An Introduction To High Performance Computing

Schedule

09:30 What is High Performance Computing (HPC)?

09.50 The Fundamentals of Computer Architectures

10.15 HPC Architectures

10.45 *COFFEE break*

Schedule

11:15 Parallel Programming Paradigms

12.00 Parallel Performance

12.45 *LUNCH*

Schedule

14:00 Using the CSCS Rosa system

14:30 Hands-on; Login, compilation, batch job submission, OpenMP, MPI and optimization

15:30 *TEA break*

16:30 Wrap-Up

17:00 Close

What is High-Performance Computing?

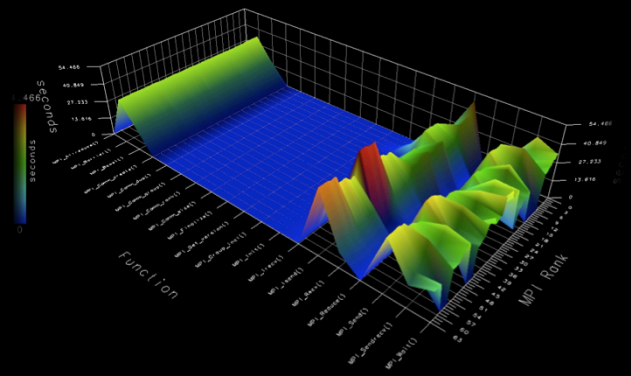
HPC Definition

There are many *definitions* but a good one is:

High Performance Computing encompasses a collection of powerful:

- hardware systems
- software tools
- programming languages
- parallel programming paradigms

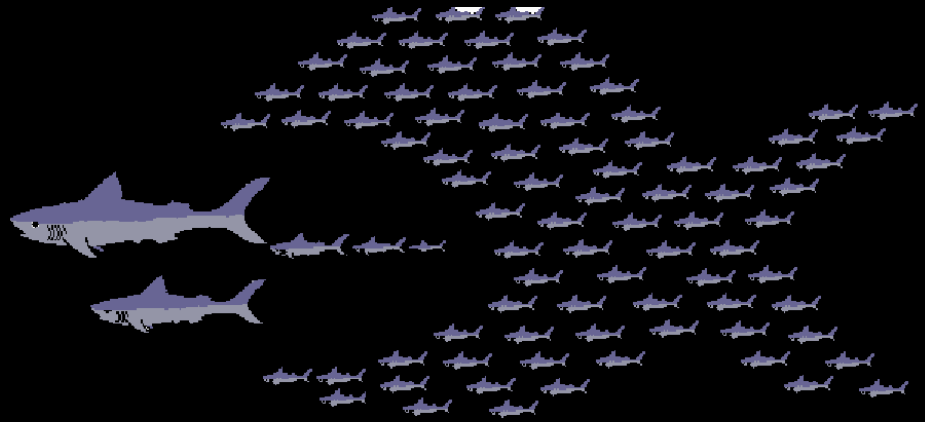
which make previously unfeasible calculations possible.



Or Simplistically !

It is all about:

Size



Speed

Why Is It Relevant ?

1. We continually demand greater computational power;
2. We want to reduce the execution time of our important applications;
3. We want to overcome the limitations of desktop computing architectures;
4. HPC-capable architectures are becoming more ubiquitous, user-friendly and affordable.

Computational Science

HPC supplements traditional scientific and engineering methods i.e.

- Writing a theory or paper design
- Performing experiments or building systems

by carrying out numerical calculations on real phenomena and/or experiments that are:

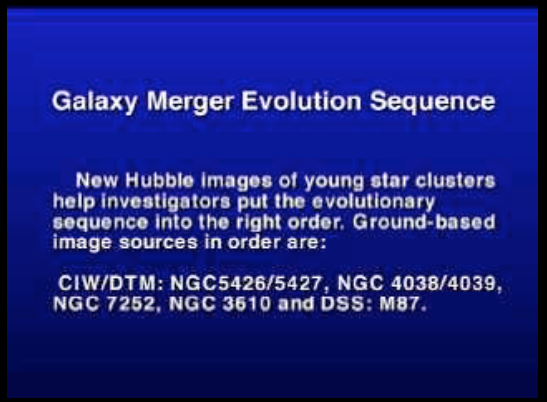
1. Too Hard

   e.g. building large wind tunnels

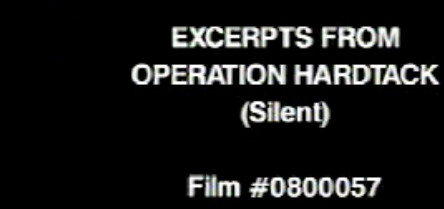
Computational Science

2. Too Expensive
   e.g. building a throw-away passenger jet

3. Too Slow
   e.g. waiting for climate or galactic evolution



4. Too Dangerous / Controversial
   e.g. testing nuclear weapons and stem cell research



Problem Domains

HPC can benefit researchers who wish to carry out huge amounts of repetitive calculations on large amounts of data and wish to obtain valid results in a reasonable time (whether it be a *Grand Challenge Problem* or a smaller project).

Typical problem domains include but are not limited to:

- Quantum Chemistry and Relativistic Physics
- Cosmology and Astrophysics
- Computational Fluid Dynamics
- Biology, Genome Sequencing, Genetic Engineering Medicine
- Global Weather and Environmental Modelling
- Engineering and Crash Test Simulations
- Computational Finance

HPC Economic Impact

Airlines

System-wide logistics optimizations evaluated on HPC systems save approx. US$100 million per airline per year.

Automotive Design

Major companies use (500+ CPUs) in CAD and CAM for crash testing, structural integrity and aerodynamics saving over US$1 billion per company per year.

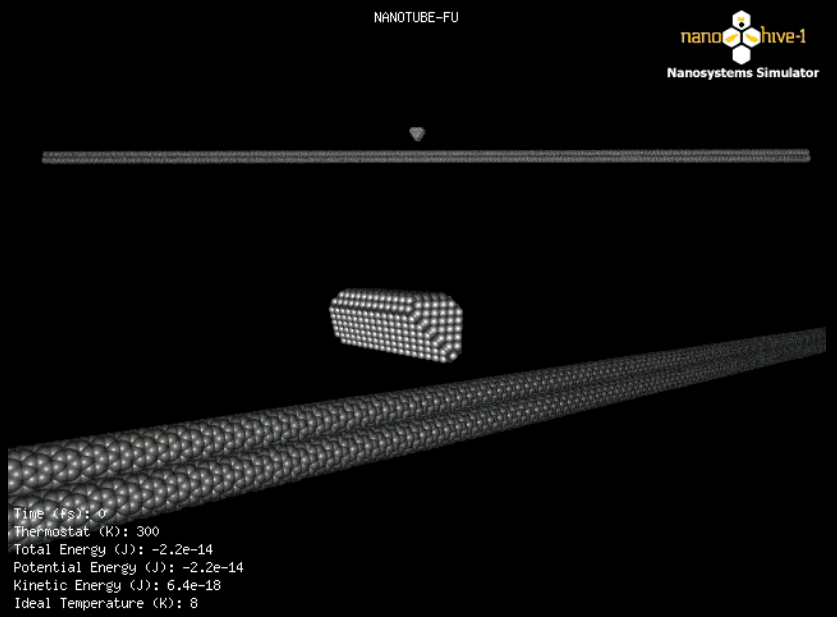
HPC Economic Impact

Semiconductor Industries

Semiconductor firms use large systems (500+CPUs) for device electronics simulation and logic validation saving approx. US$1 billion per company per year

Securities Industry

Savings approx. US$15 billion per year for US home mortgages.

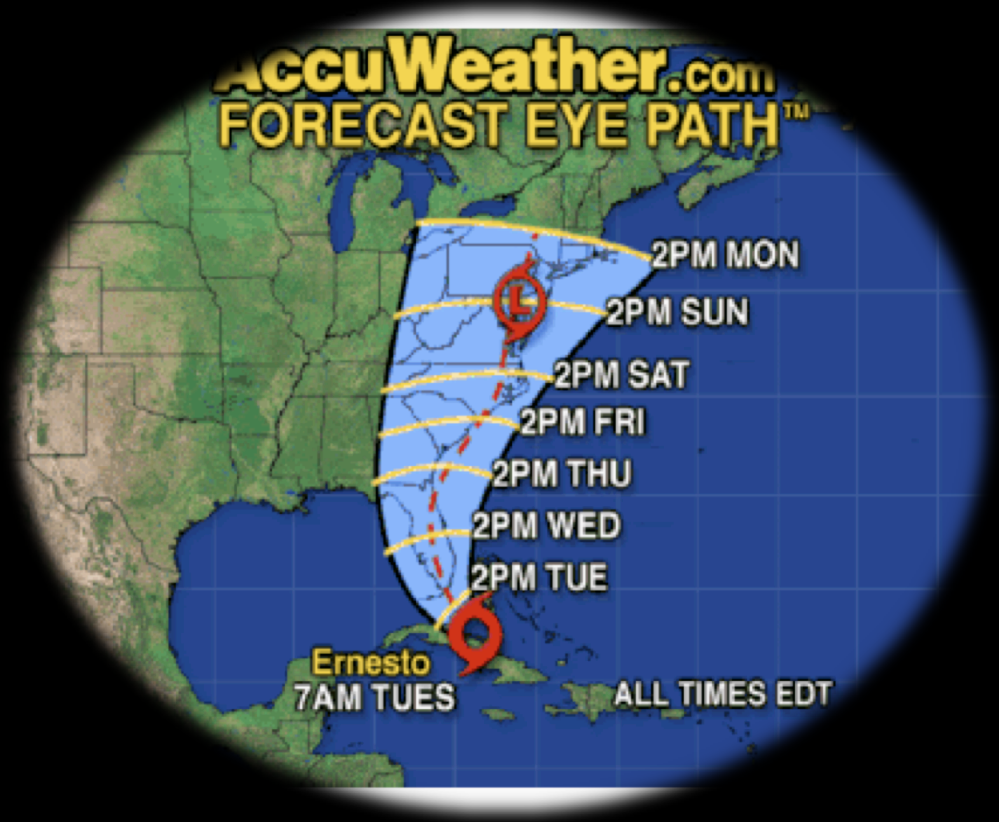


Three carbon nanotubes are anchored at their ends, and a diamondoid carbon "knife" is pushed down on the nanotubes with a 5 nanonewton force.

The system comprises ~20,000 atoms and runs for 5.5 picoseconds of simulation time

Case Studies

Cost (Economic Loss) to Evacuate 1 Mile of US Coastline is US$1,000,000.



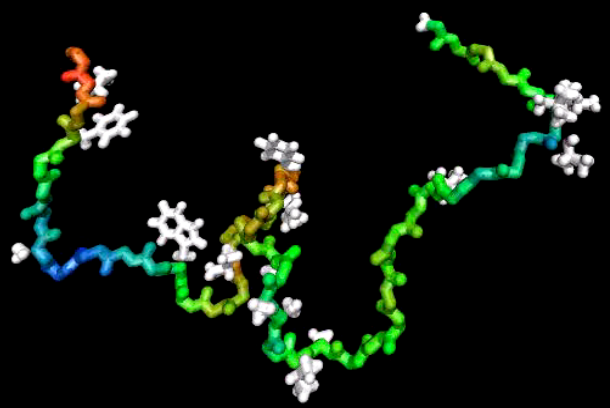
- Authorities over-warn by a factor of 3.
- Average over-warning is 200 miles of coastline or US$200 million

Case Studies

Protein Folding

To simulate the folding of a 300 amino acid protein in water with ~32,000 atoms and a folding time of 1 milisecond would require:

A 1 PetaFLOP/s (10^{15}) machine running for 1 year



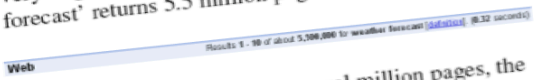
Google

Computing the Google PageRank

The World's Largest Eigenvalue Problem

What is PageRank

A search with Google's search engine usually returns a very large number of pages. E.g., a search on 'weather forecast' returns 5.5 million pages.



Although the search returns several million pages, the most relevant pages are usually found within the top ten or twenty pages in the list of results.

How does the search engine know which pages are the most important?

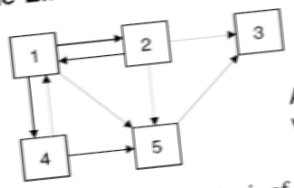
Google assigns a number to each individual page, expressing its importance. This number is known as the PageRank and is computed via the eigenvalue problem

$$Pw = \lambda w$$

where P is based on the link structure of the Internet.

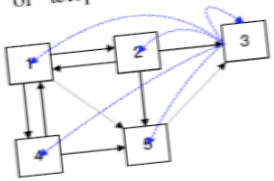
The key problem is to formulate the link structure, i.e., the matrix P, in a proper way.

The Link Structure Matrix P



A tiny internet with 5 pages.

The model forming the basis of the PageRank algorithm is a random walk through all the pages of the Internet. Let $p_t(x)$ denote the possibility of being on page x at time t. The PageRank of page x is expressed as $\lim(p_t(x))$ for $t \to \infty$. To make sure the random walk process does not get stuck, pages with no out-links (here: page 3) are assigned artificial links or "teleporters" to all other pages.



$$P = \begin{pmatrix} 0 & 1/3 & 1/5 & 1/2 & 0 \\ 1/3 & 0 & 1/5 & 0 & 0 \\ 0 & 1/3 & 1/5 & 0 & 1 \\ 1/3 & 0 & 1/5 & 0 & 0 \\ 1/3 & 1/3 & 1/5 & 1/2 & 0 \end{pmatrix}$$

The matrix P is irreducible and stochastic and therefore the random walk can be expressed as a Markov chain, and the PageRank of all pages can be computed as the principal eigenvector of P.

The PageRank Algorithm

The Google matrix P is currently of size 4.2×10^9 and therefore the eigenvalue computation is not trivial. To find an approximation of the principal eigenvector the *power method* is used:

```
w0 = initial guess
For k = 1 to 50
    wk = P*wk-1
End
Return w50
```

The special properties of the matrix P ensures that the largest eigenvalue is $\lambda = 1$, rendering normalisation in the power method unnecessary. Fast convergence of the power method makes 50 iterations adequate.

Because the computation involves an extremely large matrix, the matrix-vector multiplications must be implemented in parallel on multi-processor systems.

