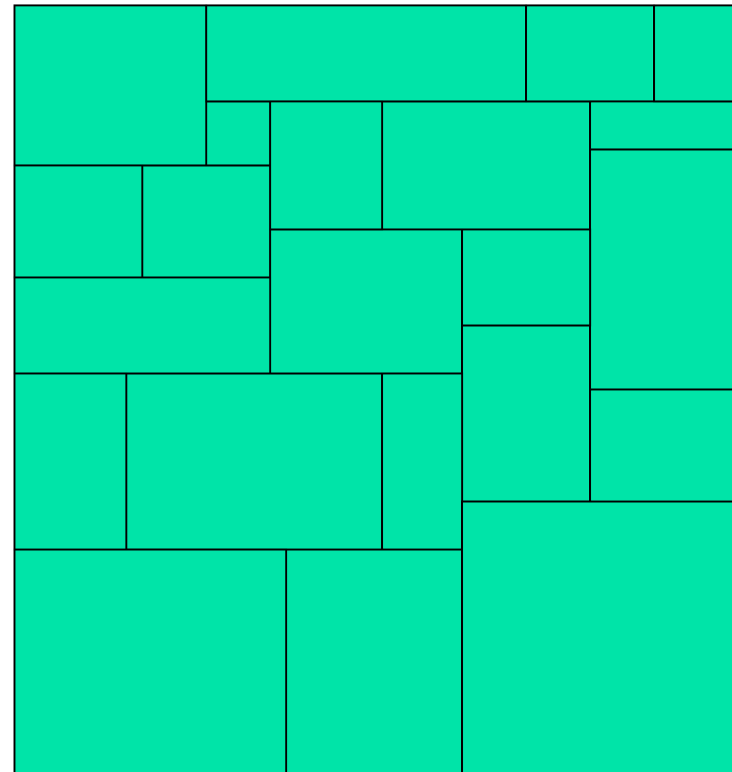
RECALL: load balancing means giving everyone roughly the same amount of work to do.

Load Balancing Is Good

- When every processor gets the same amount of work, the job is load balanced.
- We like load balancing, because it means that our speedup can potentially be linear: if we run on N_p processors, it takes $1/N_p$ as much time as on one.
- For some codes, figuring out how to balance the load is trivial (e.g., breaking a big unchanging array into sub-arrays).
- For others, load balancing is very tricky (e.g., a dynamically evolving collection of arbitrarily many blocks of arbitrary size).

Load Balancing





Load balancing can be easy, if the problem splits up into chunks of roughly equal size, with one chunk per processor. Or load balancing can be very hard.

Amdahl's Law

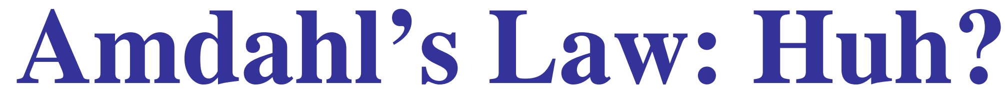
In 1967, Gene Amdahl came up with an idea so crucial to our understanding of parallelism that they named a Law for him:

$$S = \frac{1}{(1 - F_p) + \frac{F_p}{S_p}}$$

Huh?

where S is the overall speedup achieved by parallelizing a code, F_p is the fraction of the code that's parallelizable, and S_p is the speedup achieved in the parallel part.[2]

Amdahl’s Law: Huh?



What does Amdahl’s Law tell us? Well, imagine that you run your code on a zillion processors. The parallel part of the code could exhibit up to a factor of a zillion speedup. For sufficiently large values of a zillion, the parallel part would take virtually zero time!

But, the serial (non-parallel) part would take the same amount of time as on a single processor.

So running your code on infinitely many processors would still take at least as much time as it takes to run just the serial part.

Max Speedup by Serial Fraction

Maximum Speedup

10000000000
1000000000
100000000
10000000
1000000
100000
10000
1000
100
10
1

1 0.1 0.01 0.001 0.0001 0.00001 1E-06 1E-07 1E-08 1E-09 1E-10

Serial Fraction

Amdahl's Law Example

```
PROGRAM amdahl_test
  IMPLICIT NONE
  REAL,DIMENSION(a_lot) :: array
  REAL      :: scalar
  INTEGER :: index

  READ *, scalar        !! Serial part
  DO index = 1, a_lot !! Parallel part
    array(index) = scalar * index
  END DO !! index = 1, a_lot
END PROGRAM amdahl_test
```

If we run this program on infinitely many CPUs, then the total run time will still be at least as much as the time it takes to perform the **READ**.

The Point of Amdahl's Law

Rule of Thumb: When you write a parallel code, try to make as much of the code parallel as possible, because the serial part will be the limiting factor on parallel speedup.

Note that this rule will not hold when the overhead cost of parallelizing exceeds the parallel speedup. More on this presently.

