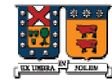
Alfa-SCAT
Scientific Computing Advanced Training



UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Concepts in Parallel Computing

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
Departamento de Física

Dr Mike Ashworth
Computational Science & Engineering Dept
CCLRC Daresbury Laboratory &
HPCx Terascaling Team

a project funded by EuropeAid

Outline

PART I

- Serial Computing
- Parallel Computing – The Hardware View
  - different architectures
- Parallel Computing – The Logical View
  - architecture independent views
  - programming environments
- Parallel Computing – Performance
  - how do we measure performance and parallel scaling?

PART II

- Parallel Computing – Overheads
  - why don't we get perfect scaling?
- Parallel Computing – Design
  - how does knowing all the above help us write a parallel code?

Disclaimer

- Parallel computing is usually the subject of a course running over (at least) one semester
- This will be a very broad overview of the subject in two lectures

I will go quickly ...

... there will be material missing ...

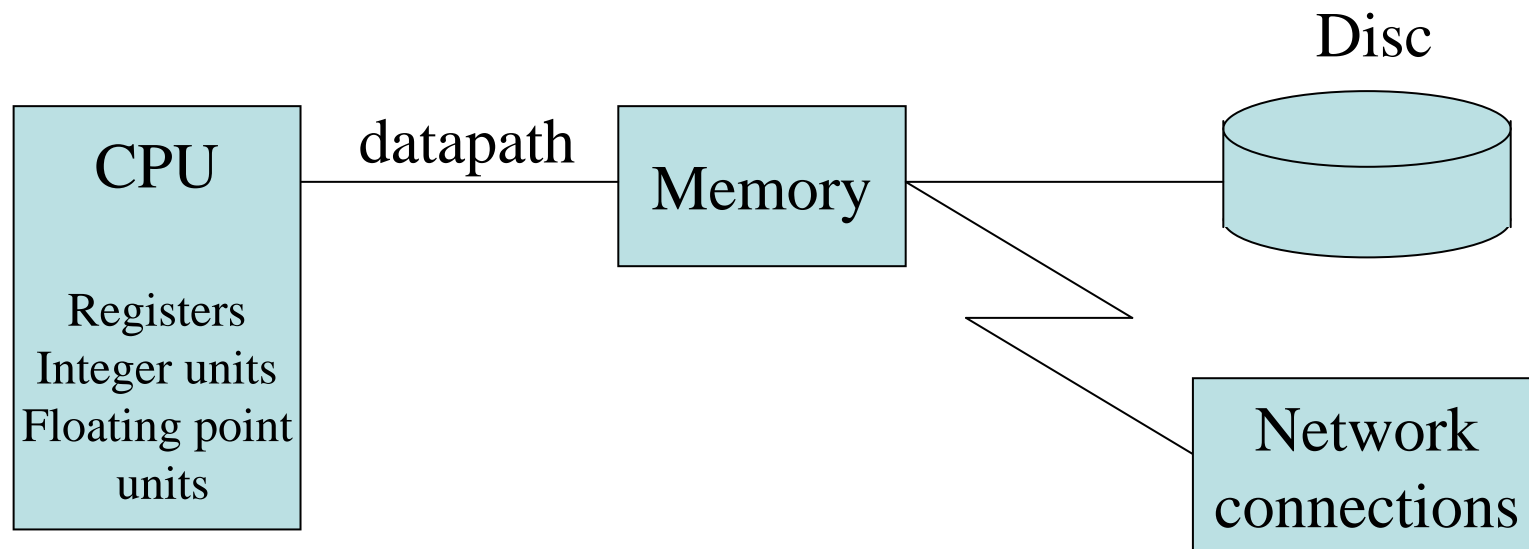
... but hopefully it will give you a useful introduction

- You should follow up with in-depth courses or self-study

Serial Computing

Serial Computing I

- The elements of conventional architectures which limit the performance of scientific computing are the processor (CPU), memory system, and the datapath

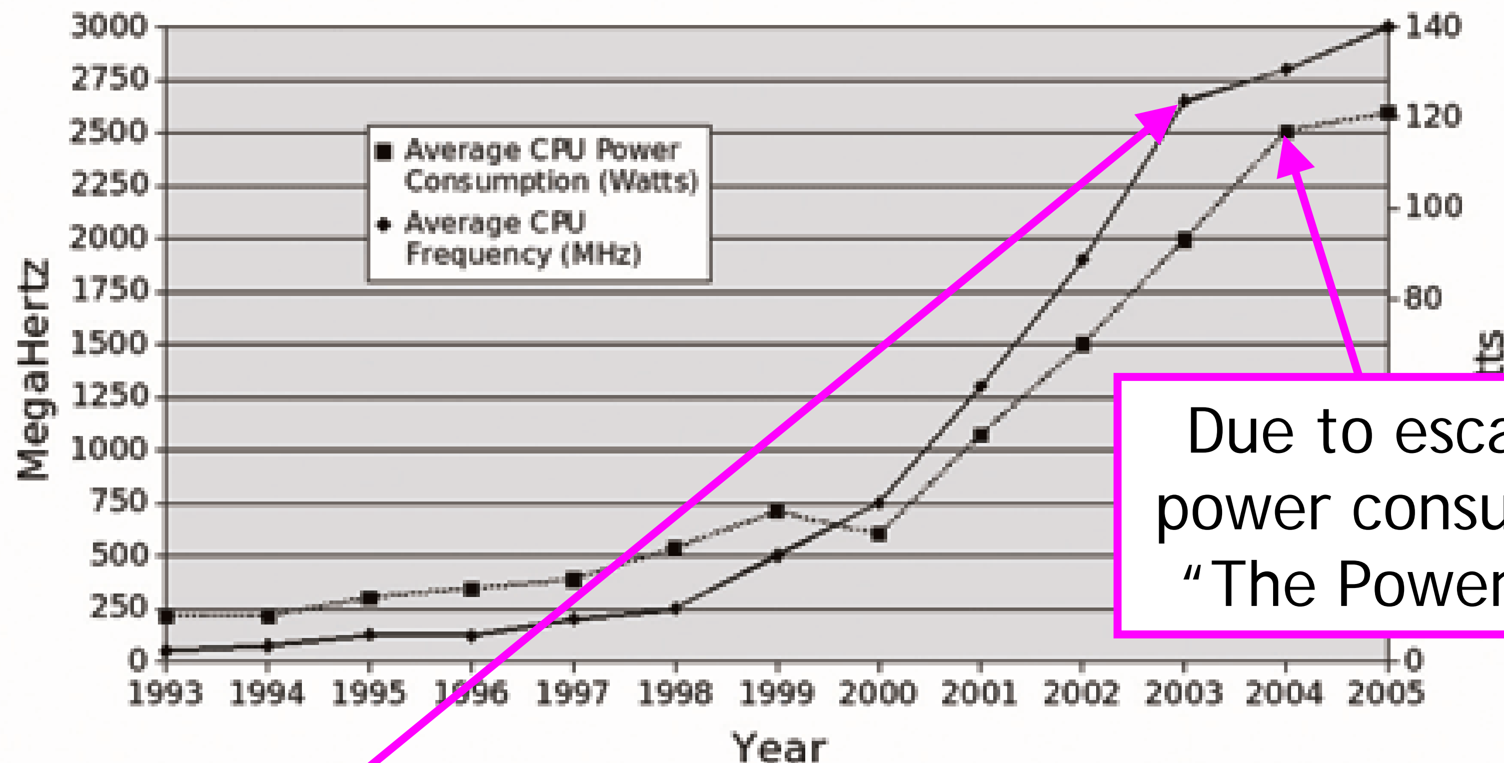


Serial Computing II

- Different applications have different requirements
  - data-intensive applications require high data throughput
  - server applications require high data network bandwidth
  - scientific applications require high processing and memory system performance
- For scientific computing the three components, processor, memory system, and the datapath, each present significant performance bottlenecks
- It is important to understand each of these performance bottlenecks
- Parallel computing addresses each of these components in significant ways
- Parallelism is present at many difference levels (granularity: fine grain -> coarse grain)

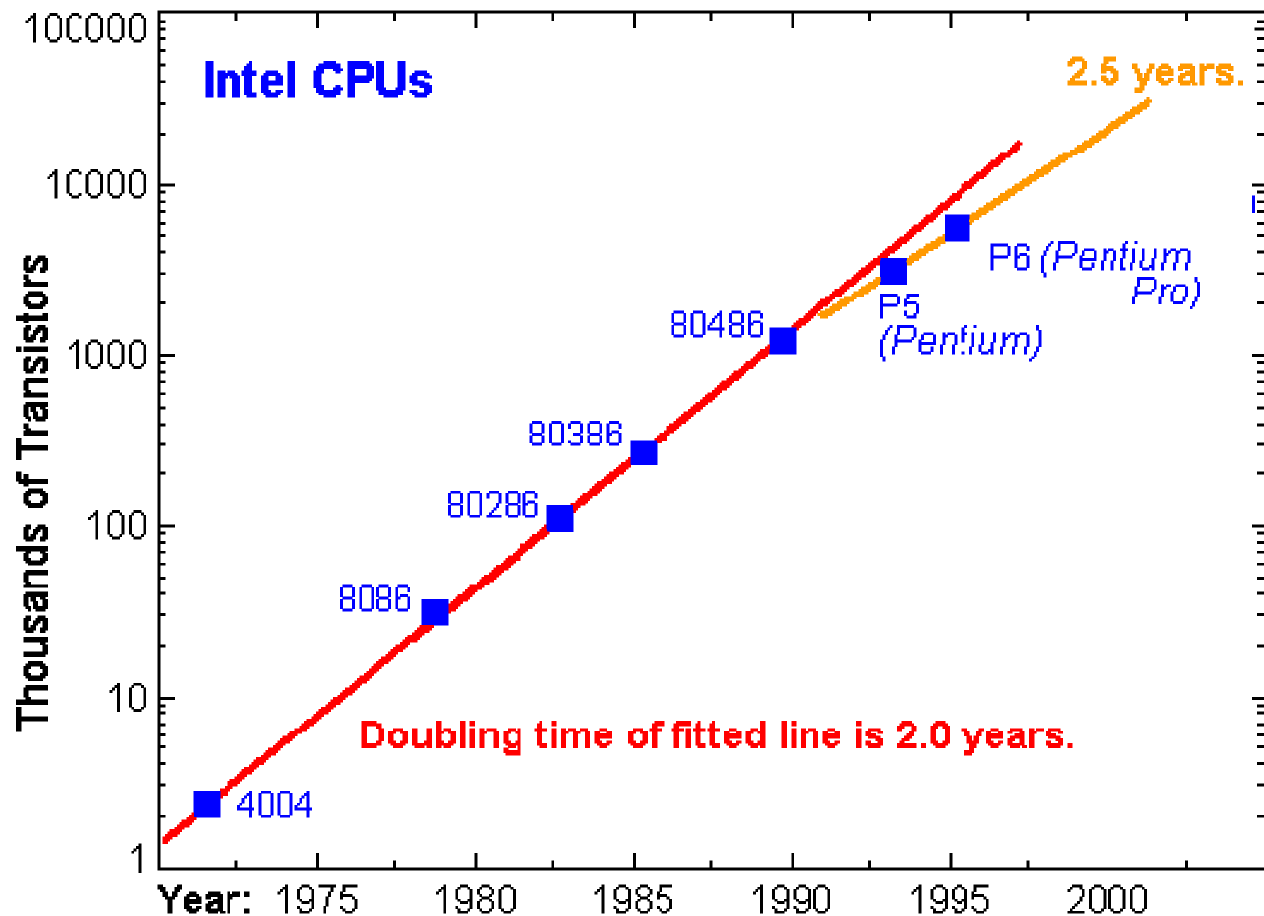
Microprocessor Clock Speeds

- Microprocessor clock speeds have increased dramatically (three orders of magnitude in two decades)



Moore's Law

- Higher levels of device integration have made available increasing numbers of transistors (Moore's Law)



Chip “Real Estate”

- The question of how best to utilize more transistors is an important one
- Up until recently processors use these resources in
  - multiple functional units (increased fine grain parallelism)
  - larger on-chip memory caches for instructions and data
- Now we are seeing “multi-core” chips with multiple CPUs per chip
  - dual-core now common
  - 4-way and 8-way imminent

a project funded by EuropeAid

Parallelism within the CPU

- All processors now use parallelism within the CPU
- Pipelined functional units allow repeated operations to be streamed like a production line
- Additional hardware allows the execution of multiple instructions in the same cycle
- The precise manner in which instructions are selected and executed provides for diversity in architectures
  - vector processing
  - pipelining
  - super-pipelining (pipelining with more stages)
  - superscalar (instruction-level parallelism)
  - VLIW (complex compile-time analysis) - superceded
  - out-of-order execution
  - speculative execution and branch prediction

a project funded by EuropeAid

Multi-processor parallelism

- But we are interested in harnessing multiple processors to increase the performance of scientific applications:

1. Reduce the time to solution for existing problems
2. Utilise large-memories for large-scale problems which can not be addressed on single-processor systems

a project funded by EuropeAid

Parallel Computing - The Hardware View

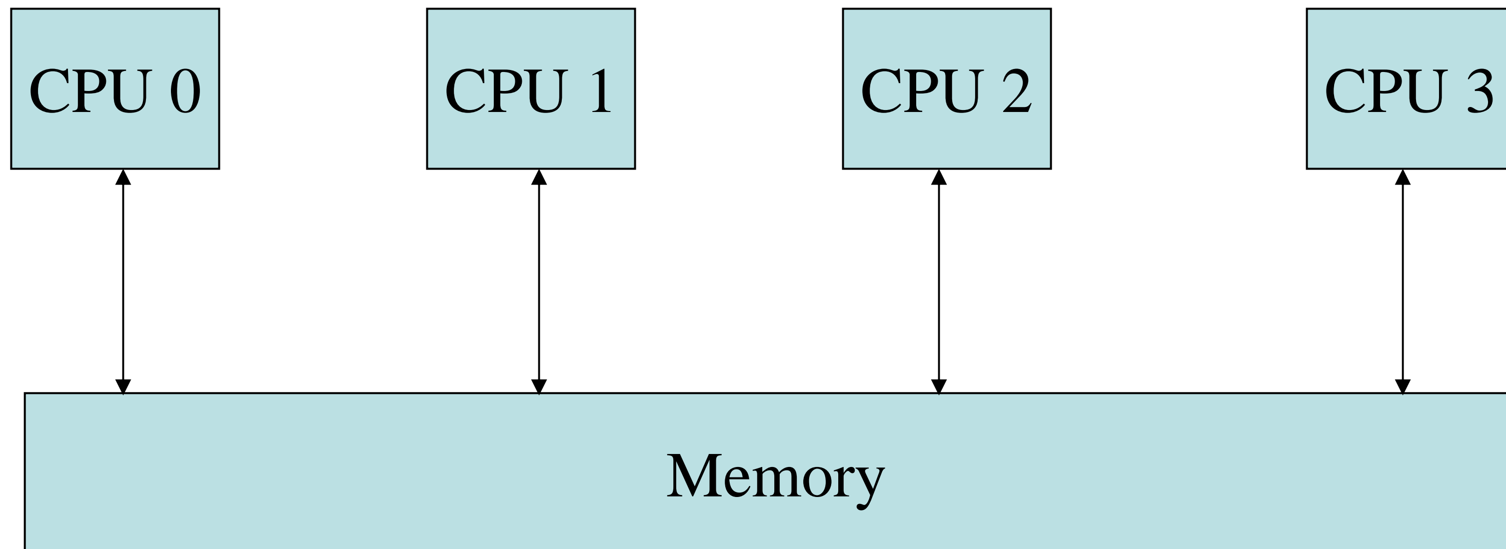
a project funded by EuropeAid

Flynn's Taxonomy

Old (1966) classification of hardware according to instruction streams and data streams:

- SISD – Single Instruction Single Data
  - this is our traditional familiar serial processor
- SIMD – Single Instruction Multiple Data
  - array processors executing a single instruction stream simultaneously in lock-step on different data (successful in the past, not common now)
- MISD – Multiple Instruction Single Data
  - redundant parallelism, as for example on airplanes that need to have several backup systems in case one fails
- MIMD – Multiple Instruction Multiple Data
  - most flexible, allows for different data to be handled in different ways – most modern machines are of this type

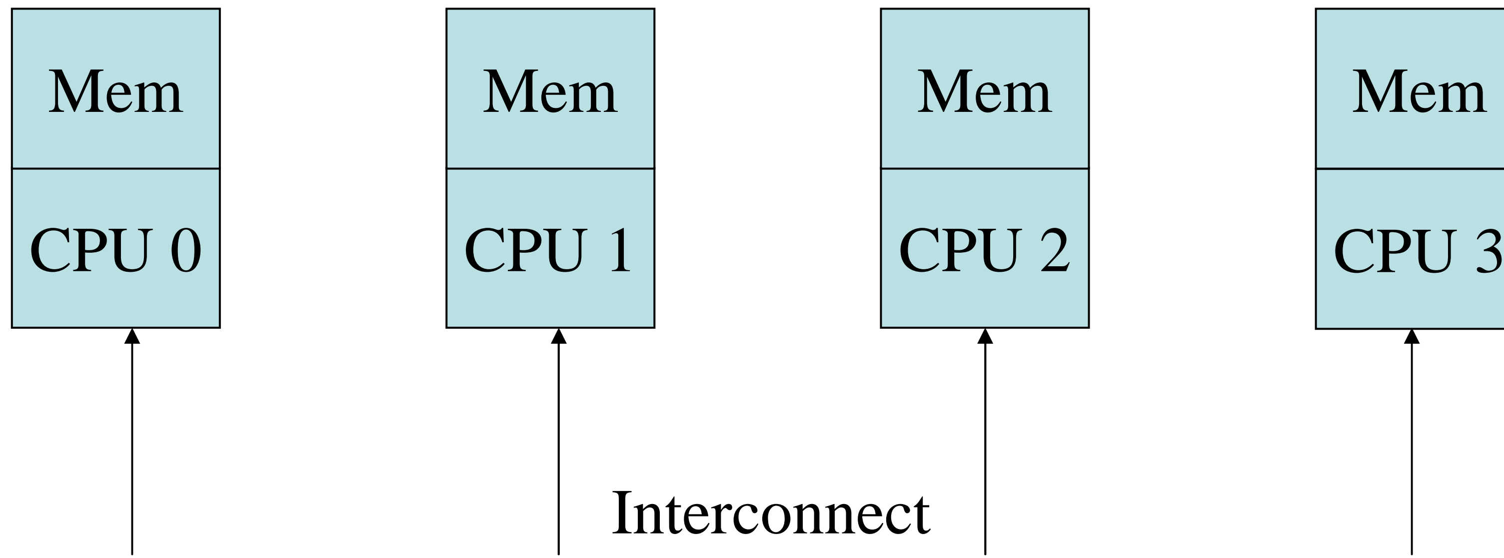
Shared Memory



Found in a dual-core PC, and also some mid-range servers.
No longer used at the high-end as contention in the shared memory limits the scalability

Sometimes referred to as Symmetric-Multi-Processor (SMP)

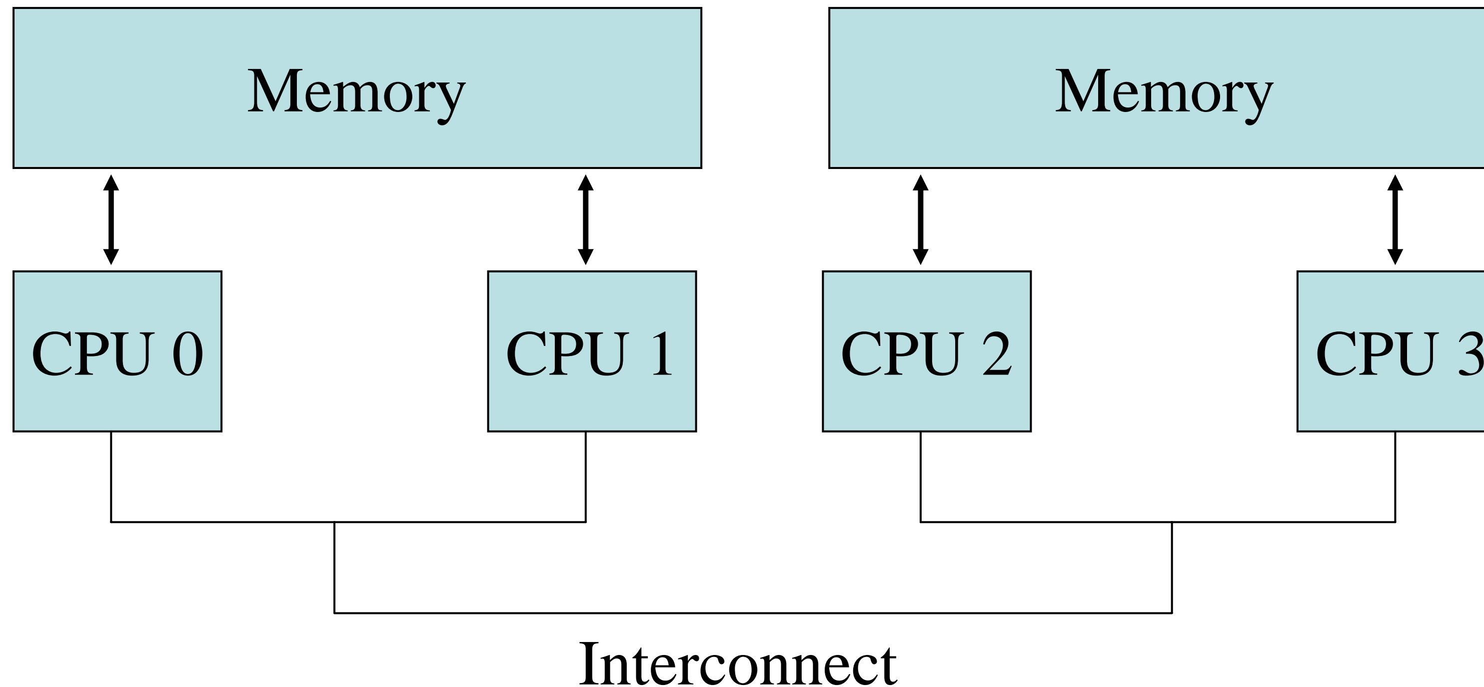
Distributed Memory



Found everywhere from a cluster of PCs to purpose-built high-end systems e.g. Cray XT3.

Performance and scalability depends on the interconnect of which there are many different types.

Clustered Shared Memory Nodes



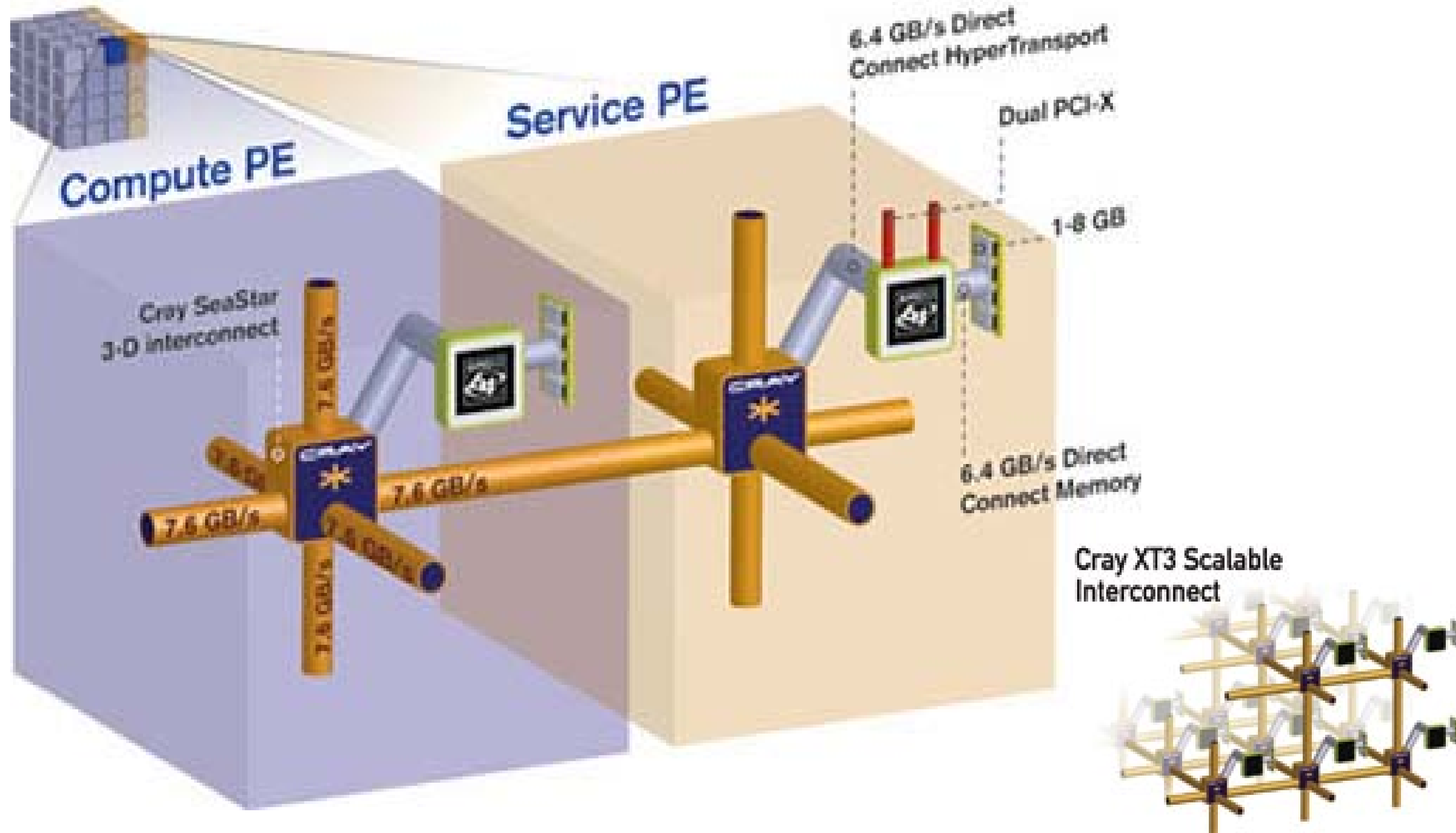
Most high-end systems are now like this e.g. Cray XT4, IBM POWER5 cluster. Also many mid-range clusters with dual-core nodes.