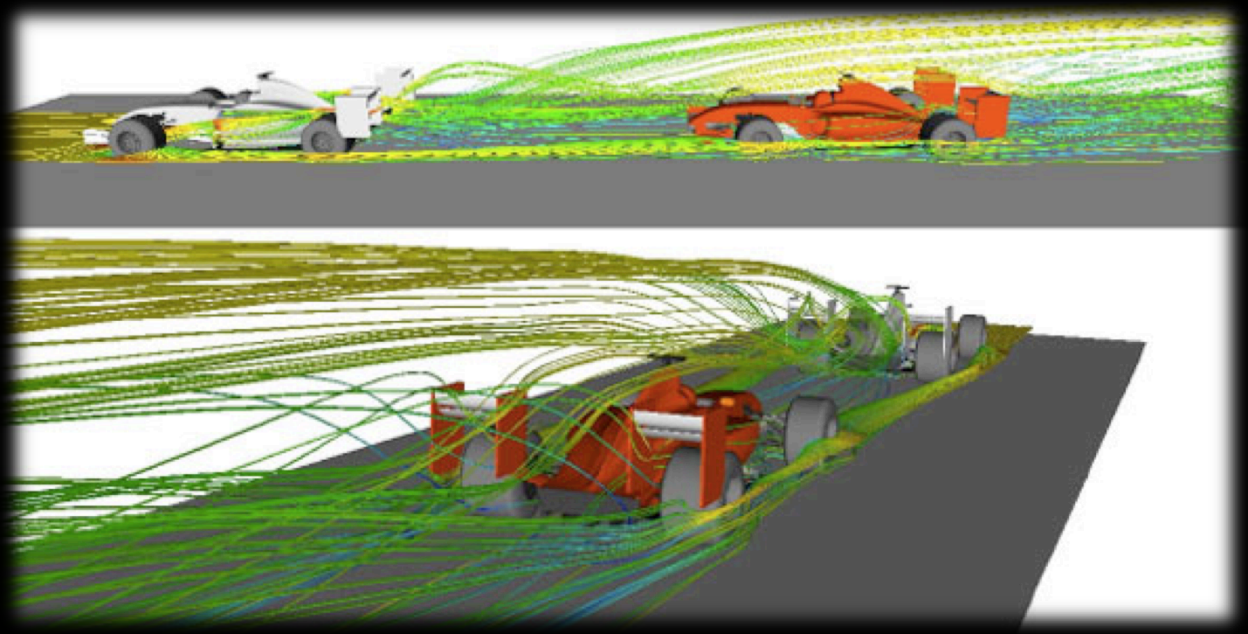
Informatics and Mathematical Modelling

DTU

Formula 1 Racing

Having won more than 150 Grands Prix, 11 drivers' and eight constructors' world titles, McLaren is one of the most successful teams in the history of Formula 1.

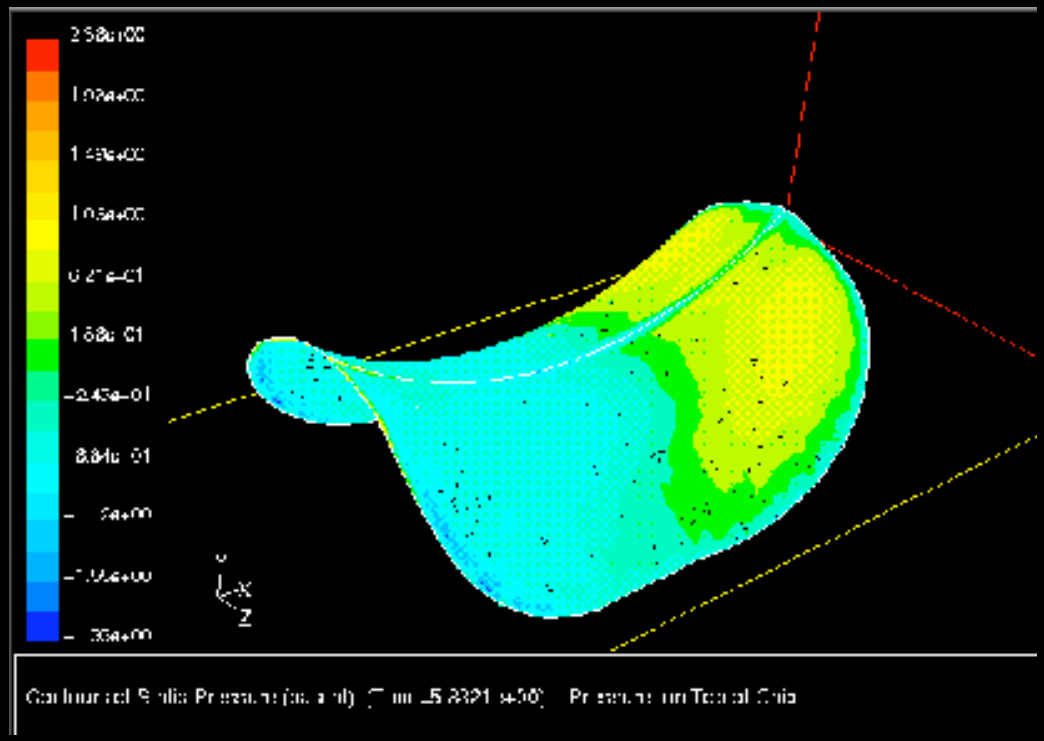
McLaren use CFD simulations to model airflows over a Formula 1 car (each of which features more than 11,000 separate components) to help with developing its shape. These calculations help to provide enhanced grip, particularly during cornering as well as help increase understanding of the behavior of the car in yaw (crosswind, cornering), steer (with the front wheels turned) and roll (ride-height variations).



Animation

- The movie *Shrek 3*, consumed close to 20 million CPU render hours
- Each frame was rendered at DreamWorks Animation, with more than 1,000 Linux desktops and more than 3,000 server CPUs
- Each frame is assigned to a different node of the renderfarm by grid software (using Platform LSF, a commercial Linux package), so that many frames can be output simultaneously.

Case Studies



Proctor & Gamble use HPC to model the production and assembly of Pringles and Pampers Nappies

Summary

HPC is being used by researchers around the world (from large groups to individuals) to complement the standard scientific method.

With access to DIY HPC hardware and a free OS such as Linux, entry-level HPC is within reach of even the most modest budget.

With that said, to fully exploit HPC one needs to obtain detailed knowledge of HPC system architectures as well as master HPC development tools and advanced programming techniques.

You will gain an insight into this in the remainder of this tutorial.

The Fundamentals of Computer Architectures

Basic Concepts

A Process

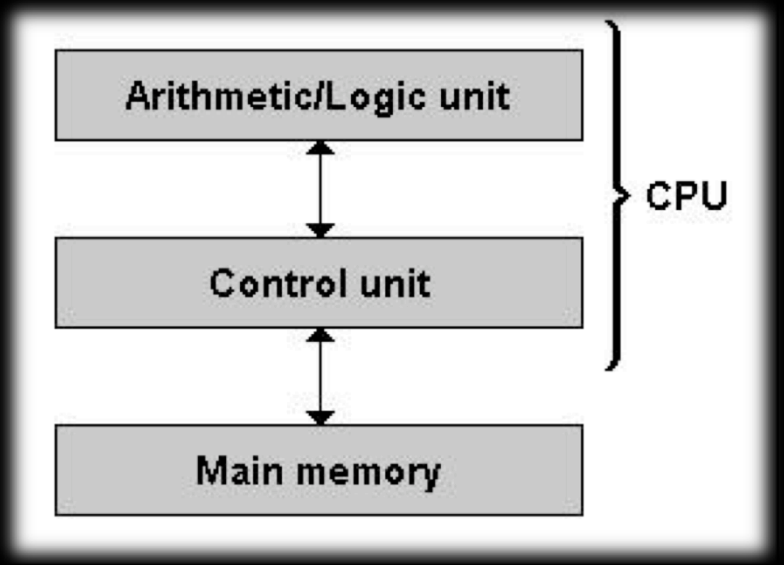
is a task being run by a computer, often simultaneously with many other tasks each taking turns on a single central processing unit (CPU).

Control Unit

Schedules the order of instruction execution.

Arithmetic/Logic Unit:

Performs calculations – e.g., adding, multiplying, checking whether two values are equal



Basic Concepts

Registers

- where data resides that is being used right now

Cache

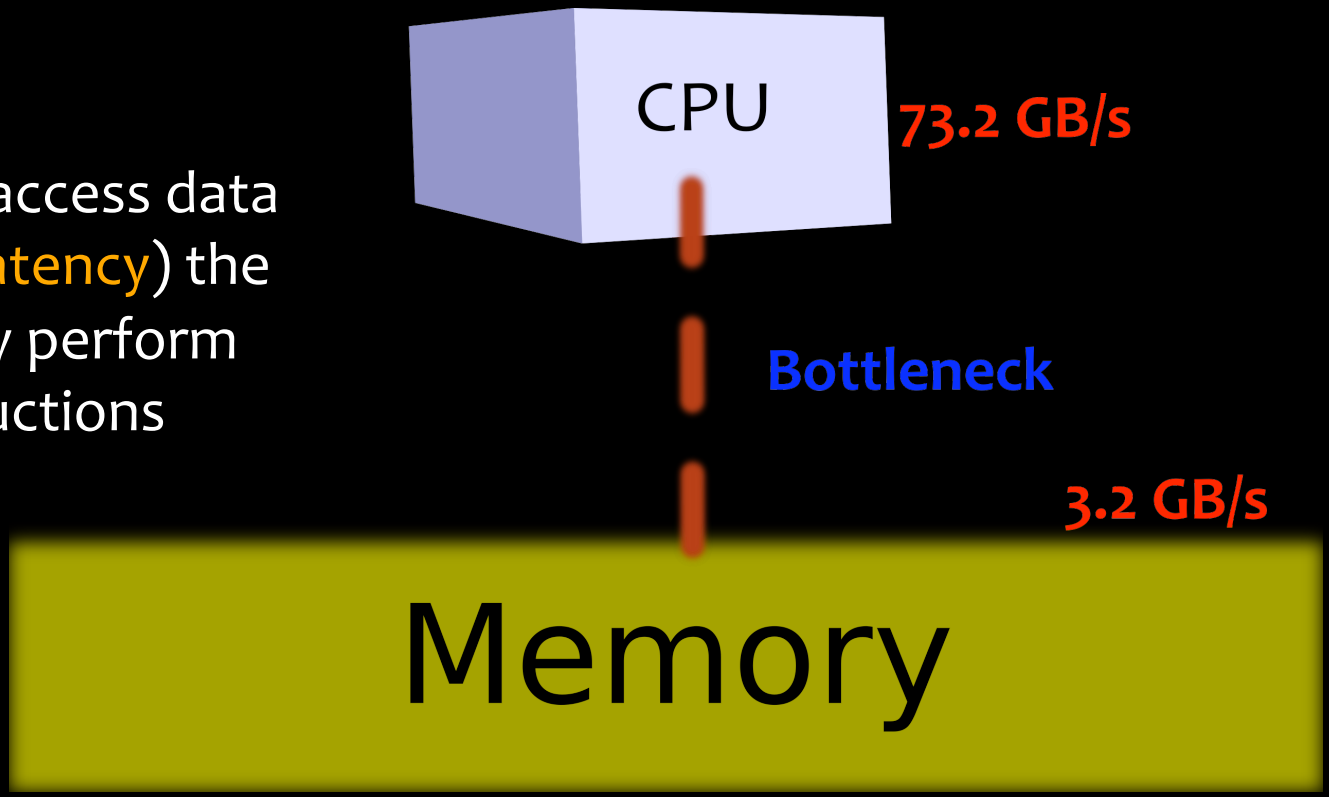
- small area of fast memory
- where data resides when it is about to be used and/or has been used recently

Main Memory

- Also called RAM ("Random Access Memory")
- Where data resides when it is being used by a program that is currently running
- Primary storage is volatile: values in primary storage disappear when the power is turned off.

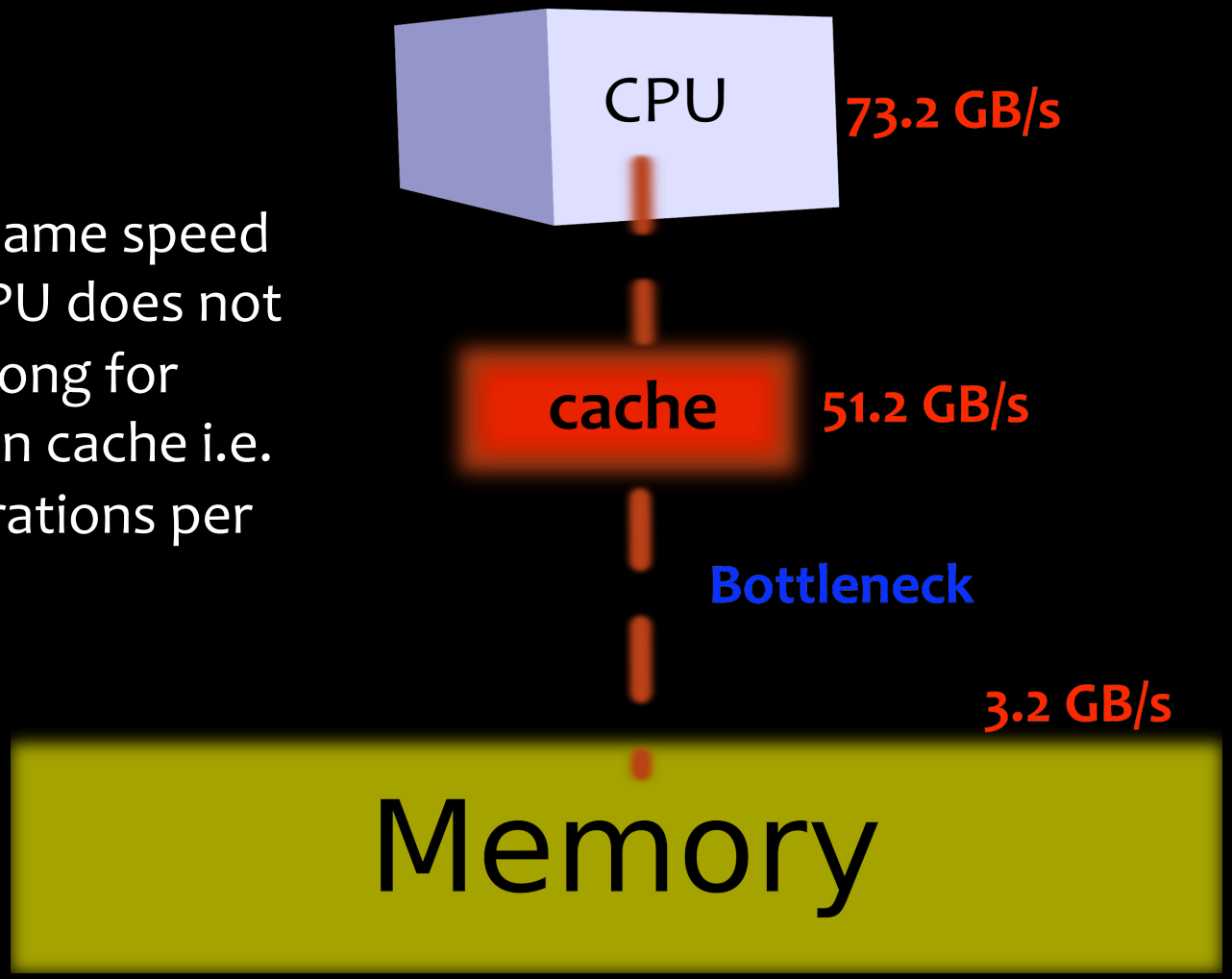
RAM is slow

In the time it takes to access data from Main Memory (latency) the CPU can theoretically perform hundreds of instructions