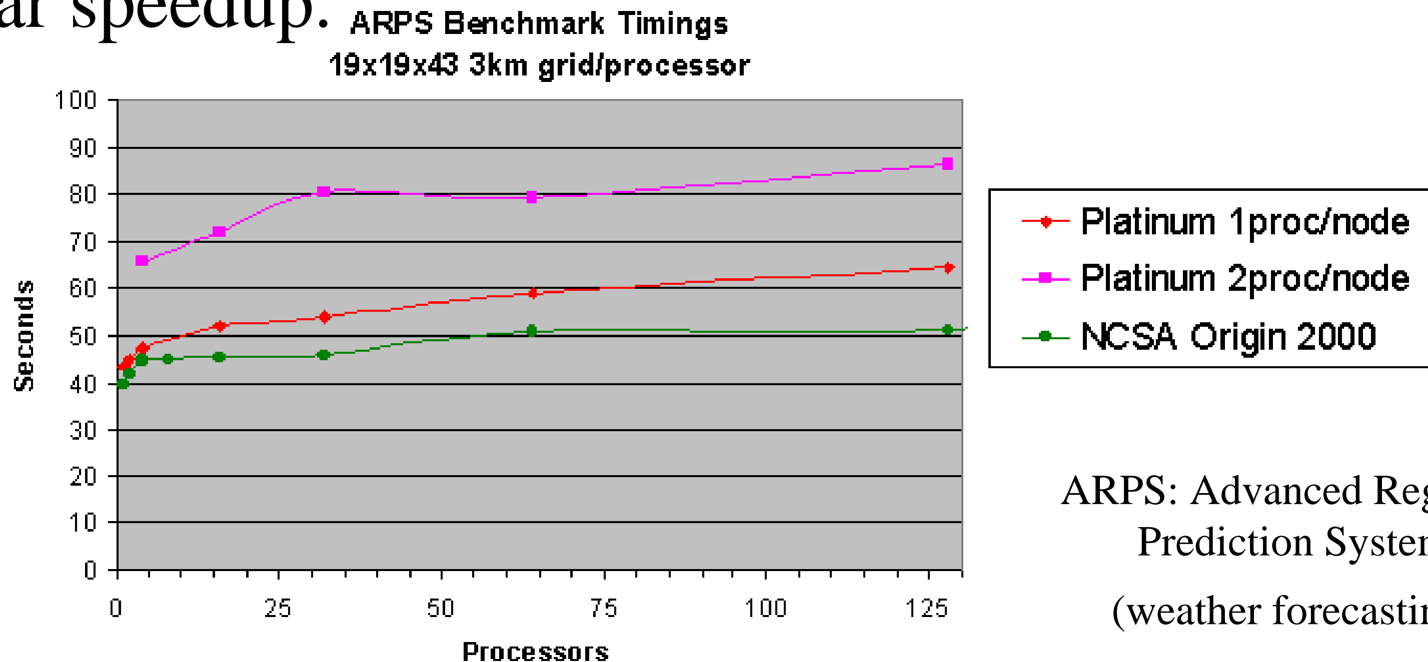
Speedup

The goal in parallelism is linear speedup: getting the speed of the job to increase by a factor equal to the number of processors.

Very few programs actually exhibit linear speedup, but some come close.

Scalability

Scalable means “performs just as well regardless of how big the problem is.” A scalable code has near linear speedup.



Platinum = NCSA 1024 processor PIII/1GHZ Linux Cluster
Note: NCSA Origin timings are scaled from 19x19x53 domains.

ARPS: Advanced Regional Prediction System

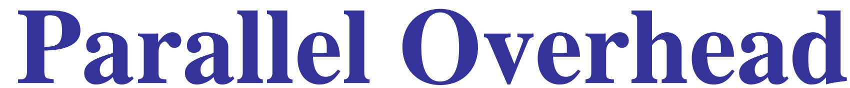
(weather forecasting)

Granularity

Granularity is the size of the subproblem that each process works on, and in particular the size that it works on between communicating or synchronizing with the others.

Some codes are coarse grain (a few very big parallel parts) and some are fine grain (many little parallel parts).

Usually, coarse grain codes are more scalable than fine grain codes, because less time is spent managing the parallelism, so more is spent getting the work done.



Parallel Overhead

Parallelism isn't free. Behind the scenes, the compiler and the hardware have to do a lot of work to make parallelism happen – and this work takes time. This time is called parallel overhead.

The overhead typically includes:

- Managing the multiple processes
- Communication between processes
- Synchronization (described later)

References

[1] © Disney
Image from
http://www.rottentomatoes.com/m/pirates_of_the_caribbean_the_curse_of_the_black_pearl/photos.php?rtp=1

[2] Amdahl, G.M. "Validity of the single-processor approach to achieving large scale computing capabilities." In *AFIPS Conference Proceedings* vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485. Cited in http://www.scl.ameslab.gov/Publications/AmdahlsLaw/Amdahls.html