Performance and scalability depends on the interconnect.

Interconnects

- Static or Dynamic
  - Static: point-to-point links, does not scale
  - Switched networks, cost grows as the square of the number of ports
- Network interface
  - I/O bus: loosely-coupled cluster
  - Memory bus: tightly-coupled multi-processor, FASTER
- Network Architectures
  - Bus: poor scalability, but performance improved with local cache
  - Crossbar: full connectivity, expensive to scale to large numbers
  - Multistage: compromise solution
- Network Topologies
  - star, linear array, hypercube, mesh (2D, 3D, toroidal), tree (fat)
  - some systems have multiple networks (e.g. IBM Blue Gene)

Example – Cray XT3

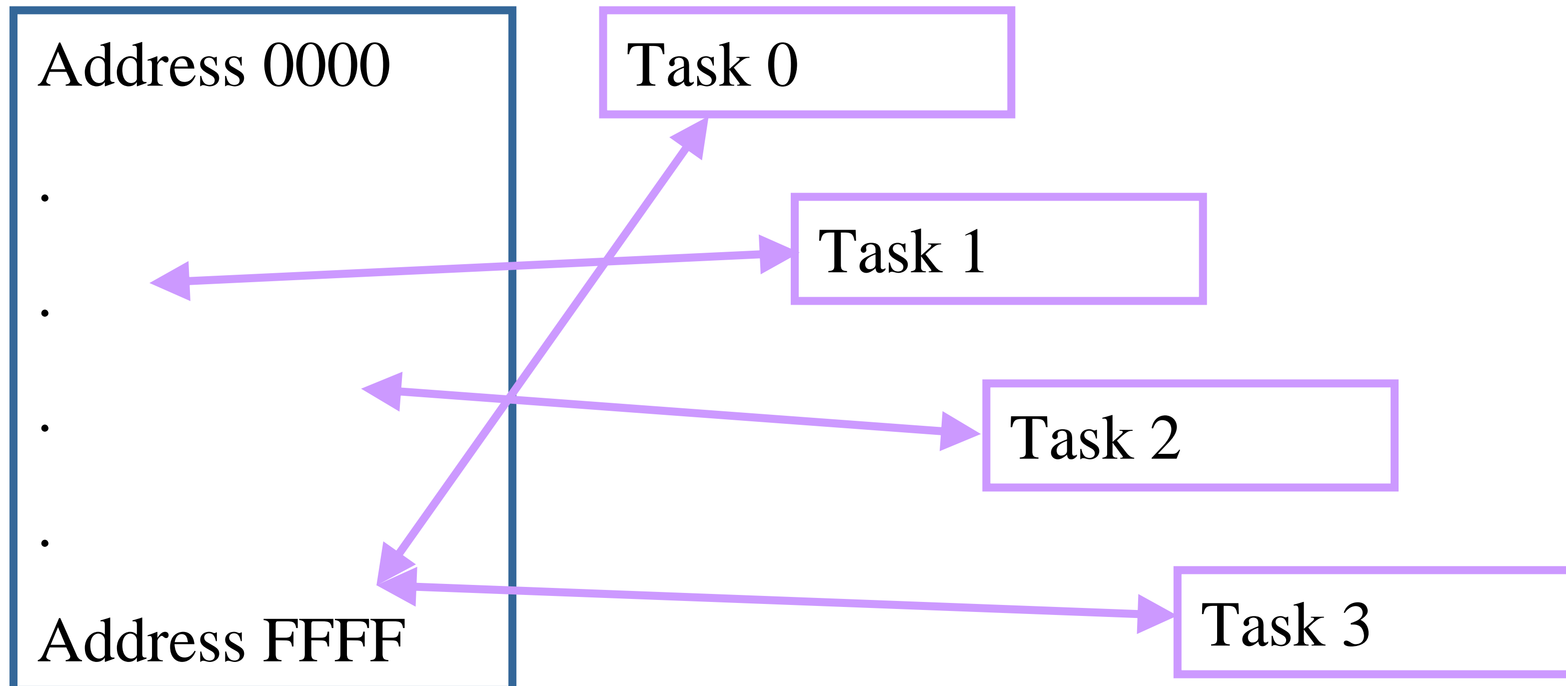


CRAY
XT3

408
407

Parallel Computing - The Logical View

Shared Address Space

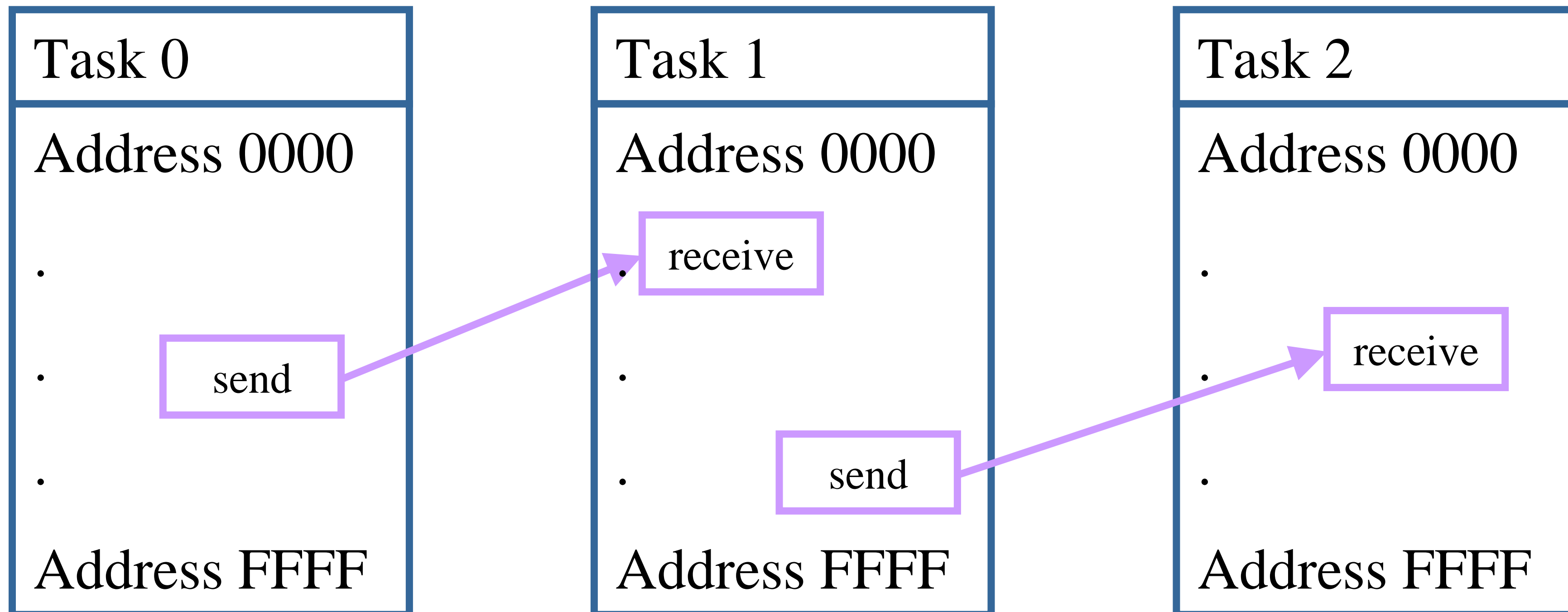


All instances of a program can access the same data. Need to prevent conflict (cache coherency).

Trivial on shared-memory hardware.

Can be implemented in distributed-memory hardware (needs hardware support e.g. processor id in address).

Message Passing



Each instance of a program has its own address space. Message passing library calls allow data transfer.

Clearly a good match for distributed-memory hardware

Also very efficient on shared-memory hardware

Expressions of parallelism I

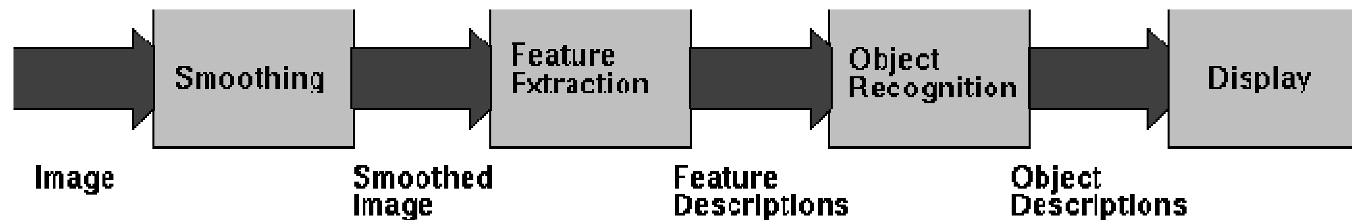
TRIVIAL PARALLELISM

- Run N copies of the serial code at the same time with different input parameters
  - e.g. ensemble forecasting in numerical weather prediction
- Minimal communications leads to high efficiency
- Sometimes called “embarrassingly parallel” but nothing wrong with this method
- If the scientific problem lends itself to this approach it is very efficient

Expressions of parallelism II

TASK PARALLELISM

- You could write 1024 different programs - NIGHTMARE!
- Maybe a small number of different programs which are assigned different numbers of processors
  - e.g. a pipeline in image processing



- Startup and shutdown costs associated with pipeline
- Applicable to limited class of problem
- Difficult to load balance

a project funded by EuropeAid

Expressions of parallelism III

DATA PARALLELISM

a.k.a. SPMD – Single-Program Multiple-Data

- Every processor runs the same executable but works with different data
- Message passing environment gives each process its rank
- Load balancing controlled by partitioning of data
  - e.g. in a grid-based problem give each processor the same number of points
- Most common approach to parallelism

a project funded by EuropeAid

Programming Environments I

- Serial language with message passing library
  - Fortran - traditional, still most common for scientific codes, new standards (F90, F2003, F2005) have greatly improved the robustness, modularity, even some object-oriented constructs
  - C - traditional, popular for systems programming, less common for scientific codes
  - C++ - extends C as a full object-oriented language, can lead to performance problems for numerical scientific codes
  - Java - robust, object-oriented, portable but inherent problems with optimisation for numerical codes
  - Message Passing Interface - de facto standard very commonly used with interfaces to Fortran and C (and Java?)

"The last decent thing written in C was Schubert's 9th Symphony" -- Anon

Programming Environments II

- Parallel languages which extend a serial language
  - Co-array Fortran, Unified Parallel C, Titanium (Java)
  - introduce an elegance that you can never achieve without changes to serial languages
  - not in common use so demands new code written from scratch (as opposed to extending applications which already exist)
  - difficult to find compilers, especially a good one, for a range of platforms
  - performance is not proven
- New parallel languages
  - Chapel (Cray), X10 (IBM), Fortress (Sun)
  - Funded through the DARPA HPCS program
  - Research area which may achieve good performance and become popular in 5-10 years time

Parallel Computing - Performance

Expectations

- It is important to understand the performance issues **before** we start designing and writing a program

- Faster results
  - We hope that with P CPUs we can solve problems (almost) P times quicker

- Larger Problems
  - Typically parallel machines have lots of memory so can do bigger problems
    - *e.g. even a 32 CPU system with 2 Gigabyte per processor has 64 GB: a lot more memory than your desktop*
    - *a BIG machine with over 1000 processors will have over a Terabyte of memory*