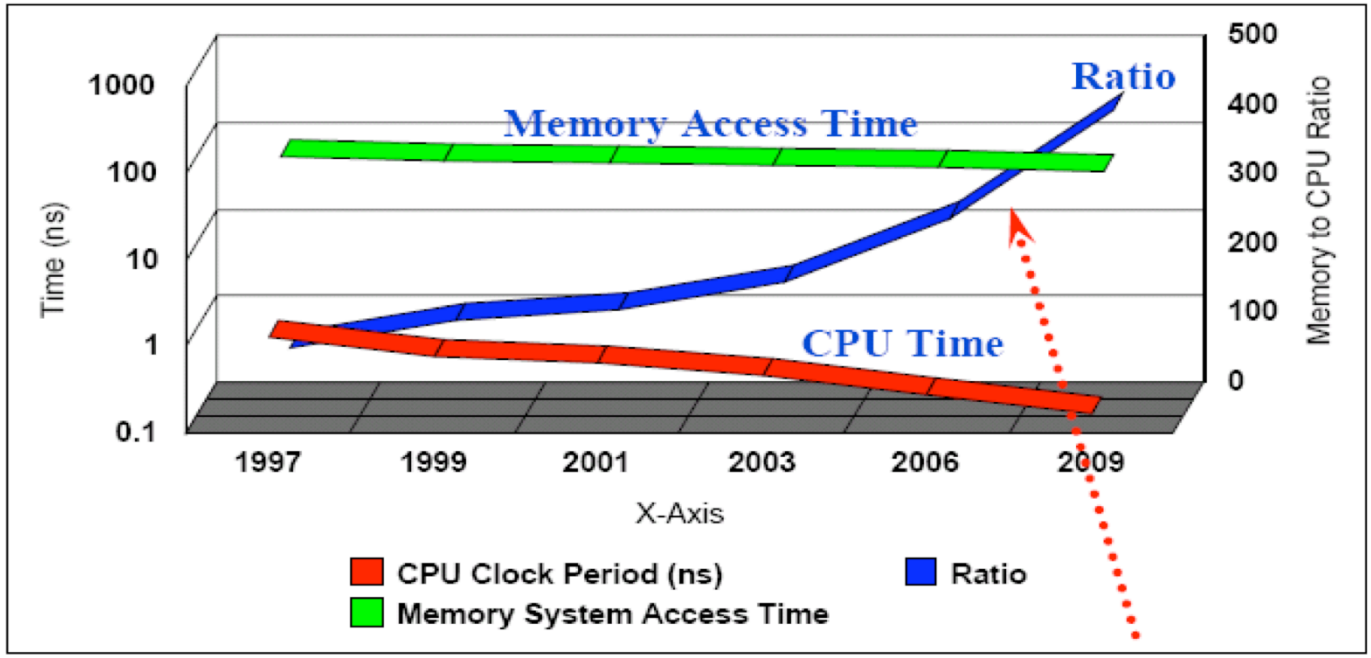
RAM is slow

Cache is nearly the same speed as the CPU, so the CPU does not have to wait as long for Data that is already in cache i.e. it can do more operations per second!



Memory Hierarchy

Latency in a Single System



THE *WALL*

25

Memory Latency

Levels of the Memory Hierarchy

Capacity
Access Time
Cost

Upper Level

Staging
Xfer Unit

faster

CPU Registers
100s Bytes
<10s ns

Registers

Instr. Operands

prog./compiler
1-8 bytes

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Cache

Blocks

cache cntl
8-128 bytes

Main Memory
M Bytes
200ns- 500ns
$.0001-.00001 cents /bit

Memory

Pages

OS
512-4K bytes

Disk
G Bytes, 10 ms
(10,000,000 ns)
10^{-5} - 10^{-6} cents/bit

Disk / Distributed Memory

Files

user/operator
Mbytes

Tape
infinite
sec-min
10^{-8}

Tape / Clusters

Larger

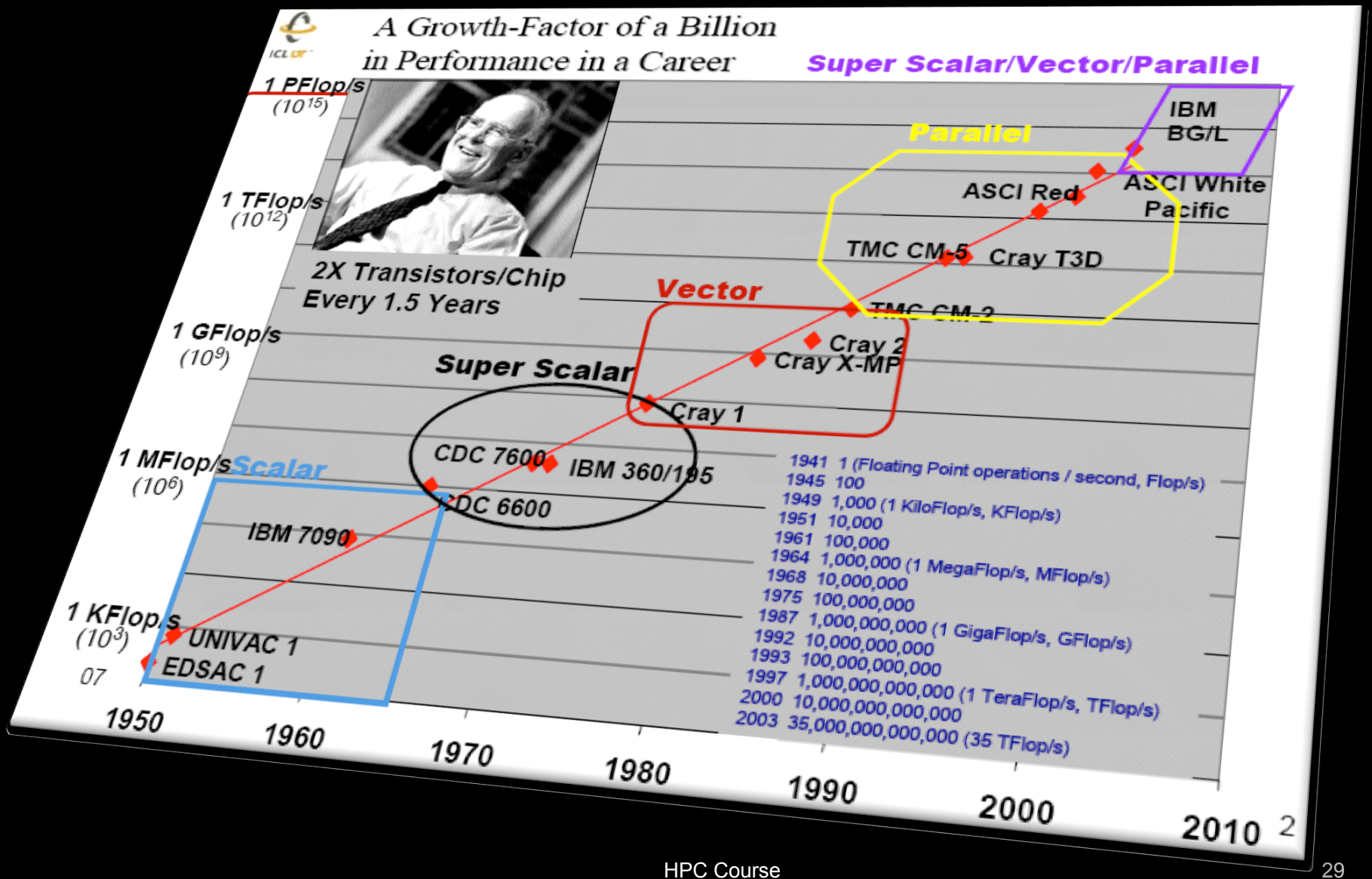
Lower Level

9

Whence HPC?

HPC technologies try to overcome the limitations of single processor memory latencies and sizes by coordinating *many single CPU's* to work together to solve a task.

HPC Architectures



HPC - Challenges

Porting existing Software

- develop a new algorithm or restructure an existing algorithm to exploit HPC architectures
- specialized training

Technology traps

- processors are developing rapidly
- communication speeds are lagging behind
- systems software further behind
- application software even further behind

HPC - Challenges

Productivity Traps

- technology distraction
- insufficient programming skills
- limited knowledge and utilisation of HPC development tools e.g.
  - Compiler flags
  - Debuggers
  - Profilers

Consequences

- longer development cycle
- portability of codes (standard compliant)
- waste of HPC resources

HPC Architectures

HPC Architectures

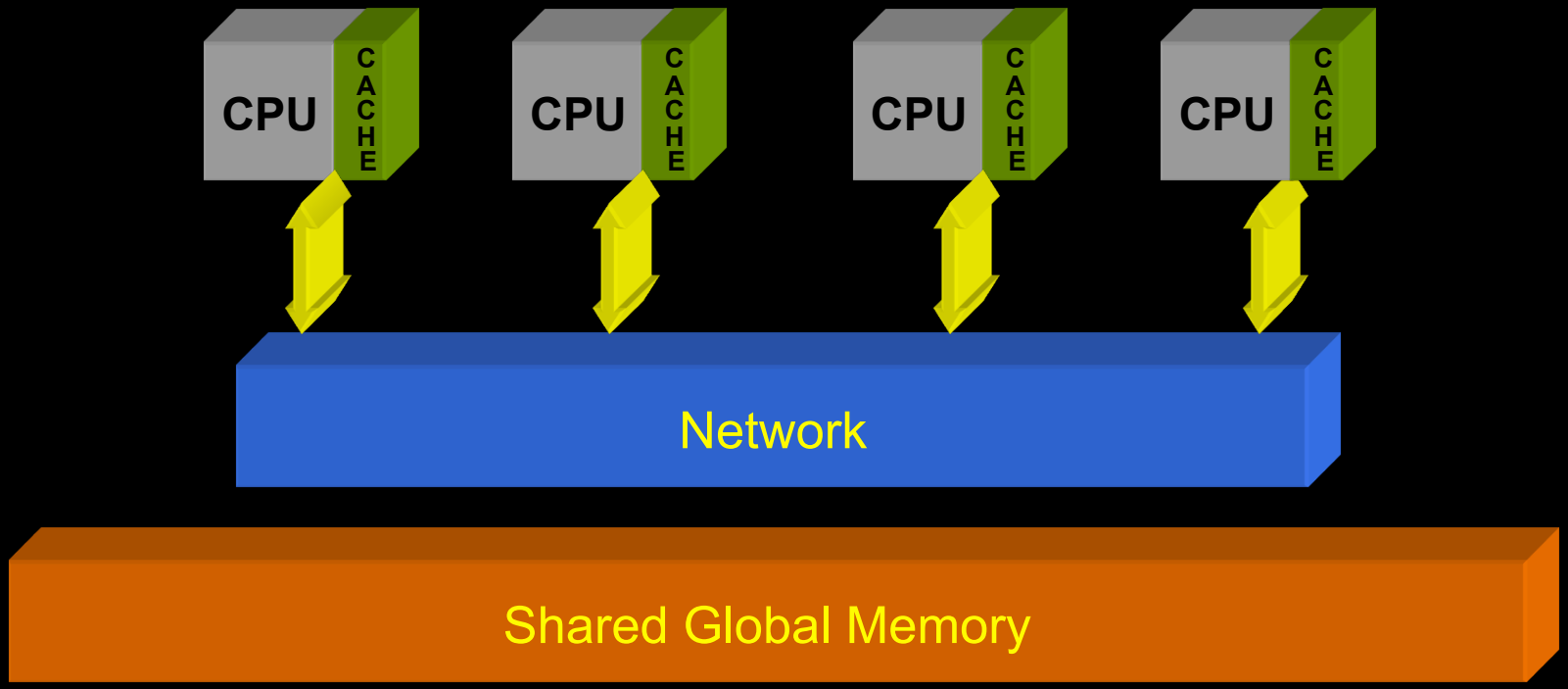
HPC architectures try to maximise performance by simultaneously employing many Processing Elements (CPUs) together to solve a given task.

In general these parallel processing machines can be classified into two main groups based on how the CPUs *view* the available memory:

1. Shared-Memory machines
2. Distributed-Memory machines

Shared-Memory

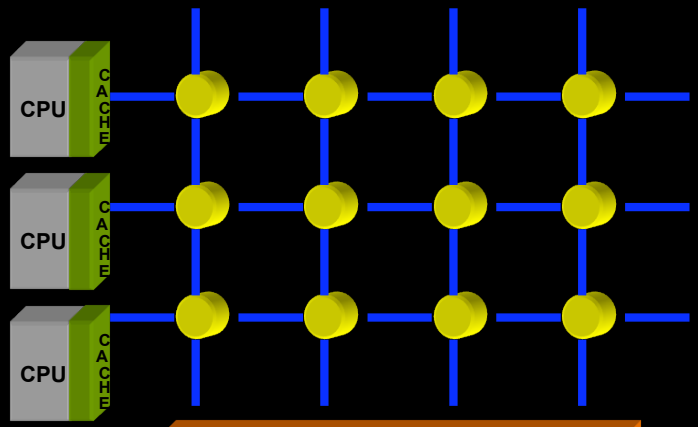
Shared-memory systems have multiple CPUs all of which share the same global address space.



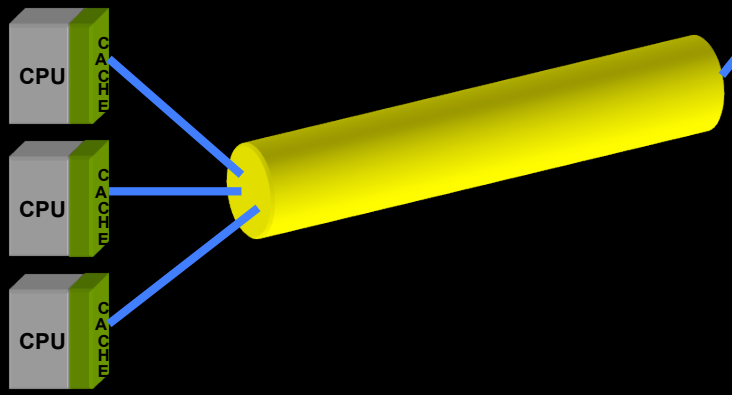
Interconnects

The network fabric is typically one of the following designs:

- Crossbar

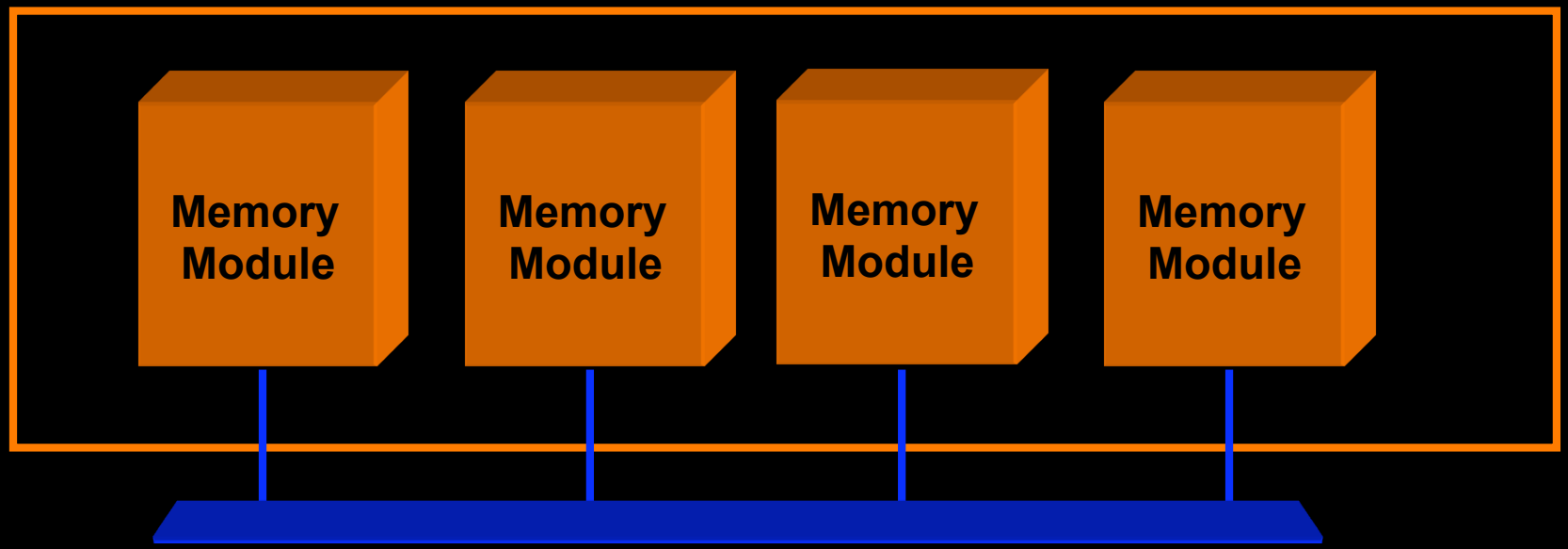


- Bus



Shared-Memory

In many systems, shared-memory is logically global but physically segregated.



This leads to two further sub-classifications based on memory latency delays.

HPC Architectures

SMP (Symmetric Multi-Processing)

The latency to access any address in the logical memory space is the same for each CPU.

NUMA (Non-Uniform Memory Access)

The latency to access any address in the logical memory space is determined by the physical distance from the CPU.

HPC Architecture

Cache-Coherency

To ensure cache consistency (*i.e. local cache has the most up-to-date copy of a shared memory resource*), cache-coherency protocols are implemented on modern systems.

NUMA systems that enforce cache-coherency are referred to as ccNUMA systems.

Advantages

- A Global Address Space provides a user-friendly programming perspective to memory (such as with the OpenMP API which will be discussed later)
- Data sharing between tasks is both fast and uniform due to the proximity of memory to the CPUs

Disadvantages

- Need for cache-coherency
- Lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsible for synchronization constructs that ensure "correct" access of global memory (i.e. prevent *race conditions*)

Vector Processors

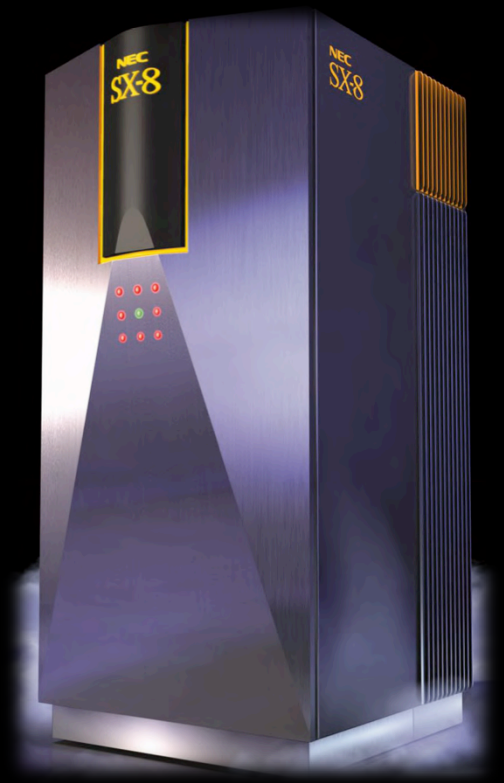
A subclass of shared-memory machines is the *vector processor*.

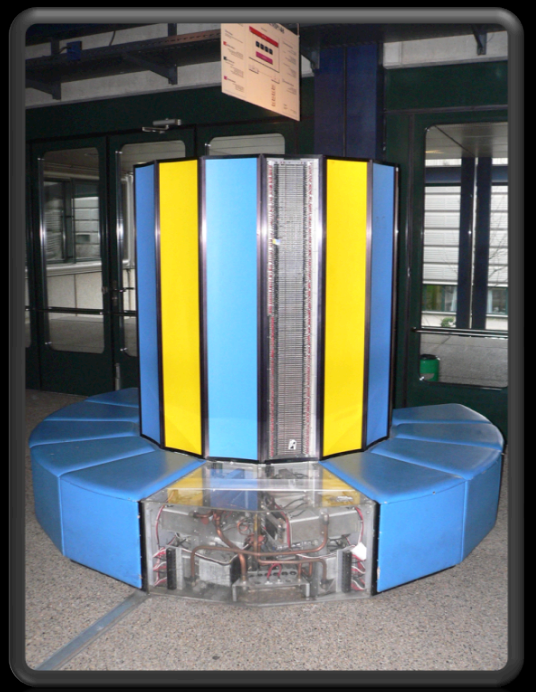
- Vector processors provide high-level operations that work on vectors (1D array of data values).
- A typical instruction might add two 64 element FP vectors e.g. D = A +C

Examples

NEC SX-8

- Eight-way SMP system in each node.
- Each CPU is a vectorprocessor
- Up to 512 nodes





Cray X-MP.

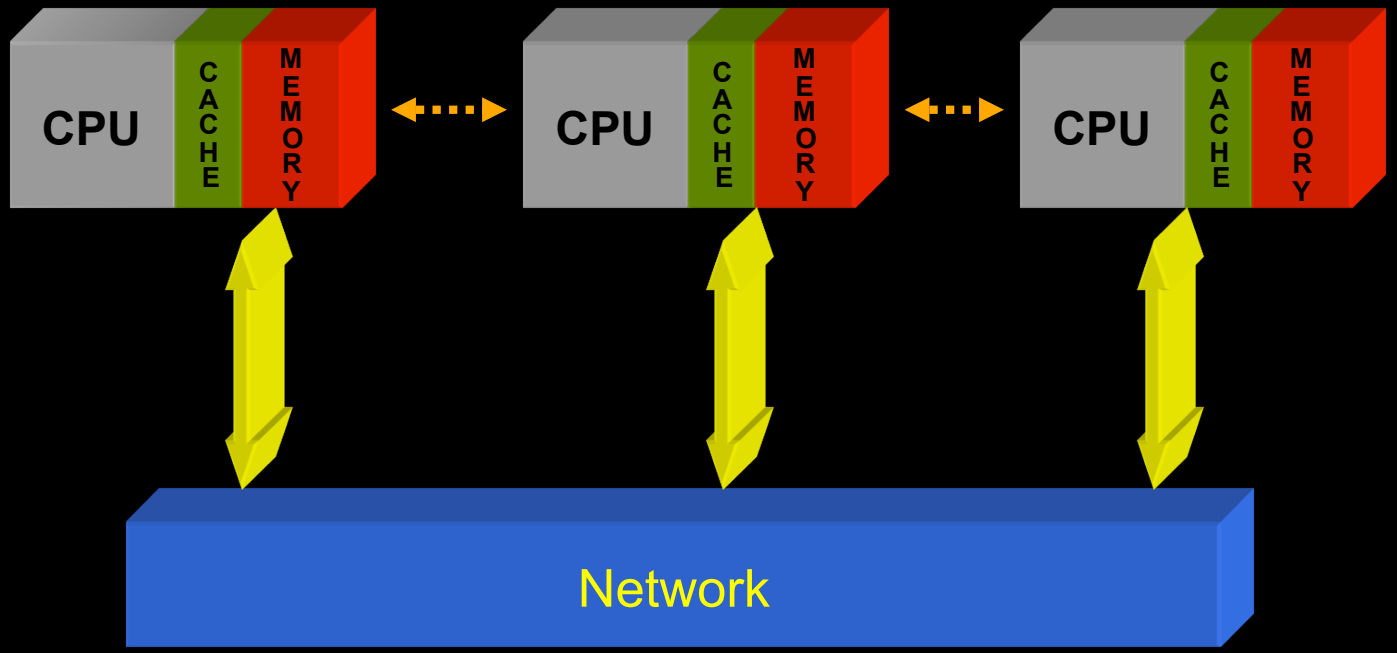
- Available with 1-4 vector processors
- Had less than half of the raw power of Microsoft's XBOX console.
- Sold for US$15 million in 1984.

Advantages

- It is equivalent to executing an entire loop (memory-parallelism)
  - Reducing instruction fetch and decode bandwidth.
- Memory access for entire vector, not a single word.
  - Reduced Latency
- Multiple vector instructions in progress.
  - Further parallelism
- Typically no cache.
  - Therefore no need for cache-coherency logic

Distributed-Memory

Distributed-memory systems have multiple nodes each with their own local memory.



Advantages

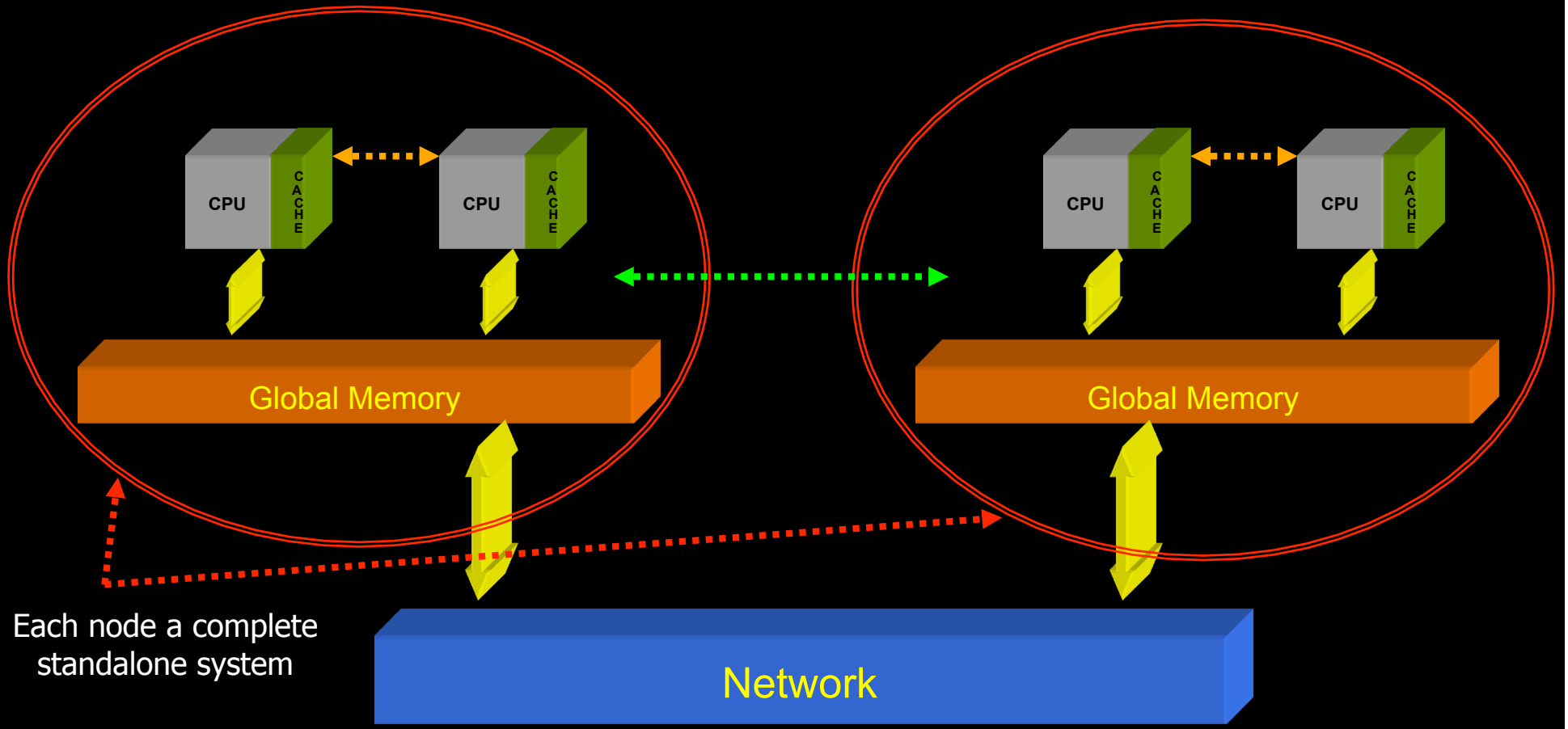
- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages

- Programmer is responsible for mapping data structures across nodes
- Programmer is responsible for coordinating communication between nodes when remote data is required in a local computation (called *message-passing*)
- Access to remote memory is significantly slower than to local memory
- Currently, only low-level programming API's (such as *MPI*) are available to perform message-passing between nodes

Clusters

Systems incorporating both shared and distributed-memory architectures (by collecting together complete standalone HPC systems.)



Constellations

Constellations are clusters where the number of CPU's per node is typically > 4 *(i.e. fat nodes)*.

HPCX (UK)

Each eServer 575 node contains 8 DCM's
- 16 CPUs
- L3 cache local to each DCM
- 32 GB shared memory per node

8 nodes make up a frame/cabinet
- 128 CPUs
- peak of 768Gflops

12 frame/cabinets in total
1536 CPUs
9.2 TeraFlops peak

Clusters

Traditional clusters have many interconnected thin nodes (1-4 CPU's). Commodity clusters are comprised of commodity of the shelf (COTS) processors and interconnects.

TSUBAME (Japan)

- 655 X4600 Sun Fire Servers
  - X4600 Server - 8 sockets (dual core up to 16 CPUs
  - up to 64GB on each server
- Up to 10,480 Opteron Cores and 21TB of RAM

Advantages

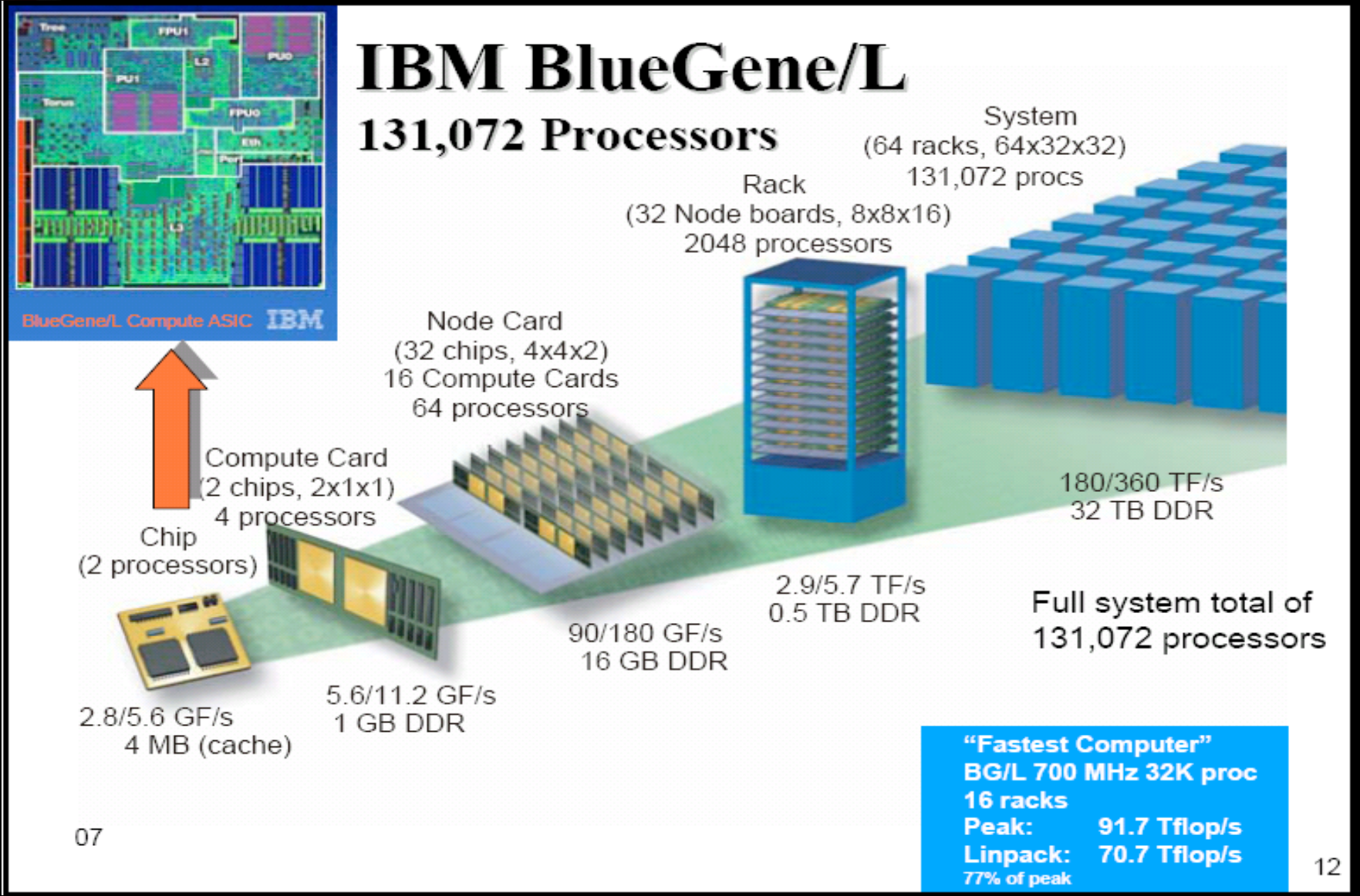
- Exploits the advantages of both shared and distributed-memory architectures at different levels.
- Can exploit both shared and distributed-memory programming paradigms (*OpenMP* and *MPI*) to solve difficult tasks.
- Can be built from commodity processors and interconnects.