a project funded by EuropeAid

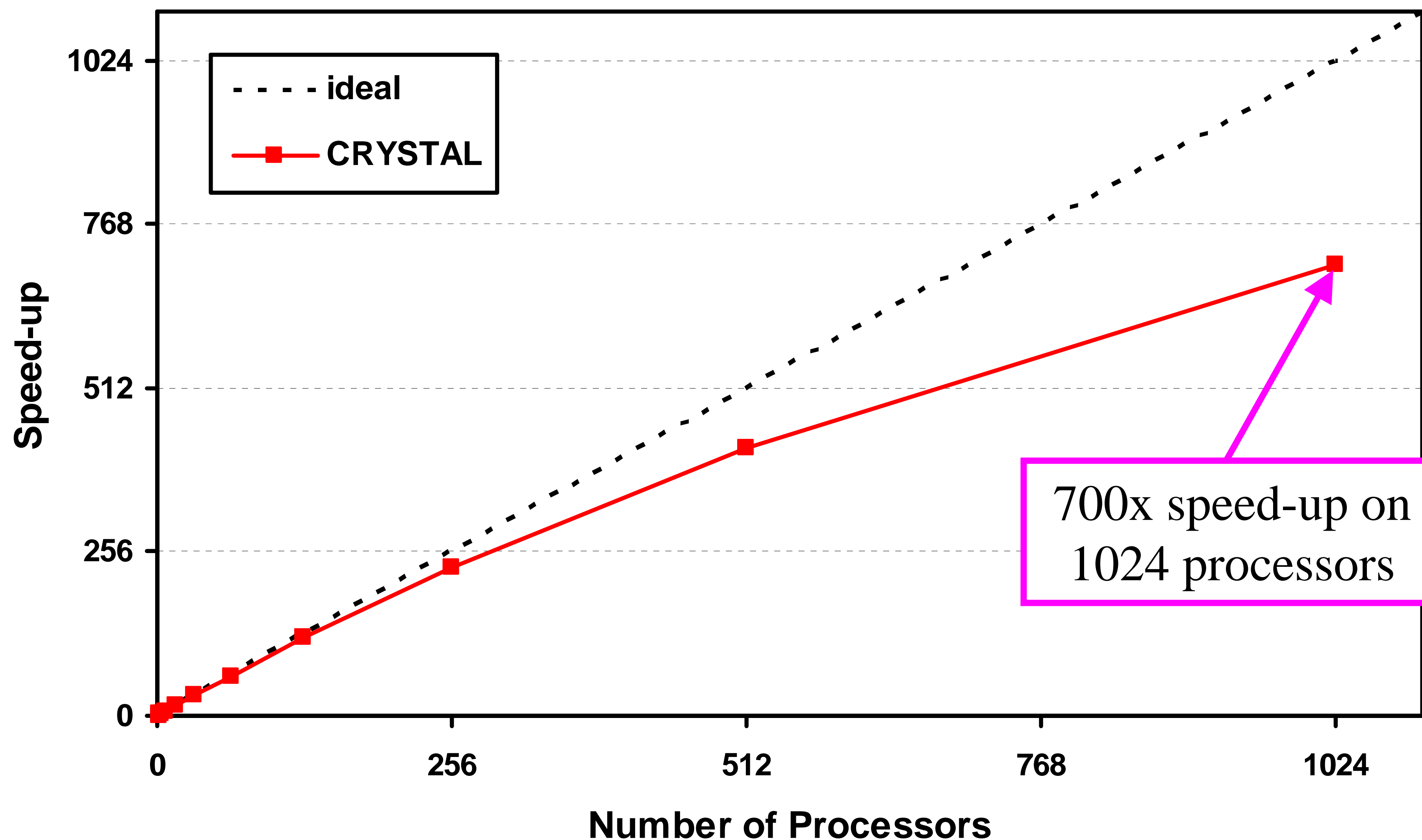
Speed-Up

- Speed-up is used to compare the performance of the same code on the same machine with different numbers of processors
- Relative speed-up is how much faster your program runs on P processors relative to 1 processor

$$S_p = t_1 / t_p$$

- Absolute speed-up is how much faster your program runs relative to THE BEST SERIAL IMPLEMENTATION
  - Sometimes these are the same. Often there is little difference. Sometimes the difference can be quite marked
    - Sometimes the best serial algorithm is not the best for parallel machines
    - Sometimes the best parallel algorithm is not the best for serial machines

Speed-Up of CRYSTAL on HPCx



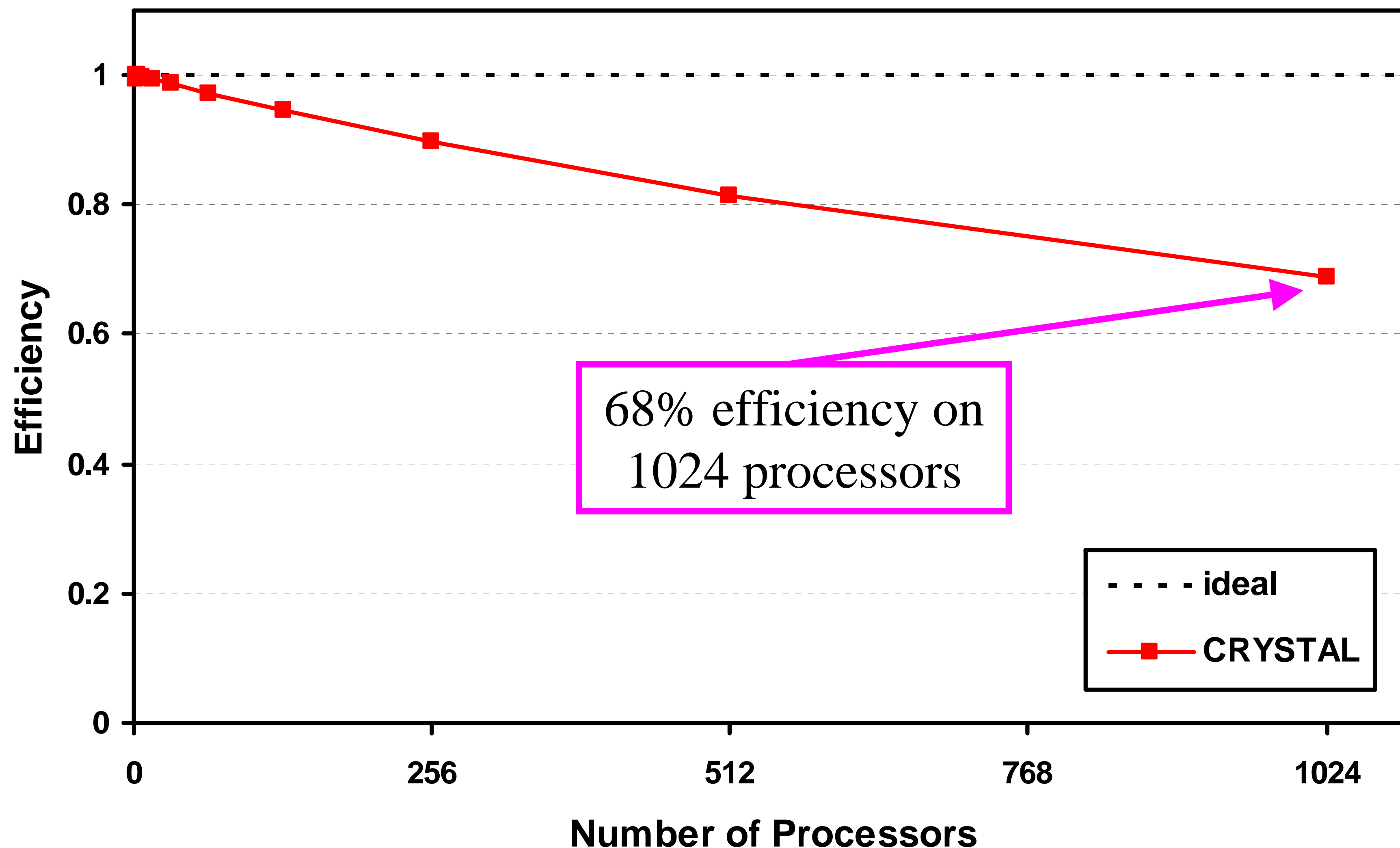
Efficiency

- The percentage of ideal speed-up obtained allows us define the Parallel Efficiency …

$$Eff_p = t_1 / P.t_p$$

- As with speed-up, with efficiency we have lost information about the actual run-time
- Never use to compare different machines
- Never use to compare different algorithms
- It is however a useful measure to see how well a code is scaling

Efficiency of CRYSTAL on HPCx



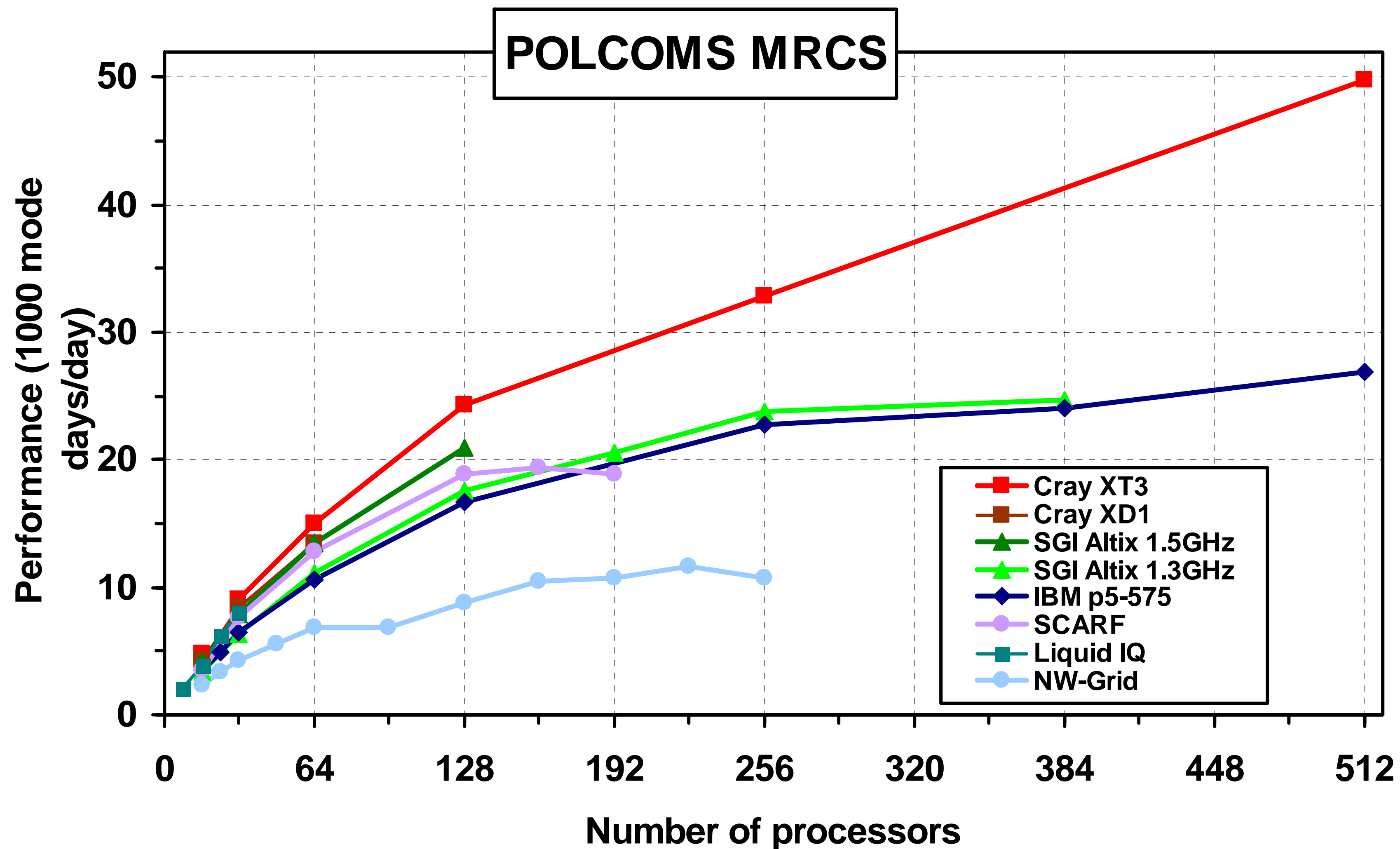
Demonstrating Performance

- Plot absolute time in seconds (wall-clock time)
  - Perfect behaviour is a reciprocal curve – difficult to see deviations from perfect behaviour at high P
- Plot absolute time as log-log
  - Everything looks like a straight line
- Best of all is to use performance …

Performance = constant / time

- Perfect behaviour is a straight line so easy to see deviations from perfect behaviour
- If the constant relates to the amount of work this can be related to something meaningful
  - e.g. model days per day in Numerical Weather Prediction
  - e.g. iterations per second in an iterative method

POLCOMS example of scaling



How To Measure Scaling

STRONG SCALING

- Keep the problem size the same as you increase the number of processors
- Problem size per processor decreases, possibly to the point where there is very little work per proc

WEAK SCALING

- Scale the problem size as you increase the number of processors
- Problem size per processor stays the same

Good WEAK SCALING

is easier to achieve than

good STRONG SCALING

Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

Actually the CRYSTAL example is very good

- Sometimes your program gets SLOWER with more processors

PART II

a project funded by EuropeAid

Parallel Computing - Overheads

Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

Limited Concurrency

- The number of tasks that can be executed in parallel is called the degree of concurrency
- The degree of concurrency is determined by the size of the dataset and the way it is partitioned
  - E.g. if a code only partitions data in one-dimension the maximum number of tasks will be the number of data points in that dimension. This limitation could be lifted by implementing a two-dimensional partitioning
- The degree of concurrency increases as the partitioning becomes finer in granularity and vice versa
- We will look at this further when we look at designing a parallel program

Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

a project funded by EuropeAid

Remaining Serial Code

If your program on one processor runs for f % of the time in perfectly parallel code, so it takes (1-f) % in serial.

So on P processors it runs for f/P % of the time doing work in parallel, but still (1-f) % in serial execution.