More Examples

Blue Gene/L (US)

- 65,536 processors made up from:
  - 64 Racks
  - Each rack has 32 node
  - Each node has 16 compute cards
  - Each node card has 2 chips and 1GB of local memory
  - Each chip has 2 cores
- 360 T/flops peak performance

BlueGene Data

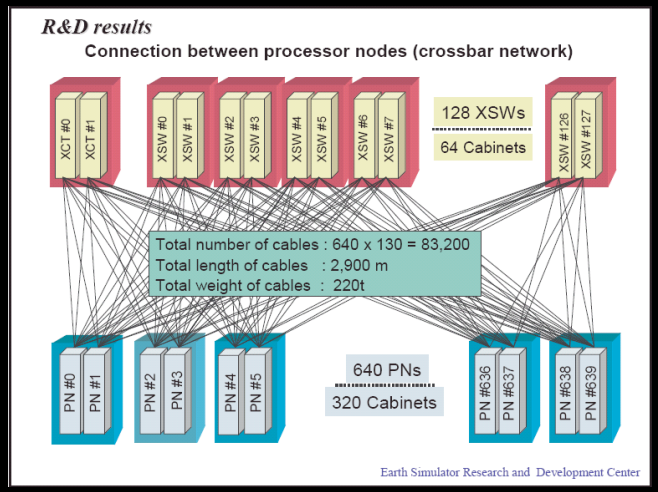


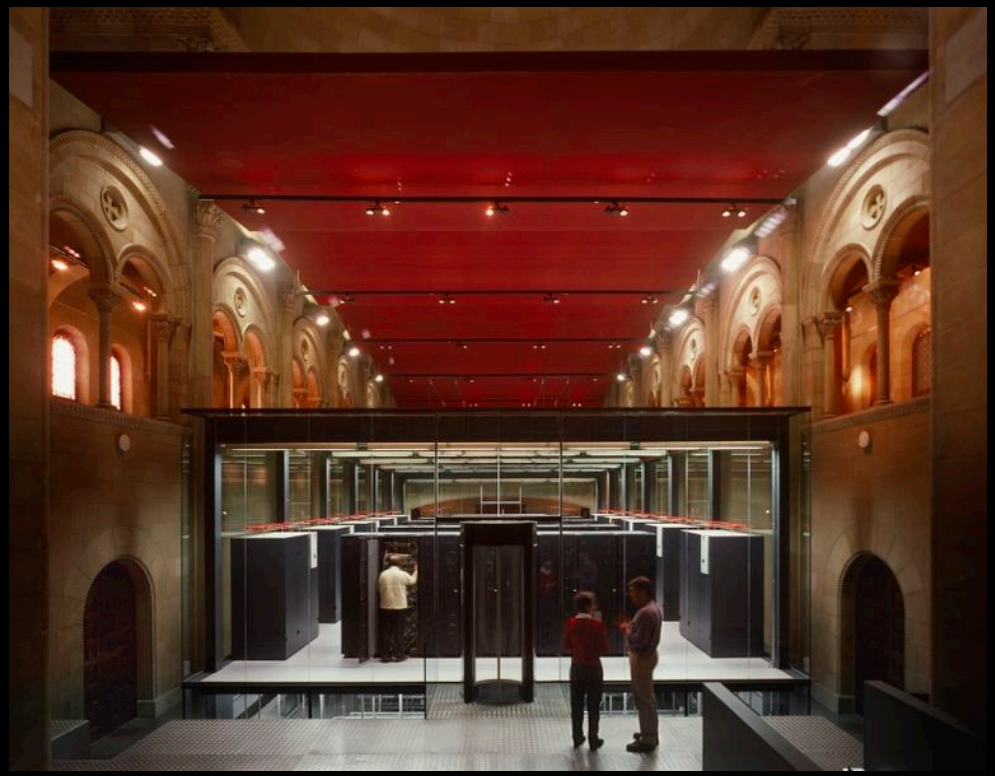
More Examples

Earth Simulator (Japan)


The Earth Simulator Center

- 5,104 processors made up from:
  - 640 Nodes
  - Each node has 8 vector processors and 16GB of shared memory
- Crossbar (over 1800 miles of cable)
- Floor area spans 4 tennis courts
- 40 T/flops peak performance





Examples

MareNostrum (Spain)

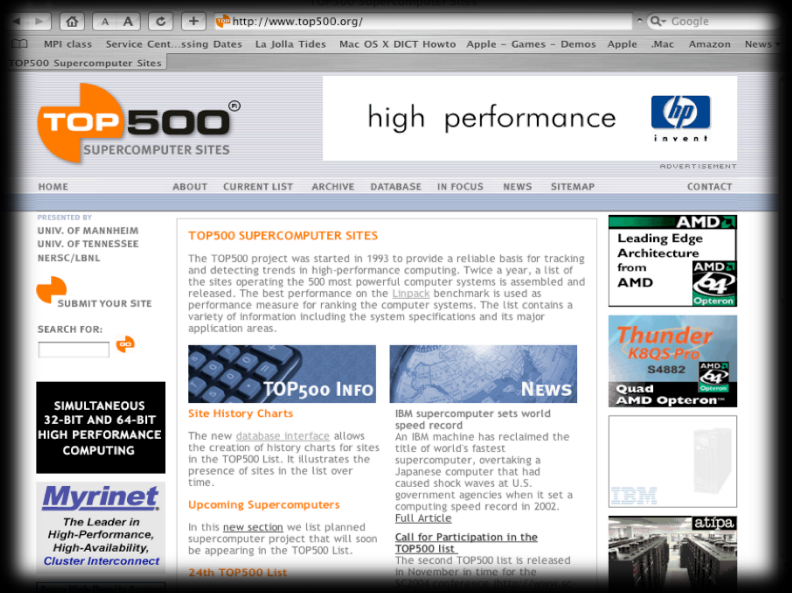
Consists of:

- 2406 computing nodes, each with dual IBM 64-bit PowerPC 970FX processors running at 2.2 GHz, 4812 CPUs in total.
- It is capable of 27.91 teraflops and a peak performance of 42.144 teraflops.
- It occupies only 120 m^2 (less than half a basketball court) and weighs 40,000 kg.
- It was largely constructed in two months in Madrid and was installed in the Barcelona Supercomputing Center, Spain

Information

Since 1993 parallel machine performance has been measured and recorded with the LINPACK benchmark, as part of Top500.

- It is a good source of information about what machines there are (were) and how they have evolved.
- http://www.top500.org
- Latest List - June 2006
- Monte Rosa– position #23

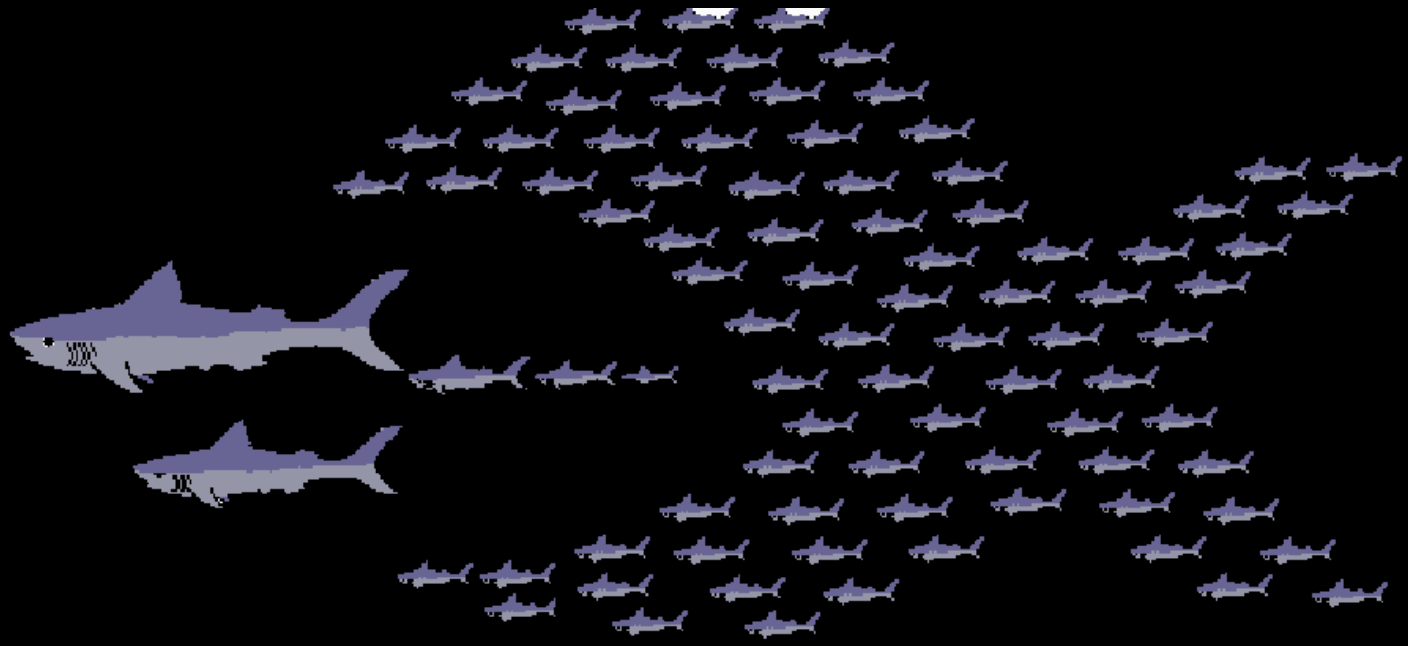


Parallel Programming Paradigms

Why Parallelisation?

Parallelism means doing multiple things at the same time:

- you can get more work done in the same time.



Resistance?

- Software inertia
  - cost of converting existing software
- Lack of training
- Limited access to resources (limited tools?)
- In parallel world, need to choose the right algorithm to suit the architecture
- Parallel algorithm may only perform well on one class of machine

Parallel Programming Paradigms

Shared Memory

Threads

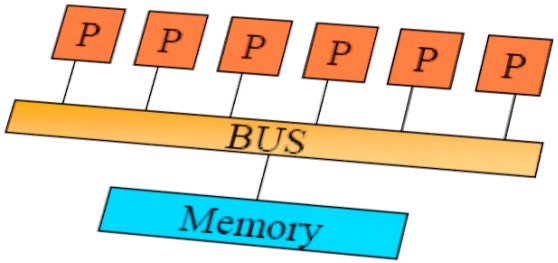
Data Parallel

Hybrid

Message Passing

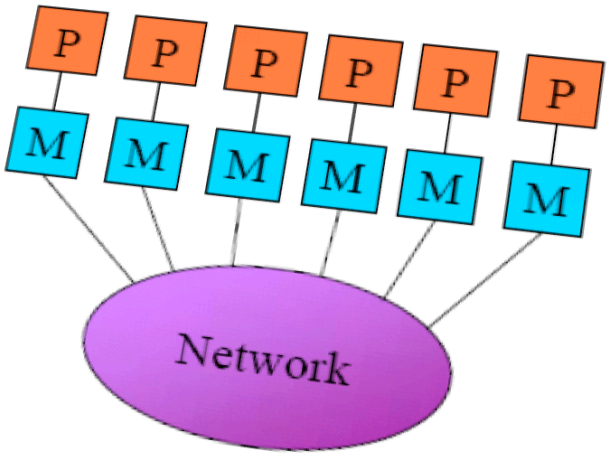
Recap

Shared vs. Distributed Memory



Shared memory - single address space. All processors have access to a pool of shared memory. (Ex: SGI Origin, Sun E10000)

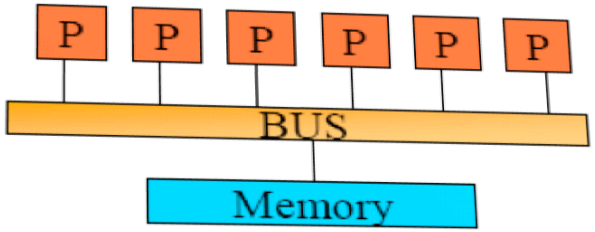
Distributed memory - each processor has it's own local memory. Must do message passing to exchange data between processors. (Ex: CRAY T3E, IBM SP, clusters)



79

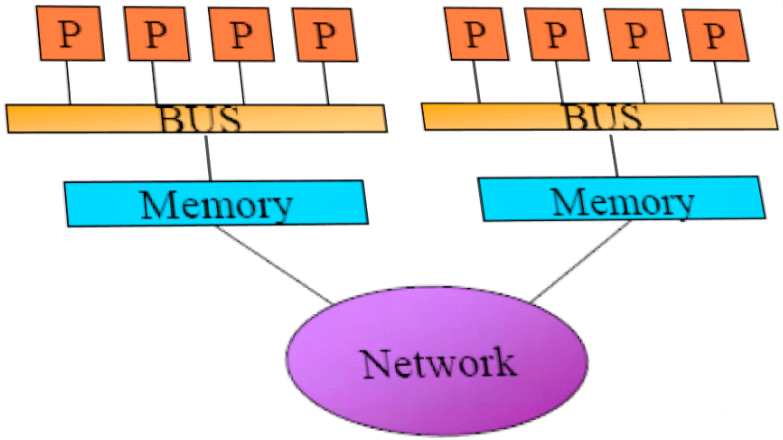
Recap

Shared Memory: UMA vs. NUMA



Uniform memory access (UMA): Each processor has uniform access to memory. Also known as symmetric multiprocessors (Sun E10000)