The speed up is therefore

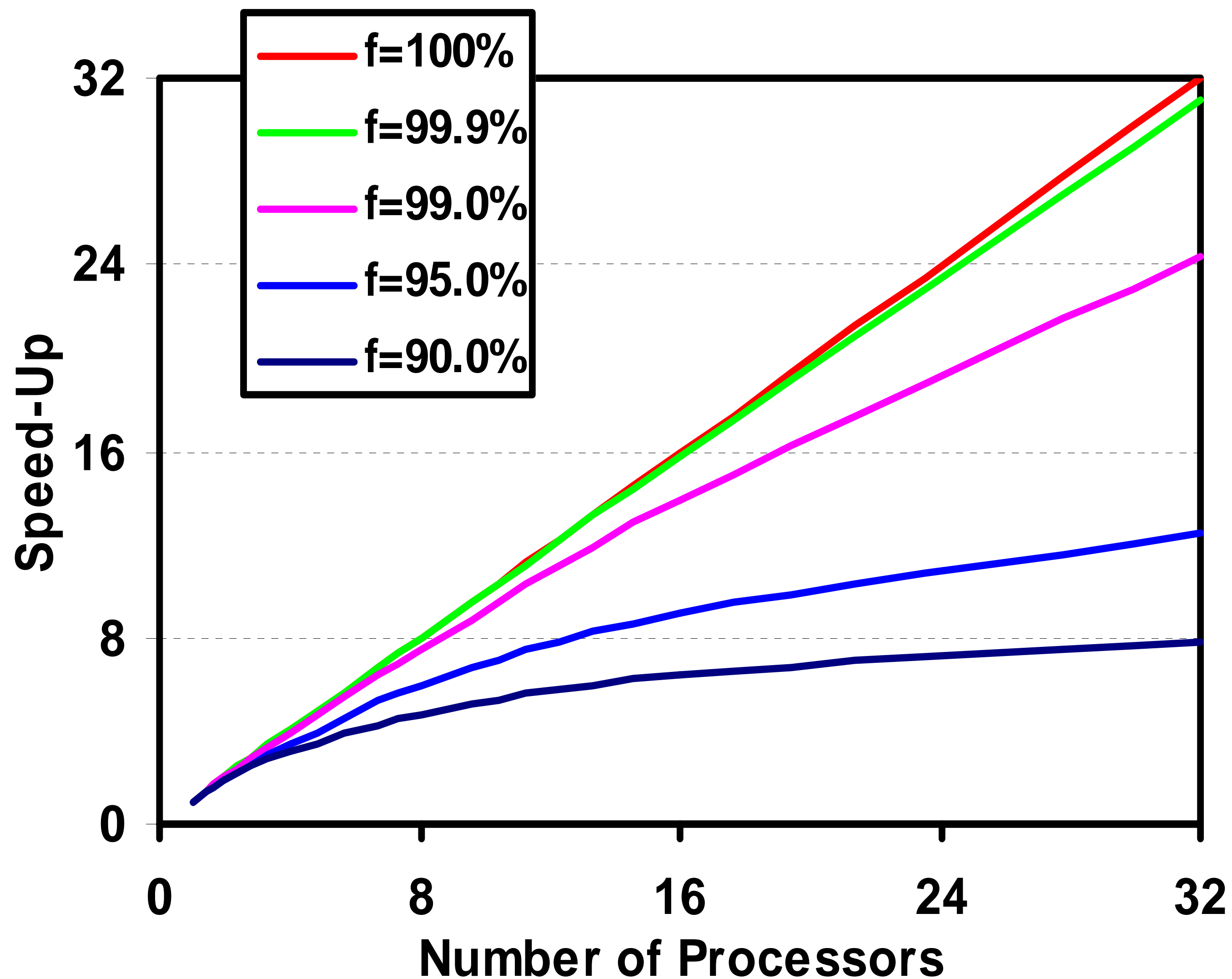
$$\text{Speed up} = 1 \;/\; ((1 - f) + f / P)$$

This is known as Amdahl's law - it is somewhat scary !

- On a infinite number of processors the speed-up is 1/(1-f), so even if your program is running 90% parallel the best speed up you can EVER get is 10 !
- The CRYSTAL results presented earlier when fitted to Amdahl's law give f=99.95%

a project funded by EuropeAid

Amdahl's Law



SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

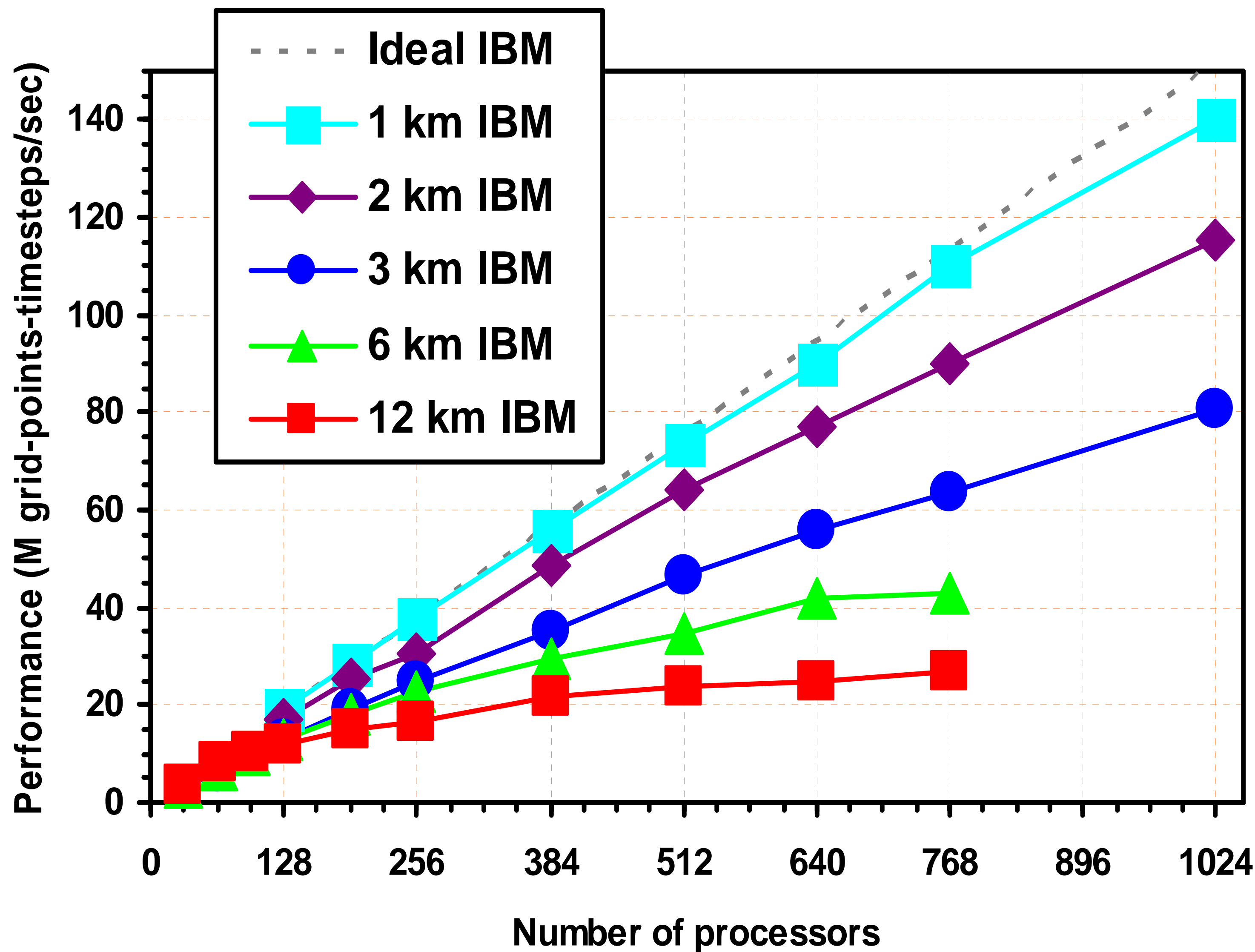
Is It Really That Bad ?

- Amdahl's Law assumes a fixed size problem
- f is usually a function of the problem being addressed
- In many problems the portion of the code that has been parallelised depends strongly on the problem size in some way (e.g. N^3) while the serial portion scales much less strongly
  - Gustavson's Law

Parallel Computing Is Best For Large Problems

- Both Amdahl and Gustafson only consider remaining serial code

Speed Up Curves for POLCOMS



Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

Load Imbalance I

- Speed of a parallel program depends on the speed of the slowest processor; the one with most work

$$\text{Load balance} = \frac{\text{sum}_{i=1,N} (\text{time}_i) / N}{\max_{i=1,N} (\text{time}_i)} < 1$$

- Simple multiplier of the time

  Actual time = Load balance * Ideal time
- Can affect the scaling as load balance generally worsens with larger numbers of processors

a project funded by EuropeAid

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Load Imbalance II

- For grid-based simulations it is usual to partition the grid into equal numbers of grid points
  - assumes work is proportional to number of grid points
- Reality is more complex:
  - variations across the grid e.g. land/sea, night/day
  - additional/reduced computation at the domain boundaries
  - communication imbalance
  - input/output e.g. gather results onto one processor for output

Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

a project funded by EuropeAid

Message passing I

Point-to-point communications

- Latency-Bandwidth model

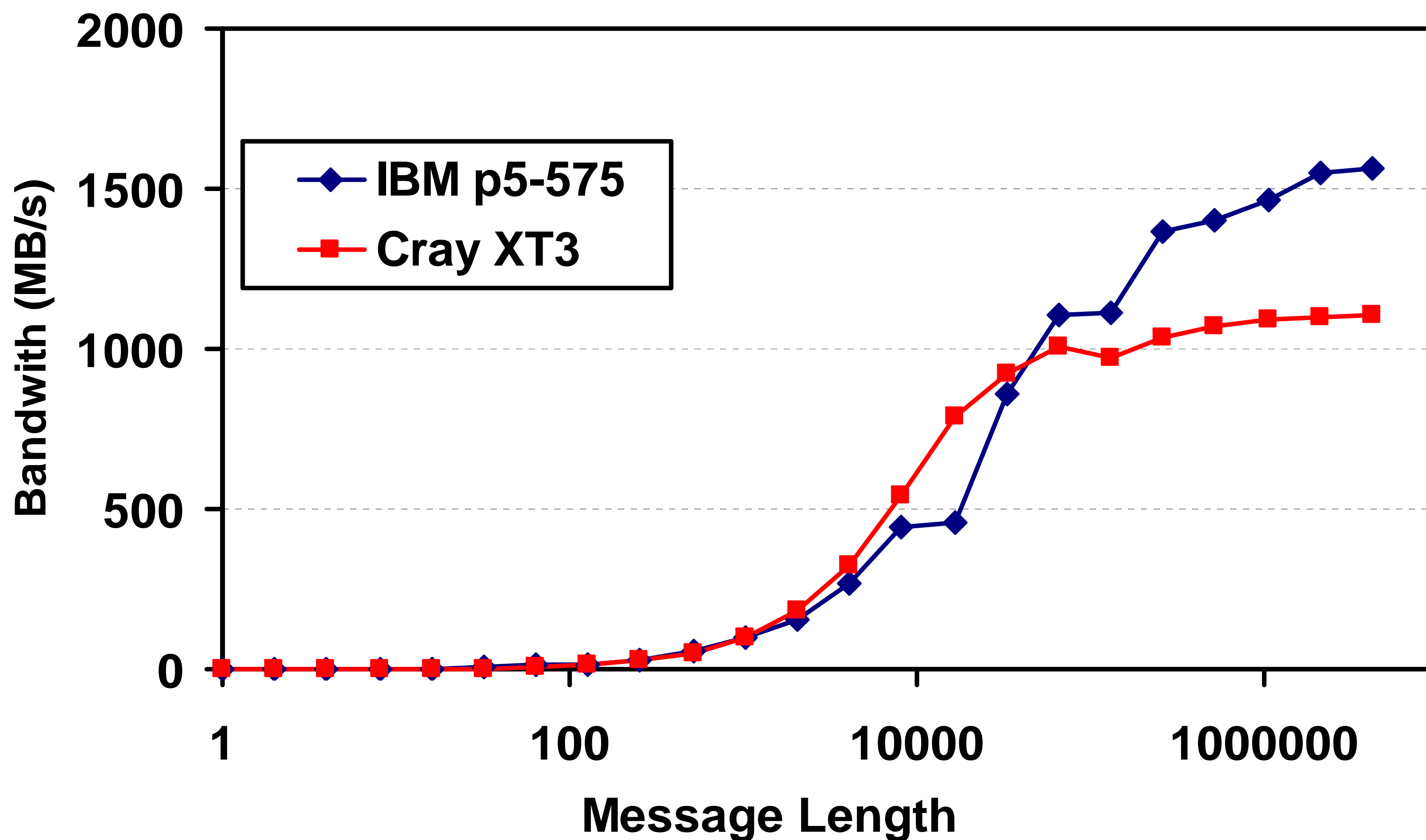
$$t = L + n/B$$

  - time, t, for message of length n bytes
  - depends on Latency, L, and Bandwidth, B
- Codes with many short messages are Latency dominated, those with long messages are Bandwidth limited
- Latency & Bandwidth are dependent on hardware
  - low latency is particularly hard (expensive) to achieve
  - latency also affected by software layers
  - Beware manufacturers hardware figures – you are unlikely to achieve this in practice!
  - Intel MPI Benchmarks