Non-uniform memory access (NUMA): Time for memory access depends on location of data. Local access is faster than non-local access. Easier to scale than SMPs (SGI Origin)



80

Shared-Memory Programming

Shared-Memory Programming

Advantage

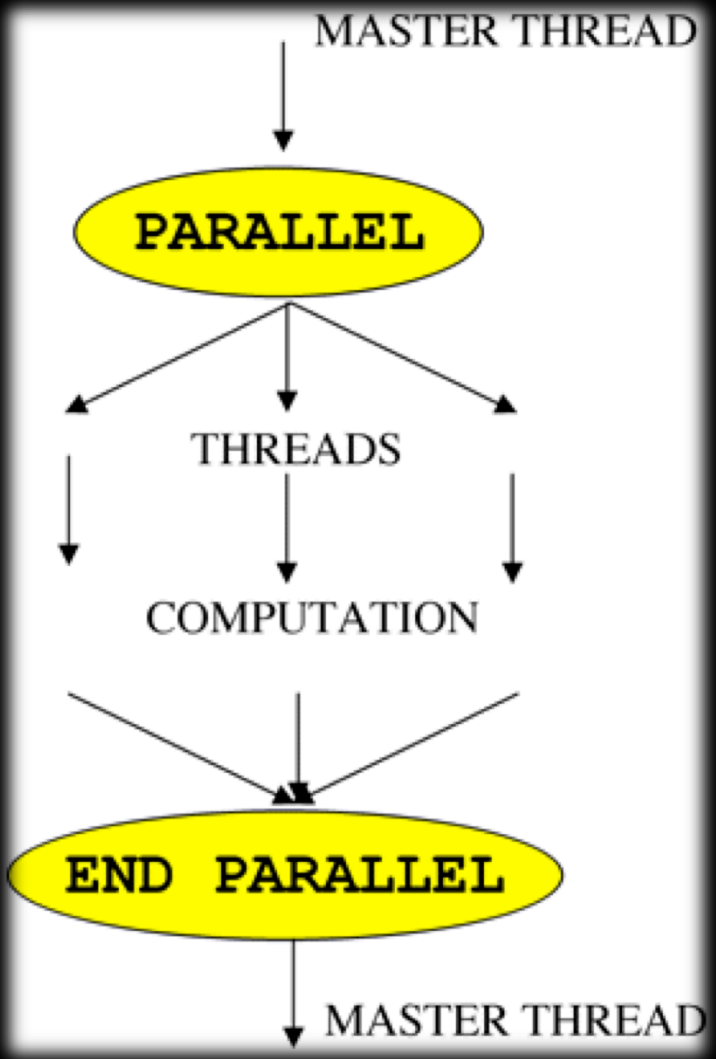
The notion of data ownership is lacking, so there is no need to specify explicitly the communication of data between tasks.

- Program development can often be simplified.

Example: OpenMP

OpenMP

Fork-Join Model



OpenMP

- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors.
- Portable/multi-platform, including Unix and Windows NT platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

OpenMP

Parallel Region

!$omp parallel	#pragma omp parallel

Worksharing

!$omp do	#pragma omp for
!$omp sections	#pragma omp sections
!$omp single	#pragma omp single
!$omp workshare	#pragma omp workshare

Fortran Example

```
program HELLO

 use omp_lib

 integer :: NTHREADS, TID

 ! Fork a team of threads giving them their own copies of variables
 !$OMP PARALLEL PRIVATE(NTHREADS, TID)

 ! Obtain thread number
 TID = OMP_GET_THREAD_NUM()
 print *, 'Hello parallel World from thread = ', TID

 ! Only master thread does this
 !$OMP MASTER
 NTHREADS = OMP_GET_NUM_THREADS()
 print *, 'Number of threads = ', NTHREADS
 !$OMP END MASTER

 ! All threads join master thread and disband
 !$OMP END PARALLEL

end program HELLO
```

C Example

```
/* OpenMP example - Hello World - C/C+ Version */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
int nthreads, tid;

/* Form a team of threads */

#pragma omp parallel private(nthreads, tid)
{

      /*Obtain the thread number */
      tid=omp_get_thread_num();
      printf ("Hello parallel world from thread=%d\n", tid);

      /*Only the master thread does this */
            #pragma omp master
            {
                  nthreads = omp_get_num_threads();
                  printf("Number of threads = %d\n",nthreads);
            }
} /* All threads join with the master and disband */
```

OpenMP

Pros:

- Incremental parallelism
  - can parallelize existing serial codes one bit at a time
- Quite simple set of directives
- Shared data!
- Partitioning operations on arrays is very simple.
- Now supported by gcc.

Cons:

- Requires shared memory multiprocessors
- Shared data!
- Having to think about what data is shared and what data is private
- Generally not as scalable (more synchronization points)

Distributed-Memory Programming

Distributed-Memory Programming

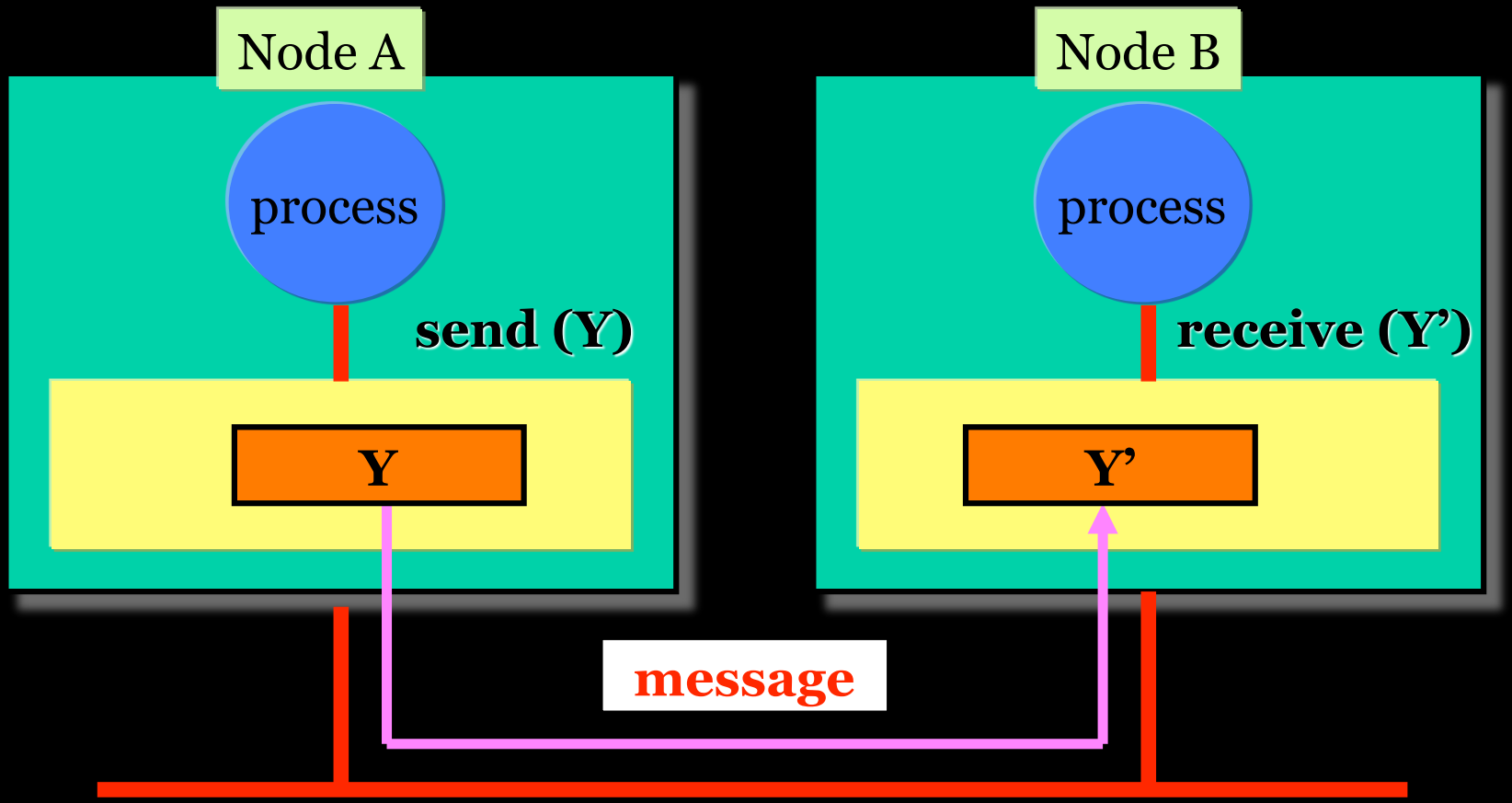
The message passing model demonstrates the following characteristics:

1. A set of tasks that use their own local memory during computation.

3. Tasks exchange data through communications by sending and receiving messages.

5. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Send/Receive



MPI

- A library of subroutines that are embedded in source code. The programmer is responsible for determining all parallelism.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996.
- Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html.

MPI BASICS

MPI has over 125 functions, but you can do much with just 6:

Initialisation for communications:

- **MPI_INIT()**: initializes the MPI environment
- **MPI_COMM_SIZE()**: returns the number of processes
- **MPI_COMM_RANK()**: returns this process's number (rank)

Communicating data between processes:

- **MPI_SEND()**: sends a message
- **MPI_RECV()**: receives a message

Exit in a "clean" fashion when finished communicating:

- **MPI_FINALIZE()**

Fortran Example

```
program HelloWorld

    include 'mpif.h'

    integer :: myrank=0       ! rank of my process
    integer :: ierr,np=1      ! number of processes

    call MPI_INIT( ierr )

    call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)
    print *, 'Hello World! I am process', myrank, ' of ', np

    call MPI_FINALIZE(ierr)

end program HelloWorld
```

C Example

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
  int myrank;     /* rank of my process */
  int np;         /* number of processes */

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&np);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  printf("Hello World! I am process %d of %d\n", myrank, np);

  MPI_Finalize();
}
```

MPI

Pros:

- Very portable
- Requires no special compiler
- Requires no special hardware but can make use of high performance hardware
- Very flexible - can handle just about any model of parallelism
- No shared data! (You do not have to worry about processes "treading on each other's data" by mistake.)
- Can download free libraries for your Linux PC!

MPI

Challenges:

- All-or-nothing parallelism (difficult to incrementally parallelise existing serial codes)
- No shared data! Requires distributed data structures
- You generally have to write more code
- Partitioning operations on distributed arrays can be messy.

Summary

Pure MPI Advantages

1. Portable to distributed and shared memory machines.
2. Scales beyond one node
3. No data placement problem

Pure OpenMP Advantages

1. Easy to implement parallelism
2. Lower latency, high bandwidth
3. Implicit communication
4. Coarse and fine granularity
5. Dynamic load balancing

Pure MPI Disadvantages

1. Difficult to develop and debug
2. Higher latency, low bandwidth
3. Explicit communication
4. Difficult load balancing

Pure OpenMP Disadvantages

1. Only on shared memory machines
2. Scale within one node
3. Possible data placement problem
4. No specific thread order

Conclusion

There are a number of parallel programming models available.

Consider the trade-offs:

- Potentially faster execution turnaround
- Longer software development time
- Obtaining machine access
- Cost = f(development) + g(execution time)
- Hardware – need to understand machine architecture

Do the benefits out-weigh the costs?

Parallel Performance

Speedup

The goal of *speedup* is to use P processors to make a program run P times faster

Speedup is the factor by which the program's speed improves:

$$\text{Speedup}(p \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(p \text{ processors})}$$

Speedup

The execution time depends on what the program does!

A parallel program spends time in:

- Work
- Synchronization
- Communication
- Extra work (overheads)

A program implemented for a parallel machine is likely to do extra work (than a sequential program) even when running on a single processor machine!

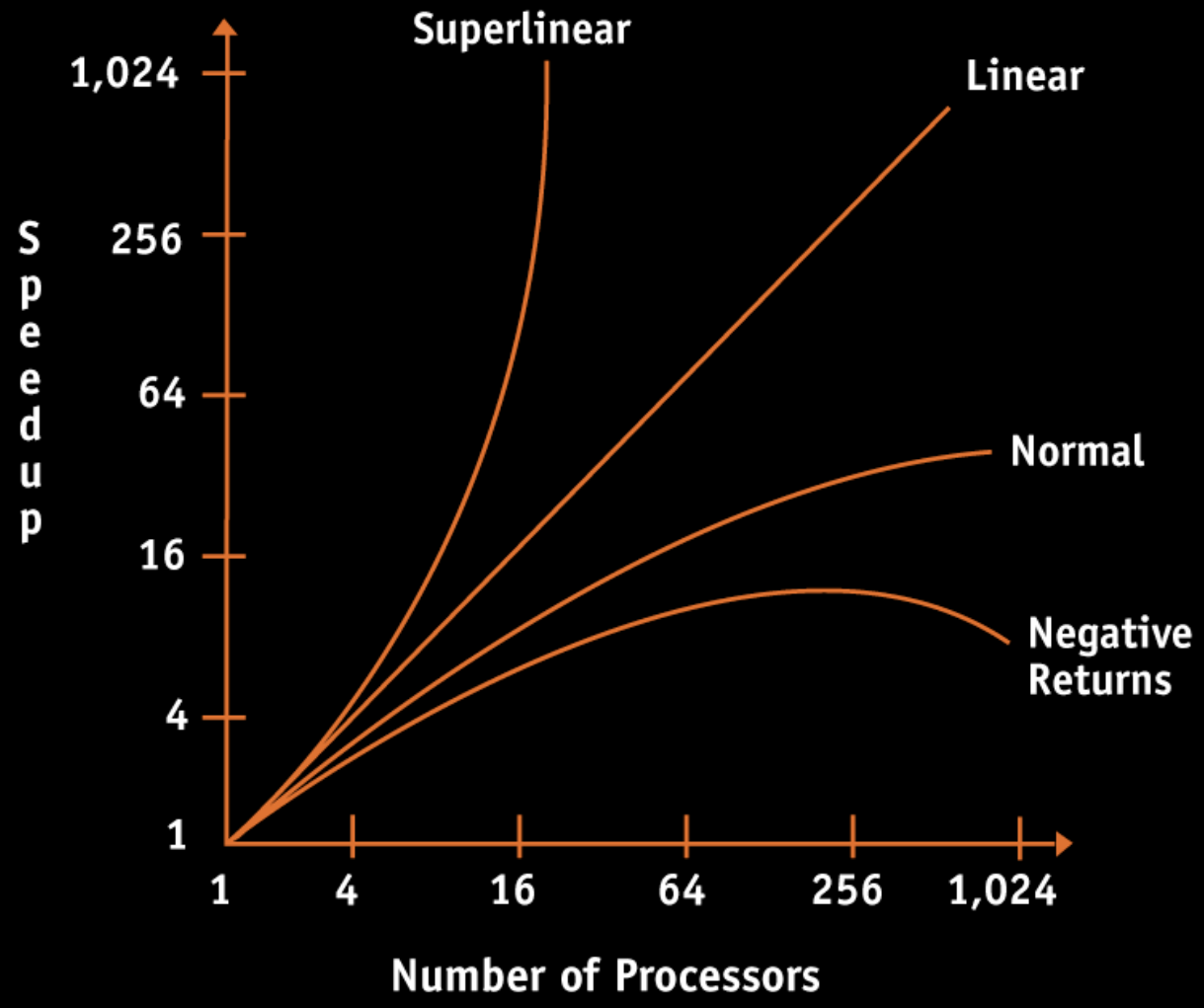
Speedup

$$\text{Relative Speedup}(p \text{ processors}) = \frac{\text{Time}(\text{Par. Alg., 1 processor})}{\text{Time}(\text{Par. Alg., } p \text{ processor})}$$

$$\text{Absolute Speedup}(p \text{ processors}) = \frac{\text{Time}(\text{Seq. Alg., 1 processor})}{\text{Time}(\text{Par. Alg., } p \text{ processor})}$$


LINKSCEEM

Speedup


Superlinear
Linear
Normal
Negative Returns
1,024
256
64
16
4
1
Speedup
1
4
16
64
256
1,024
Number of Processors

Superlinear Speedup

Sometimes P processors can achieve a speedup > P

- usually the result of improving an inferior sequential algorithm
- can legitimately occur because of cache and memory effects
  - More processors typically also provide more memory/cache.
  - Total computation time decreases due to more page/cache hits.

Question

Programs A and B solve the same problem using different algorithms:

Facts:

- both are run on a 100-processor computer
- program A gets a 90-fold speedup
- program B gets a 10-fold speedup

Which one would you prefer to use?

Answer

All that matters is the overall execution time

- what if A runs sequentially 1,000 times slower than B?
- always use the *best sequential time* (over all algorithms) for computing speedups!
- and the best compiler!

Calculation of π

This program calculates *pi* using a simple integral approximation:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

We approximate by summing from k=1 to N of 4/(1 + (k-1/2)**2) .
The only input data required is N.

The code is of particular interest, as we examine a

- Sequential Version
- OpenMP Version

Template Serial Code