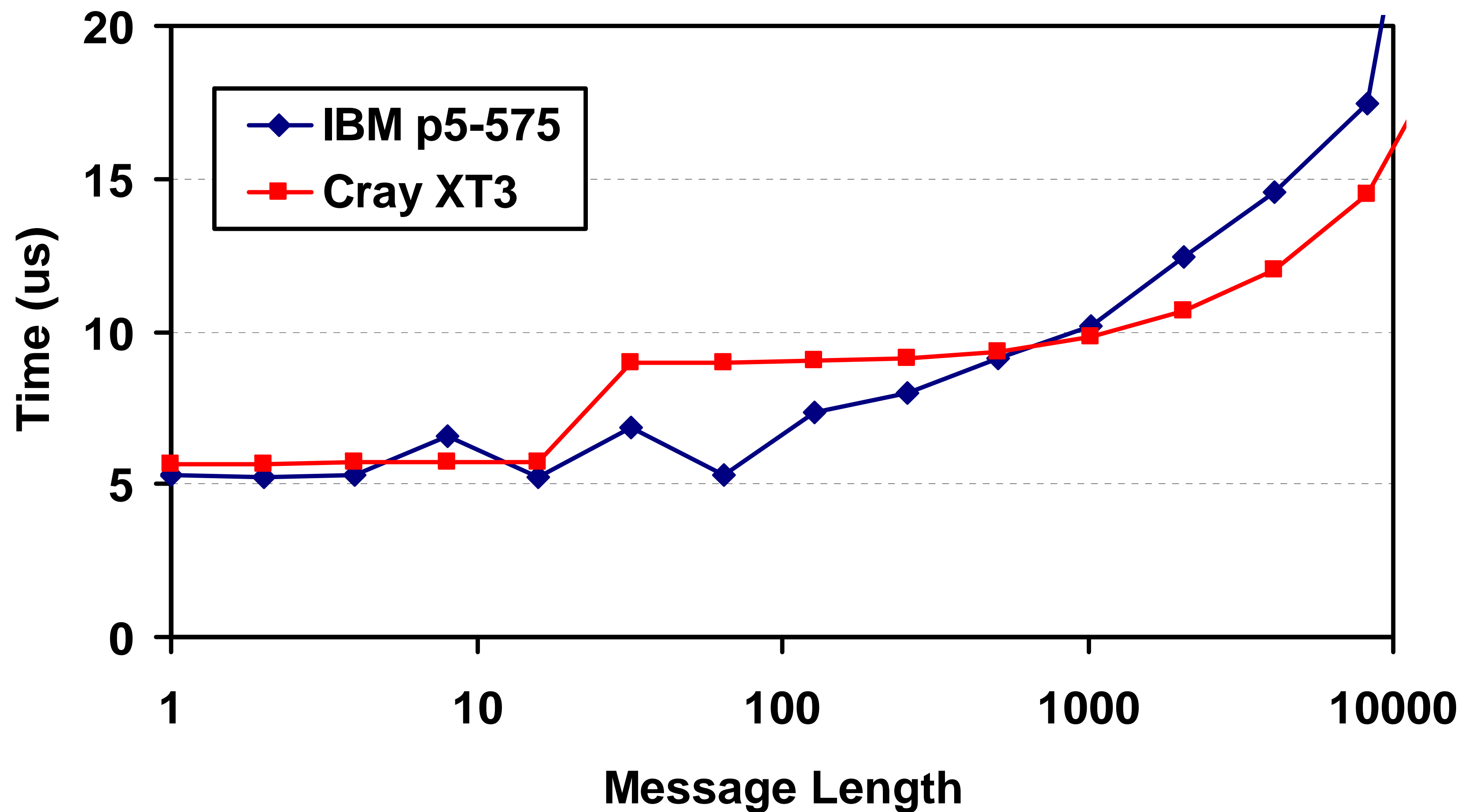
a project funded by EuropeAid

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Measurement of Bandwidth



Measurement of Latency



Message passing II

Global communications

- Operations like broadcast, global sum, global max, gather/scatter, etc.
- Often implemented as a tree algorithm
  - need log P stages to reach all P processors
- Also Latency dominated for small data sizes
- Can be made "SMP-aware" for clusters of SMPs
  - first do global op within the SMP node, making use of fast shared memory communications
  - then go across the network using only one link per SMP node

Effect of comms on scaling

Serial code limits speed-up as you add more processors

Comms can slow you down as you add more processors

Speed-up

60
40
20
0

0 256 512 768 1024

Number of Processors

- pure Amdahl 99% parallel
- Amdahl plus comms

a project funded by EuropeAid

Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

a project funded by EuropeAid

Memory and Cache Issues

- Access to caches and to main memory can change as we increase the number of processors and therefore affect the scaling
- One common effect is

SUPER-LINEAR SCALING

- On larger numbers of processors the problem size on EACH processor is smaller, fits better into cache and runs faster
- Result:

Speed-up > 1

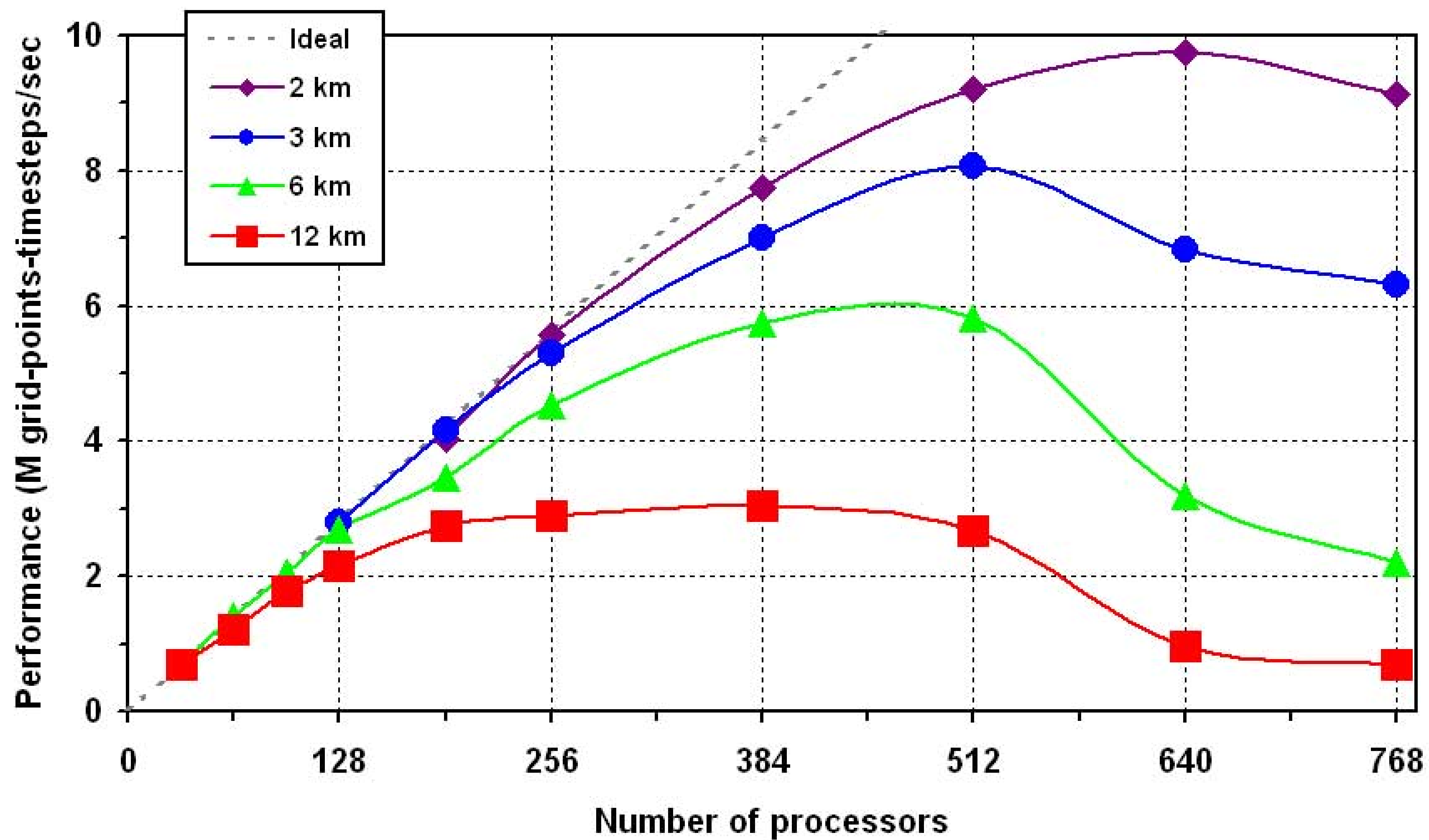
Hindrances to Perfect Scaling

Why Don't We Get Perfect Scaling?

- Limited Concurrency in the problem
- Remaining Serial code (other processors idle)
- Load Imbalance (those with less work have to wait)
- Message Passing (communications takes time)
- Memory and cache issues
- Other shared resources (e.g. input/output)

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

POLCOMS with full output



Parallel Program Design

a project funded by EuropeAid

Parallel program design

- We have seen a variety of programming models
- The most common – suitable for most applications and efficient on most hardware – is …

Single Program Multiple Data

Data parallelism

Serial language (Fortran or C) + MPI

- Within this model the crucial design decision is

how to partition the data

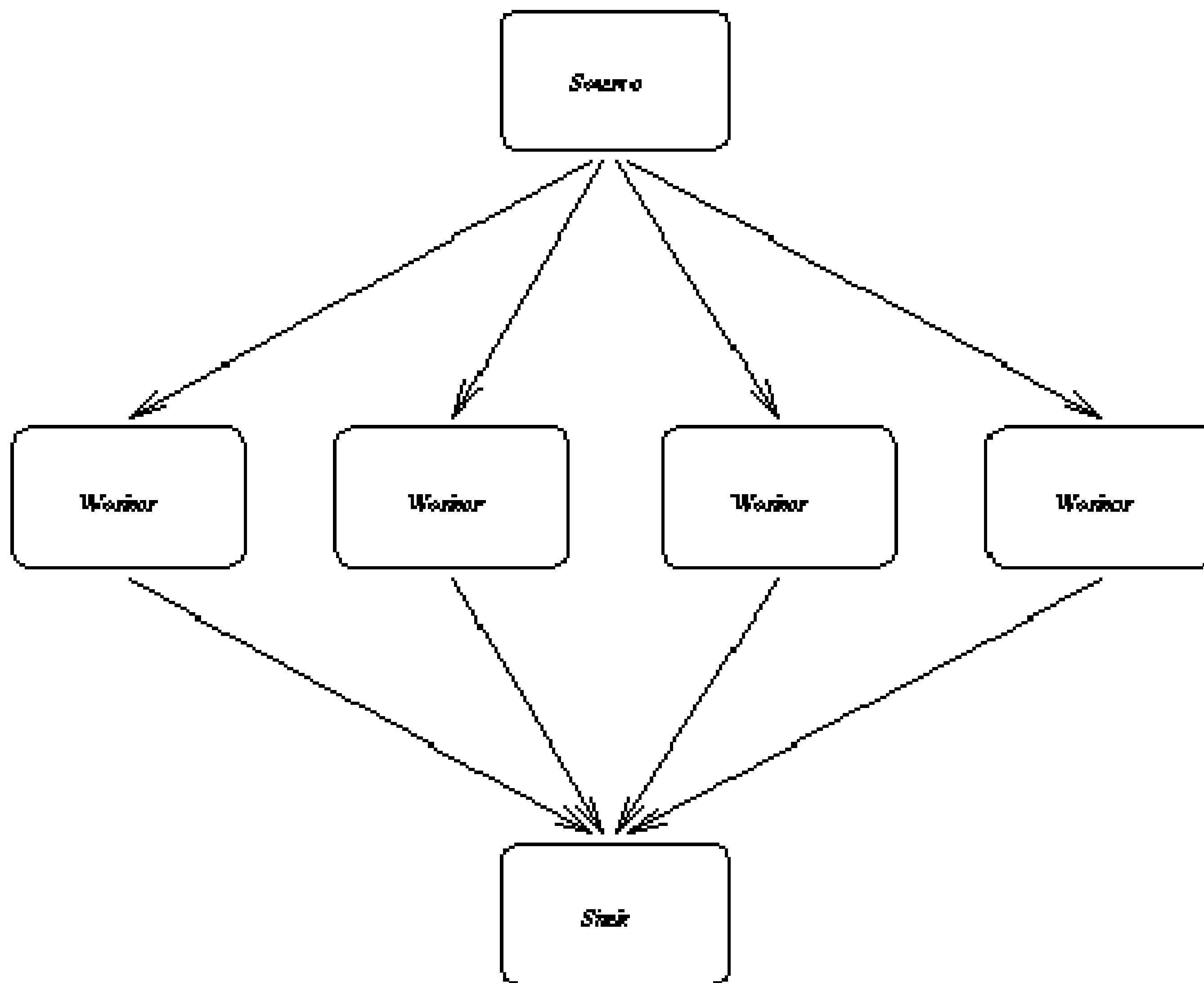
it affects load balance, communications, serial parts etc.