```
program calc_pi
    implicit none
    integer n,i
    double precision :: w,x,sum,pi,f,a,start, finish, timef

    f(a) = 4.0 / (1.0 + a*a)
    n=100000000

    start=timef()

    w=1.0/n
    sum=0.0

    do i=1,n
      x = w * (i - 0.5)
      sum = sum + f(x)
    end do

    pi = w * sum

    finish=timef()
    print*,"value of pi, time taken:"
    print*,pi,finish-start
end
```

Parallel Strategy

Parallel strategy:

- break the loop into portions which can be executed by the processors.

For the task of approximating pi (with *P* processers):

- each processor executes its portion of the loop N/P times.
- each processor does work without requiring any information from the other processors
- there are no data dependencies
- no communications except at end
- this is known as Embarrassingly Parallel (EP)

Embarrassingly parallel

- Computationally intensive
- Minimal or no communication
- Minimal I/O

Parallel Code

```
program calc_pi
    implicit none
    integer n,i
    double precision w,x,sum,pi,f,a,start, finish, timef

    ! f(a) = 4.0 / (1.0 + a*a)
    n=100000000
    start=timef()
    w=1.0/n
    sum=0.0
!$OMP PARALLEL PRIVATE(x,i), SHARED(w,n), &
!$OMP REDUCTION(+:sum)
!$OMP DO
    do i=1,n
      x = w * (i - 0.5)
      sum = sum + f(x)
    end do
!$OMP END DO
!$OMP END PARALLEL
    pi = w * sum
    finish=timef()
    print*,"value of pi, time taken:"
    print*,pi,finish-start
 end
```

Execution

To run the program on N threads we enter either of the following, depending on the login shell:

(sh or bash shell)

```
export OMP_NUM_THREADS=N
./pi.out
```

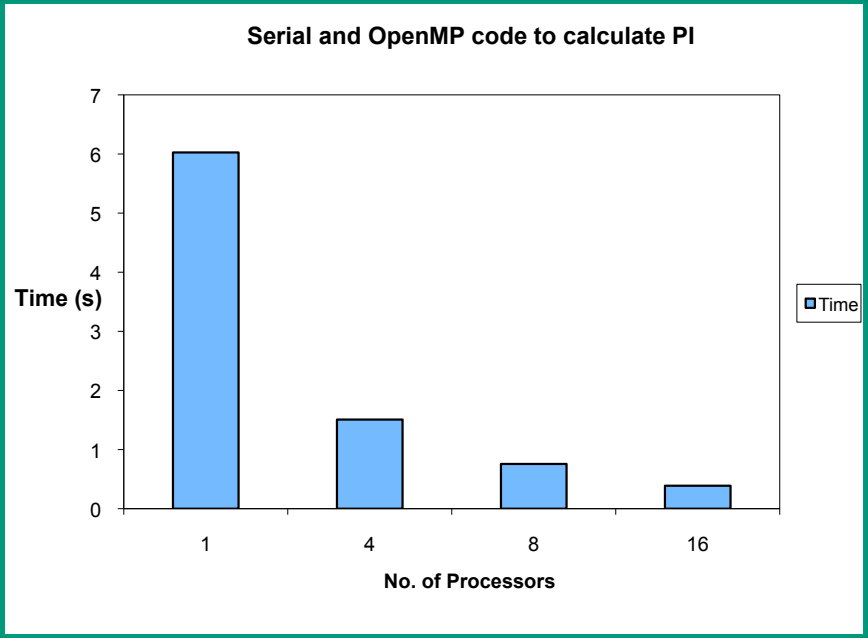
or (csh shell)

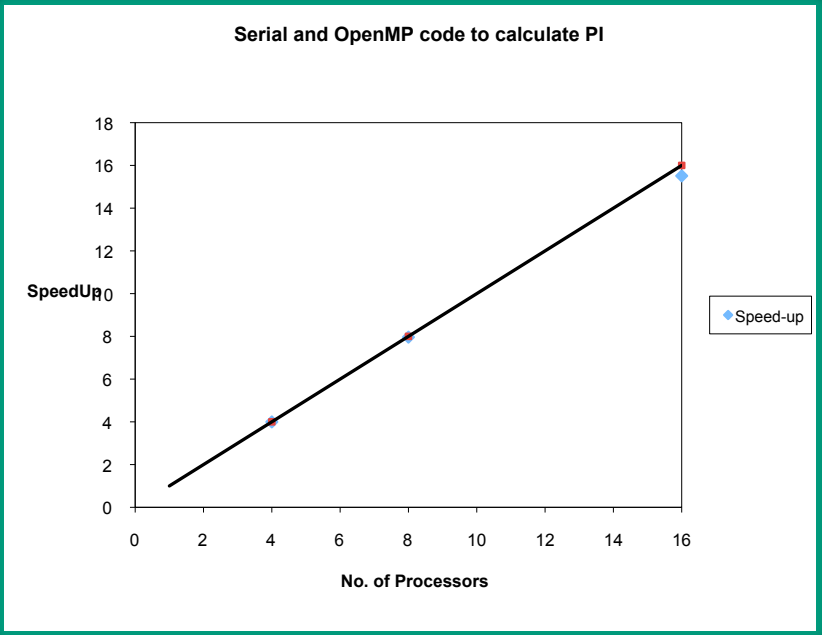
```
setenv OMP_NUM_THREADS N
./pi.out
```

On N processors, you should get a speed up close to N, and thus an efficiency close to 100%.

Execution

The graph below illustrates the results we obtained - the timings taken, in milliseconds - to calculate pi using 100 million iterations.





Limits to Speedup?

All parallel programs contain:

- Parallel regions
- Serial regions

Serial sections limit the parallel performance

Practical limits

- Communication overhead
- Synchronization overhead
- Extra operations necessary for parallel version

Other Considerations

- Time used to re-write (existing) code

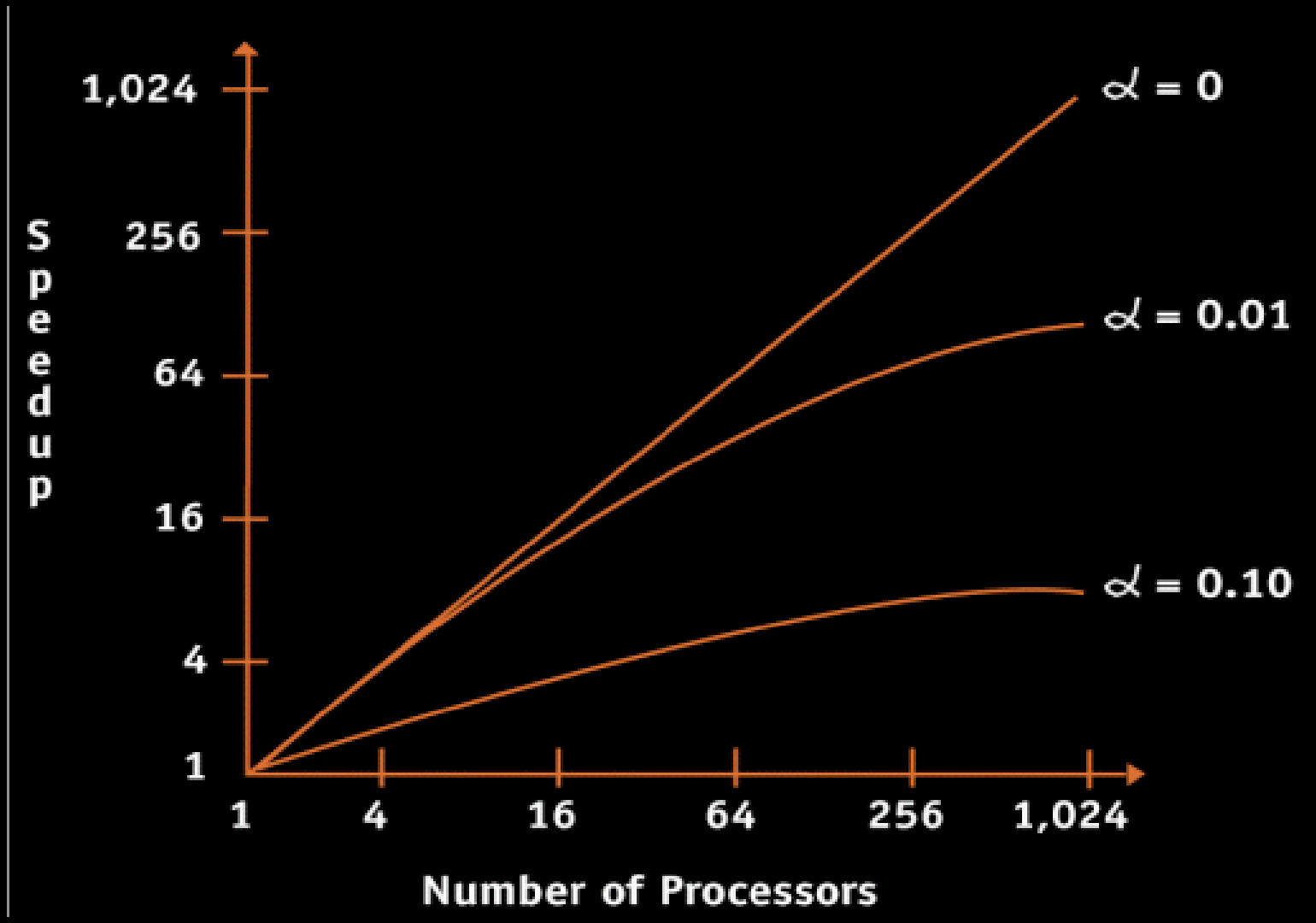
Limits to Speedup?

Amdahl's Law is a law governing the maximum *speedup* of using parallel processors on a problem, versus using only one serial processor.

Let P represent the fraction of the program that is parallel.

$$speedup = \frac{1}{(1-P) + \frac{P}{N}} \Rightarrow \lim_{N \to \infty} \frac{1}{1-P}$$

Amdahl's Law


1,024
256
64
16
4
1
Speedup
α = 0
α = 0.01
α = 0.10
1
4
16
64
256
1,024
Number of Processors

Scalability

Assume a problem of size *P* using *N* processors takes time *T*

Strong Scaling

As P remains fixed, αN processors solve the problem in T/α time.

Weak Scaling

For a problem of size αP, αN processors solves the problem in time T.

Real Performance

Peak Performance is skyrocketing

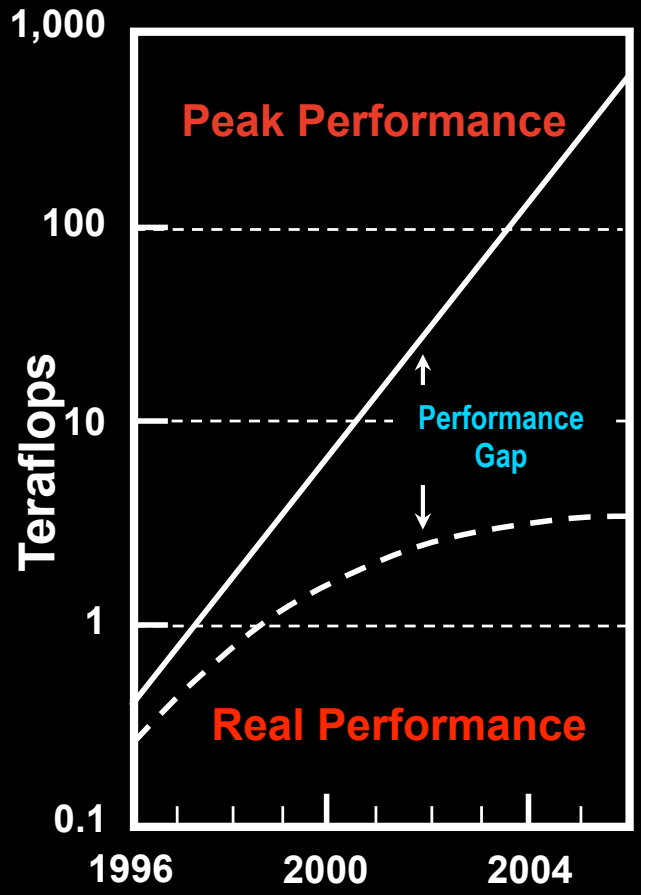
- In 1990's, peak performance increased 100x; in 2000's, it will increase 1000x

But efficiency (the performance relative to the hardware peak) has declined

- was 40-50% on the vector supercomputers of 1990s now as little as 5-10% on parallel computers of today

Close the gap through ...

- Mathematical methods and algorithms that achieve high performance on a single processor and scale to thousands of processors. More efficient programming models and tools for massively parallel supercomputers



Performance Considerations

PARALLEL PERFORMANCE FACTORS

LOAD BALANCING

DATA DEPENDENCY

GRANULARITY

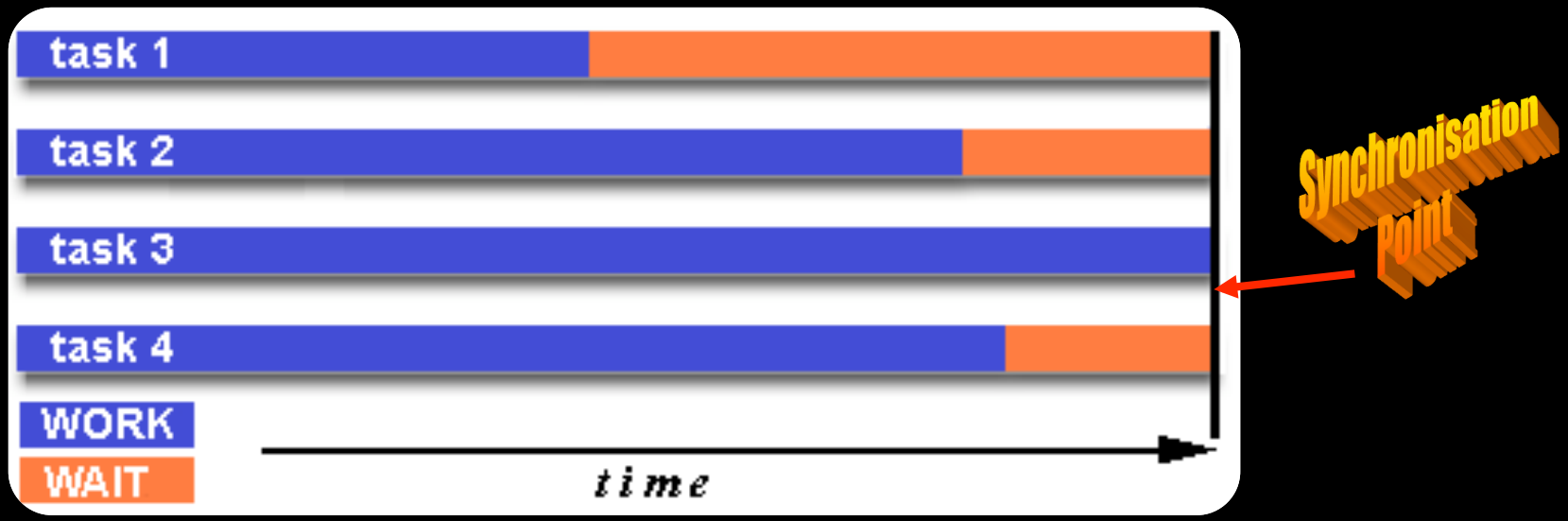
SYNCHRONISATION

COMMUNICATION PATTERNS

LOAD BALANCING

Load balancing refers to the practice of distributing work among processes so that ***all*** are kept busy ***all*** of the time. It can be considered a minimization of process idle time.

For example, if all processes are subject to a barrier synchronization point, the slowest process will determine the overall performance.



OBTAINING LOAD-BALANCE

Load balance is dependent on good data decomposition i.e. dividing up the problem data so that each processor has an even distribution of work to compute.

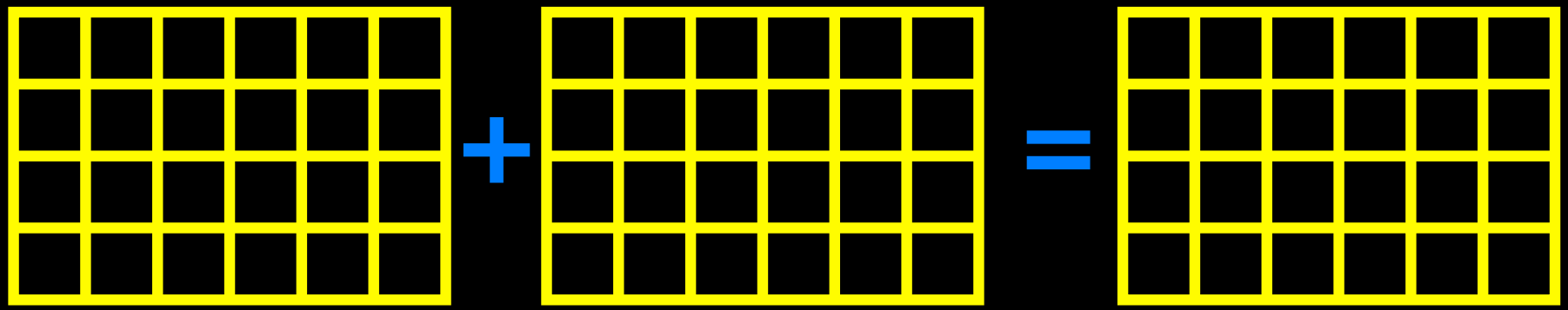
Data Decomposition Strategies:

1. Static Decomposition
2. Dynamic Decomposition

STATIC DECOMPOSITION

Evenly partition the work each process receives

e.g. some array/matrix calculations (also image operations) can be statically decomposed so that each processor can be given equal work to perform



STATIC DECOMPOSITION

Similarly for loop iterations, where the work done in each iteration is similar, evenly distribute the iterations across the processors.

Original Loop

```
do 1 = 1, K
   call some_work()
end do
```

MPI Version

```
Start = K/P * rank
End = K/P * (rank + 1)-1

do 1 = Start, End
   call some_work()
end do
```

OpenMP Version

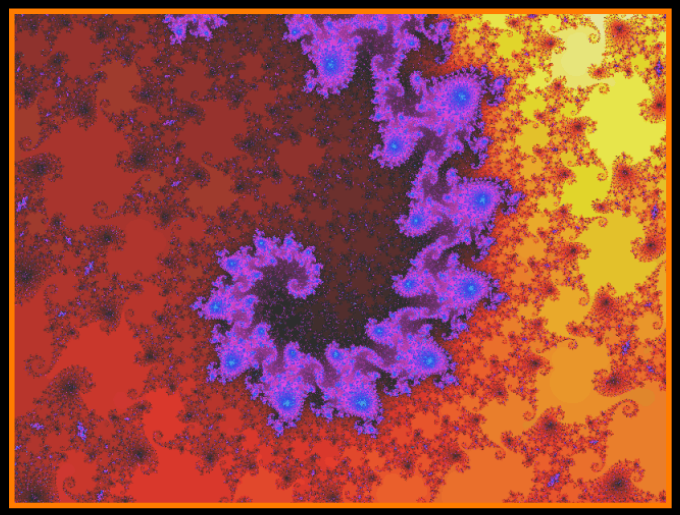
```
!$OMP PARALLEL DO
do 1 = Start, End
   call some_work()
end do
!$OMP END PARALLEL DO
```

DYNAMIC DECOMPOSITION

Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:

Some examples:

- Sparse Matrices
- N-Body Simulations
- Mandelbrot Set etc…



DYNAMIC DECOMPOSITION

When the amount of work each task will perform is intentionally variable, or unpredictable, it may be helpful to use a *scheduler - task pool* approach.

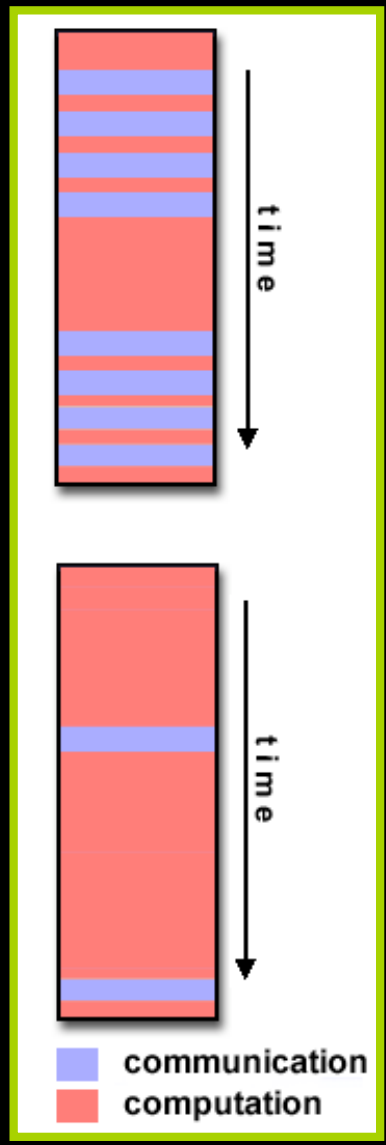
As each task finishes its work, it queues to request a new piece of work.

It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

GRANULARITY

Granularity is the relationship between the amount of computation and communication.

It is a measure of how much work gets done before processes have to communicate.



GRANULARITY

Does maximum parallelization mean maximum speed-up?

i.e. if N processors are available should the program be spilt into N tasks?

Requires:

- effort to decompose problem
- communication overheads passing data between processes and synchronising processes - communication is disproportionately slow compared to computation

Trade-off:

grain-size against efficiency and execution time.

- Reducing grain-size may reduce execution-time but be less efficient.

COMMUNICATION FACTORS

For some problems, increasing the number of processors will:

1. Decrease execution time attributable to computation
2. May increase execution time attributable to communication

Time required for communication is dependent upon system communication bandwidth parameters.

The time (T) required to send W words between any two processors is given by:

$$T = L + W/B$$

where

L = Latency (time required to send a zero byte message)

B = Bandwidth (W/s)

DATA DEPENDECY

A data dependency exists when there is multiple access to the same storage location. They frequently inhibit parallel execution.

```
DO J = MYSTART,MYEND
        A(J) = A(J-1) * 2.0
CONTINUE
```

This code has a data dependency: A(J-1) must be computed before A(J).

- How does this affect an OpenMP DO directive?

If processor 2 has A(J) and processor 1 has A(J-1), the value of A(J) is dependent on:

Distributed memory:

- Process A obtaining the value of A(J-1) from Process B

Shared memory:

- Whether Process 2 reads A(J-1) before or after Process 1 updates it

LOOP UNROLLING

- The iterations of an inner loop are decreased by a factor of two or more by explicit inclusion of the very next one or several iterations.
- Loop unrolling can allow traditional compilers to make better use of the registers and to improve overlap operations.
- Improves speed due to decreased number of condition tests and branching, at the expense of the code size.

LOOP UNROLLING

Consider the following loop:

```
DO I=1,N
        A(I) = foo()
END DO
```

By unrolling the loop to depth 4 we can perform 4 independent operations in parallel (using 4 registers) as well as reduce the number of code branches and increments of variable I.

```
DO I=1,N,4
        A(I) = foo()
        A(I+1) = foo()
        A(I+2) = foo()
        A(I+3) = foo()
END DO
```

DEADLOCK

DEADLOCK

Deadlock:

Two or more processes are waiting for an event or communication from one of the other processes before they can proceed.

Example:

Two processes - both programmed to read/receive from the other before writing/sending:

TASK1	TASK2
X = 4	Y = 8
SOURCE = TASK2	SOURCE = TASK1
RECEIVE (SOURCE,Y)	RECEIVE (SOURCE,X)
DEST = TASK2	DEST = TASK1
SEND (DEST, X)	SEND (DEST, Y)
Z = X + Y	Z = X + Y

DEADLOCK

Possible Solution:

One solution is to change the order of the SEND and RECEIVE in one of the tasks.

```
PROCESS1
----------
X = 4
SOURCE = PROCESS2
RECEIVE (SOURCE,Y)
DEST = PROCESS2
SEND (DEST, X)
Z = X + Y
```

```
PROCESS2
----------
Y = 8
DEST = PROCESS1
SEND (DEST, Y)
SOURCE = PROCESS1
RECEIVE (SOURCE,X)
Z = X + Y
```

Another: *NON-BLOCKING message passing*:

- call returns immediately after call is initiated
- does not wait to be certain that communication buffer is safe to use.

SUMMARY

Parallelism overheads include:

- cost of starting a thread or process
- cost of communicating shared data
- cost of synchronizing
- extra (redundant) computation

Each of these can be in the range of milliseconds (= millions of arithmetic ops) on some systems

Tradeoff:

Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work.

TASK DECOMPOSITION

As well as data decomposition, tasks can also be distributed for parallel execution.

Two types of task decomposition are:

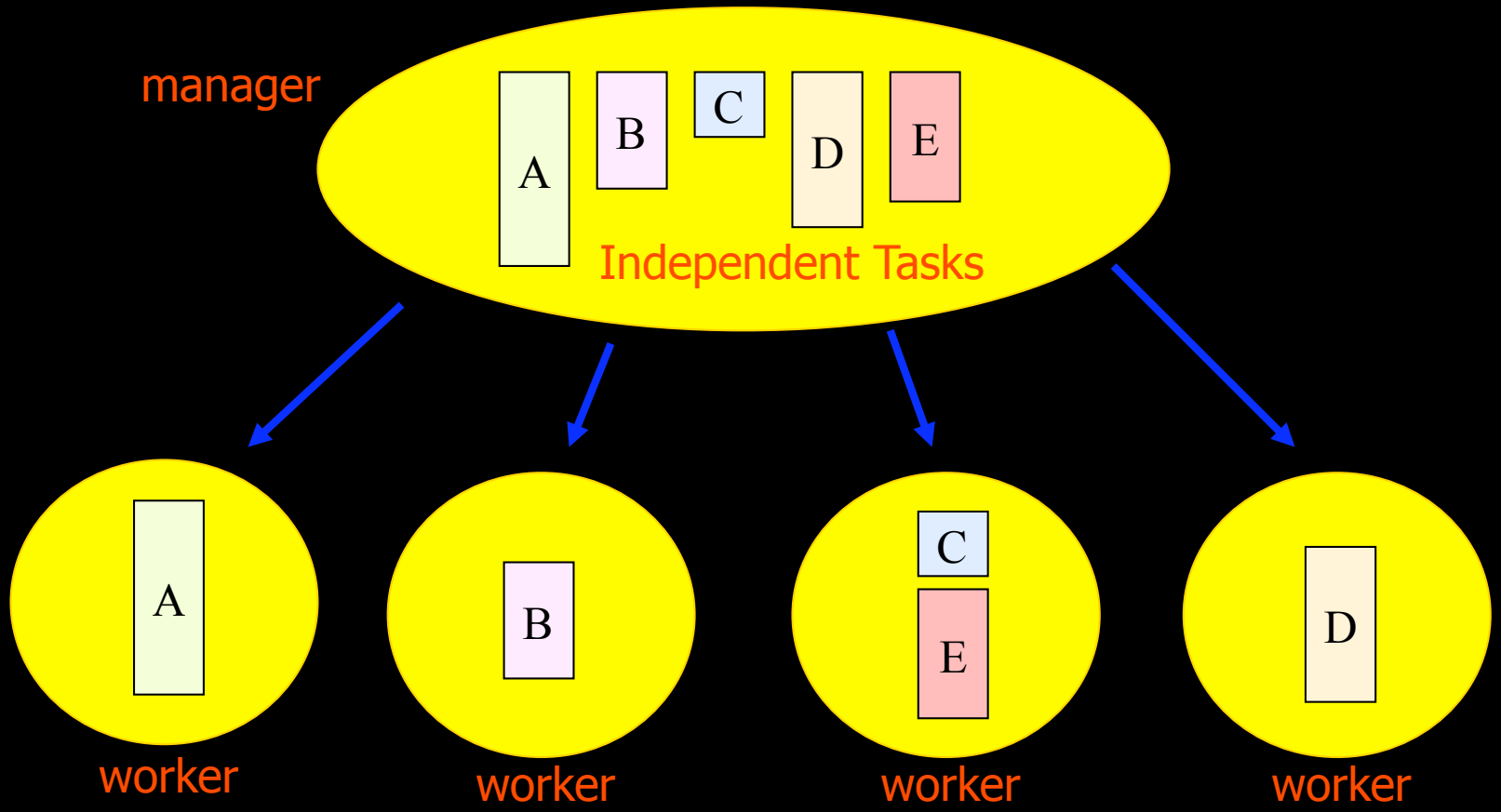
Trivial Task Decomposition

Functional Decomposition (Pipelining)

TRIVIAL TASK DECOMPOSITION

Also called embarrassingly parallel computations.

- Calculation consists of independent tasks



Pipelining

If a repeatable task can be decomposed into dependent subtasks then a pipelining strategy could be advantageous; consider a task T that is composed of dependent subtasks A,B,C & D.