Task farm I

- Data is divided into small sections, tasks, which can be worked on independently
- More tasks than processors
- Each processor is given a task to work on, and sends back results when finished, then given another task
- Usually a master processor which sends and receives tasks, plus N-1 worker processors
- No communication between workers – tasks are independent – major restriction
- Self load balancing – as soon as a worker runs out of work it is given more – only imbalance is when they finish
- Master can become a communications bottleneck

a project funded by EuropeAid

Task farm II



Trivial task farm

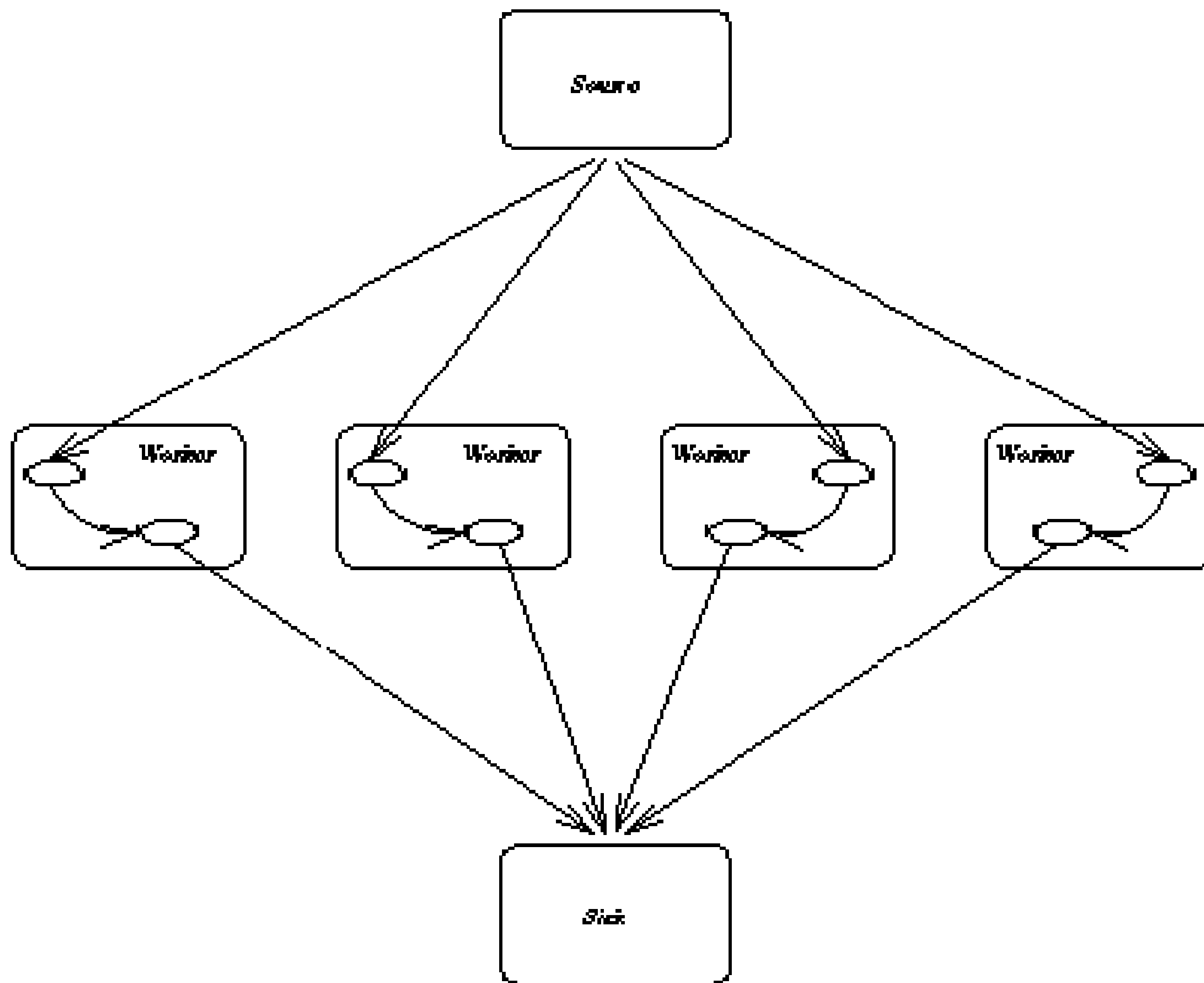
- Task farm approach often used to implement trivial parallelism
- Start up one task farm rather than a large number of processors
- Same number of tasks as processors
- Master passes out initial conditions to workers and then starts work on a task as well

a project funded by EuropeAid

Buffered task farm I

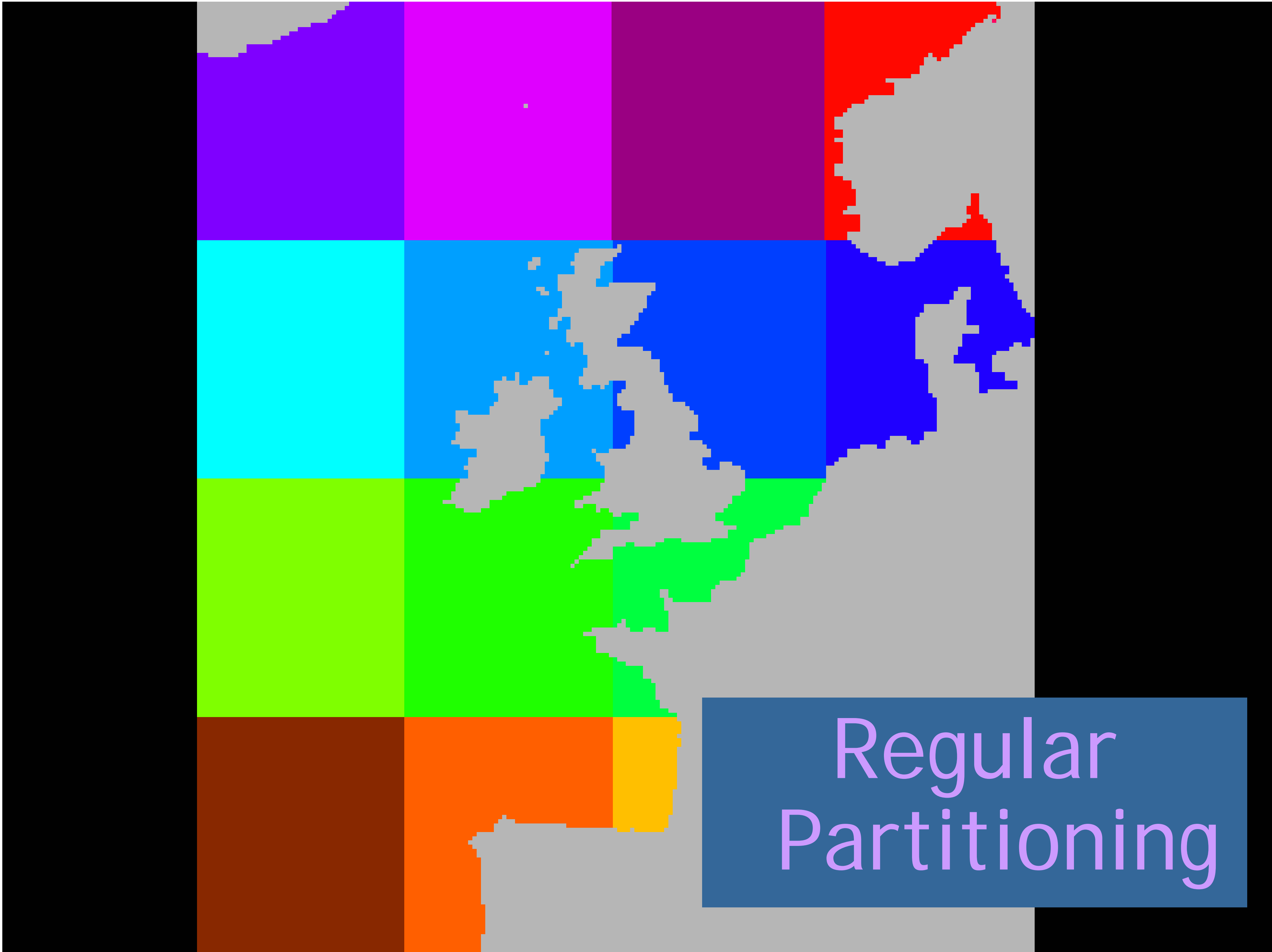
- We can help to reduce the communications overhead by ensuring that workers have more work to do while they are waiting for a response from the master
- Initially send them two pieces of work
- The second is queued in a buffer
- Start with the buffered task as soon as the first is finished
- New task from master replaces the buffered task

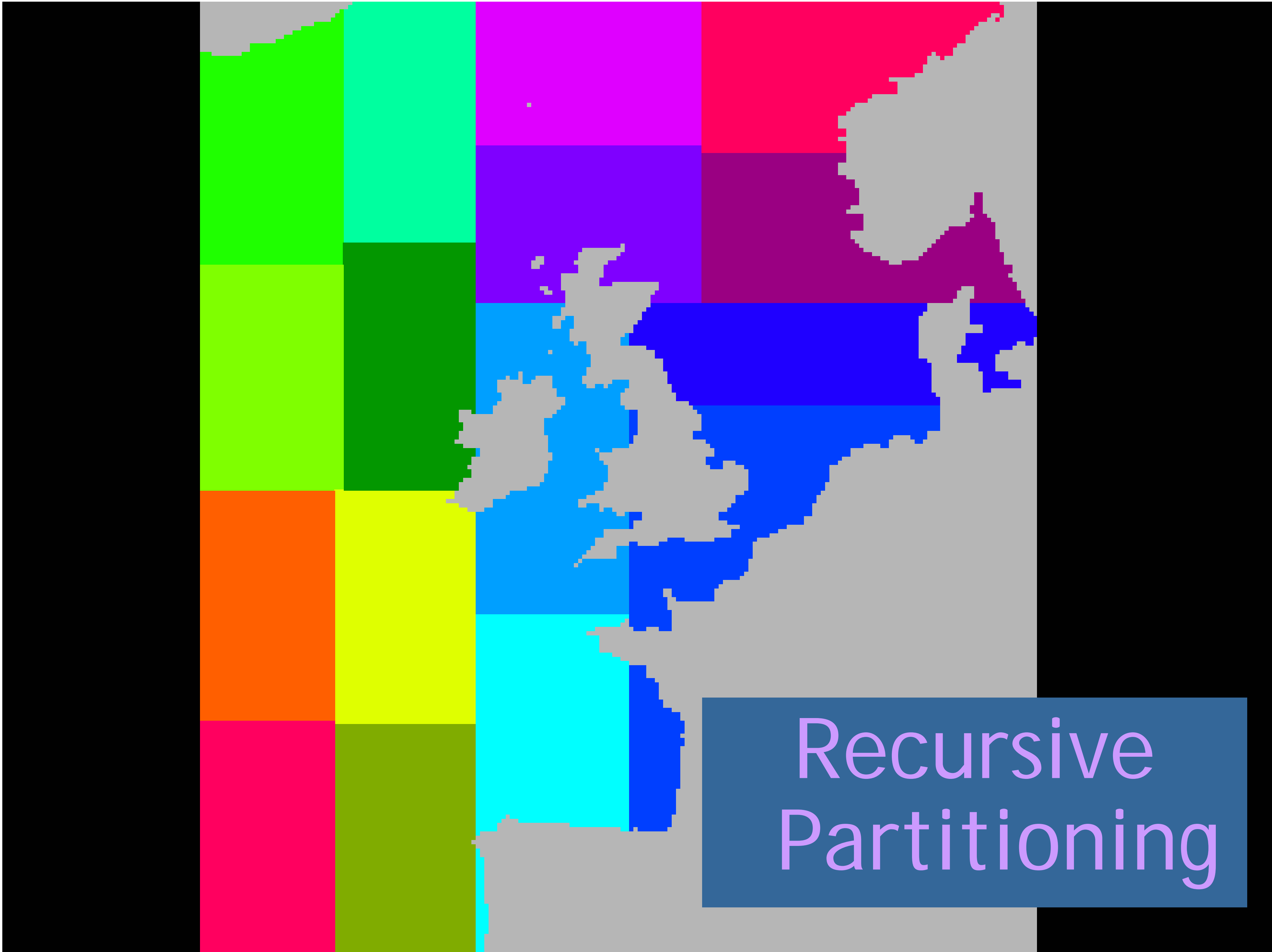
Buffered task farm II



Data partitioning I

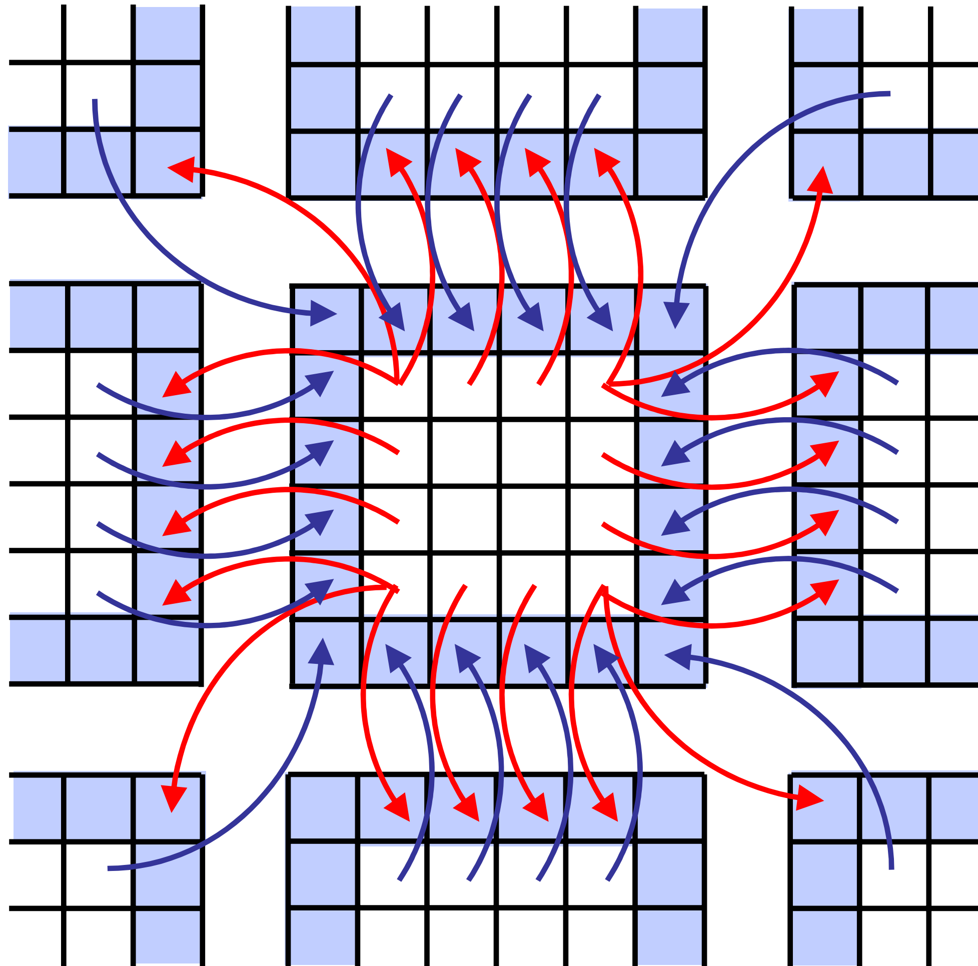
- Partition the problem domain into sub-domains
- Distribute sub-domains among processors
  - Usually one per processor
  - Sometimes more than one per processor but usually an unnecessary complication
- Aim to ensure that data is distributed as evenly as possible between the processors
  - If work is proportional to the number of grid points, distributing the grid points evenly gives a good load balance
  - However, may need to look also at the communications load balance
- Also aim to minimise the communications
  - E.g. minimising the number of cut edges in a FE mesh
- Owner computes new values, boundary exchange to update halo data …


Regular Partitioning


Recursive
Partitioning

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Boundary exchange I



- White – sub-domain
- Blue - halo
- Halo width depends on accuracy of the scheme
- Shown here in 2D - extends to 3D
- Compute ~ N^3/P - volume
- Comms ~ $N/P^{1/3}$ - surface
- Comms becomes more important as P increases

- 8 directions with inefficient single-point *corner* messages – but the recursive partitioning shown before improves this

Owner computes

- The Owner Computes Rule generally states that the process assigned a particular data item is responsible for all computation which changes its value

e.g.

a(i,j) = a(i,j) + 0.5*a(i+1,j) + 0.5*a(i-1,j) + ...

- The ranges of i and j cover the points we own
- Assignments only to points we own
- Never assign e.g. to a(i-1,j)
- Can reference points we do not own on RHS

Boundary exchange II

- We don't want processes communicating every time they need a single element from a neighbour
- Send boundary data after it has been updated
- Receive boundary data before it is used

```
a(i,j) = … … …
send boundary values of a to neighbours
…
…
receive boundary values of a from neighbours
… = … a(i+1,j) …
```

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Boundary exchange III

- We know that the LHS elements are local
- Look at the RHS terms – they determine the communications dependencies
- E.g. where the i-j grid corresponds to a longitude-latitude grid we have the following …

… = … a(i+1,j) …	Send data W	(-i direction)
… = … a(i-1,j) …	Send data E	(+i direction)
… = … a(i,j+1) …	Send data S	(-j direction)
… = … a(i,j-1) …	Send data N	(+j direction)
… = … a(i+1,j+1) …	Send data SW	
… = … a(i+1,j-1) …	Send data NW	
… = … a(i-1,j+1) …	Send data SE	
… = … a(i-1,j-1) …	Send data NE	

Partial exchange

- Note that we may not have to send in all directions - look at the RHS code

e.g.

```
a(i,j) = 0.25*(a(i,j) + a(i+1,j) + a(i+1,j+1)+a(i,j+1))
```

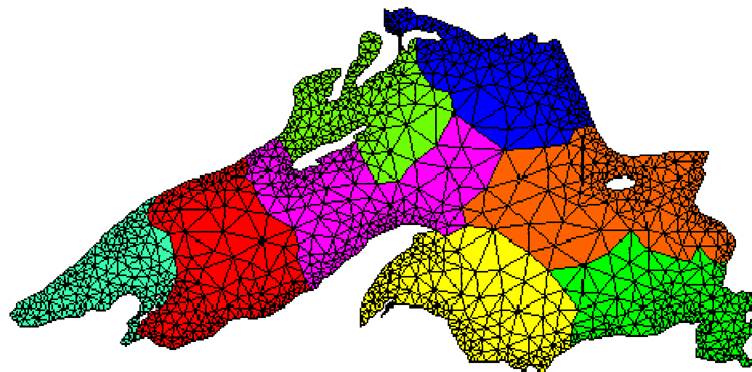
- Only need to send in three directions – S, W and SW – not all eight
- The communications can therefore be made much more efficient by paying attention to the requirements of the algorithm

Unstructured grids

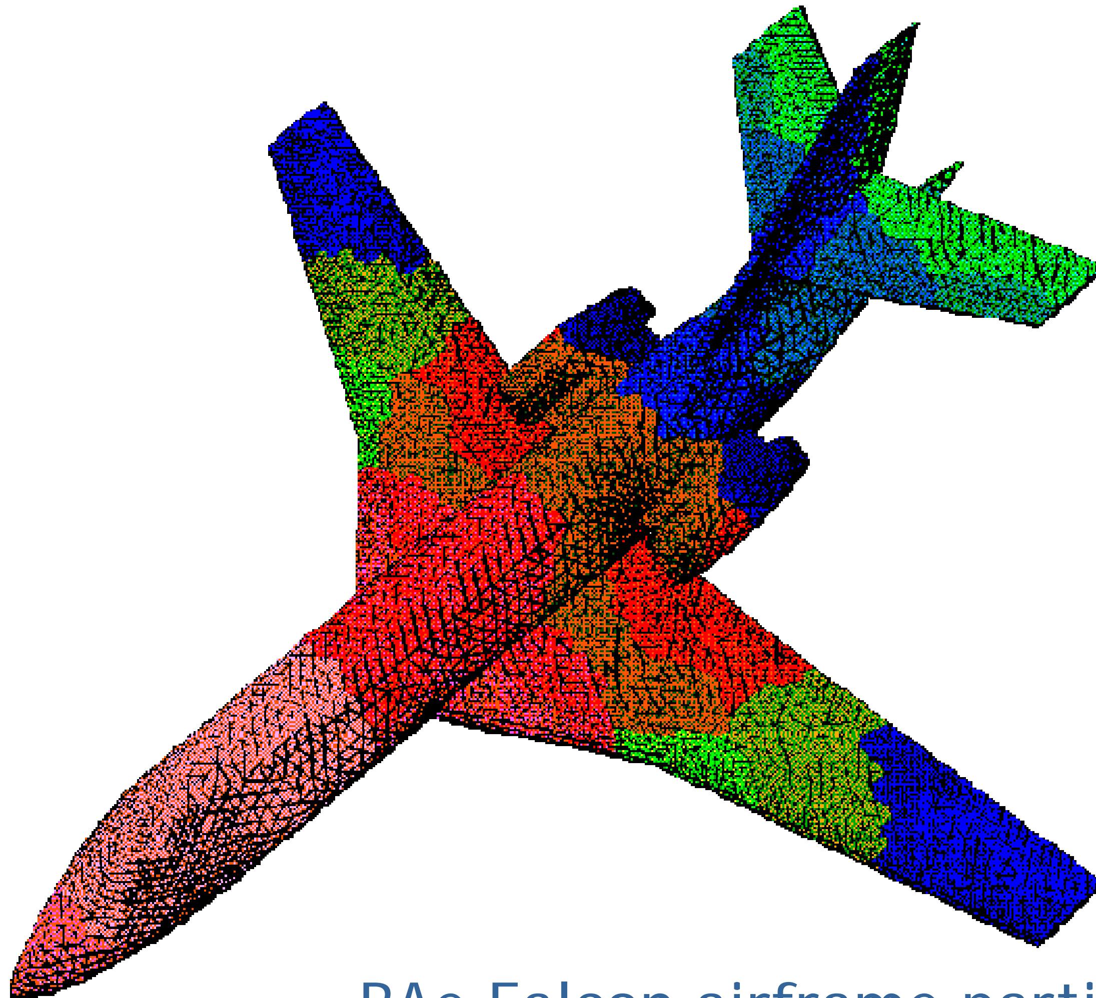
- Same principles apply as for structured grids
- Boundaries are lists of points rather than rows & columns
- Use a graph partitioning program to partition the grid
  - most popular is Metis

    http://www.cs.umn.edu/~metis
  - provides optimal load balance
  - minimizes cut edges for minimal communications

E.g. surface grid for Lake Superior



Partitioning example



BAe Falcon airframe partitioned for 16 processors

Checklist for parallelisation

- Partition the problem domain
  - set up new indexes for the bounds of the sub-domain
- Assign sub-domains to processors
- Change loop bounds to run over the local sub-domain
  - ensure the *owner computes* rule is obeyed
- Determine the communications dependencies
  - examine the relationship between LHS assigns and RHS references to non-local data
- Insert communications calls
- Test correctness
  - Results should be bit-wise identical with number of processors
- Test performance
  - Scaling of performance with number of processors

Comment on correctness

I said "Results should be bit-wise identical with number of processors" … is this really true?

- For halo exchange … YES
- For some global operations … NO
- Beware global sum
  - on a digital computer (a+b)+c .ne. a+(b+c)
  - global sum typically does a local sum on the processor then goes out over the network
  - include some test code to do the sum in the serial way for when you are testing for correctness

```
#ifdef EXACT
    gather data onto master, sum, broadcast result
#else
    do a parallel global sum
#endif
```

Summary

We have looked at …

- Serial Computing
  - we need parallel computers to solve large problems
- Parallel Computing – The Hardware View
  - there is a range of different architectures
- Parallel Computing – The Logical View
  - there is a range of architecture independent views
  - a range of programming environments
- Parallel Computing – Performance
  - we know how to measure performance
- Parallel Computing – Overheads
  - why know what causes poor parallel scaling
- Parallel Computing – Design
  - knowing all the above helps us write a parallel code

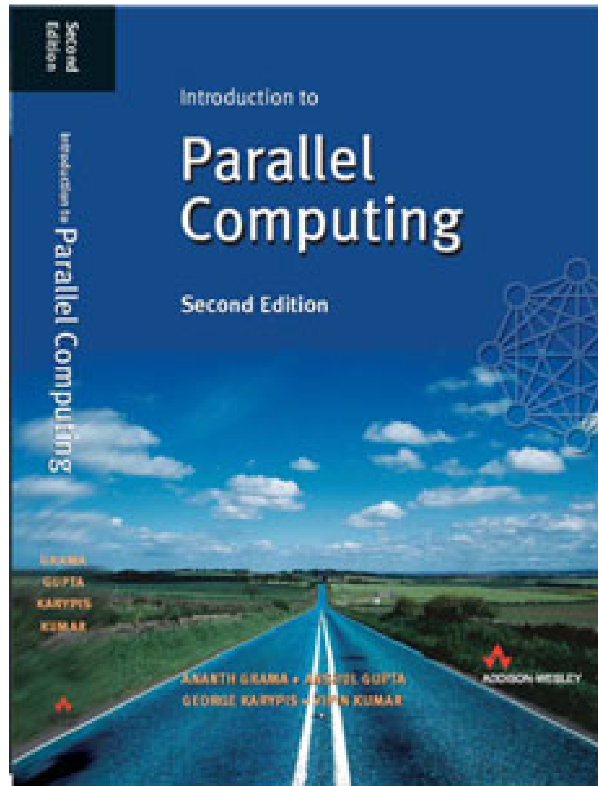
Checklist

What should you remember from this course?

1. Most common parallel programming technique
   - Single-Program Multiple-Data (SPMD)
   - Data parallelism
   - (Fortran or C) + MPI
2. Data partitioning is key to program design and scalability
3. Scalability depends on a range of factors
   - load imbalance, communications etc.
4. Investigate scalability of your code by plotting performance vs. number of processors

Resources

Introduction to Parallel Computing



Slides and exercises are available on the website

Grama, Gupta, Karypis & Kumar

http://www-users.cs.umn.edu/~karypis/parbook/

a project funded by EuropeAid

MPI Resources

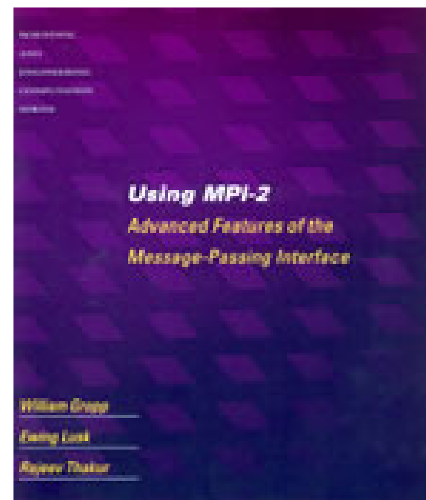
- MPI Home

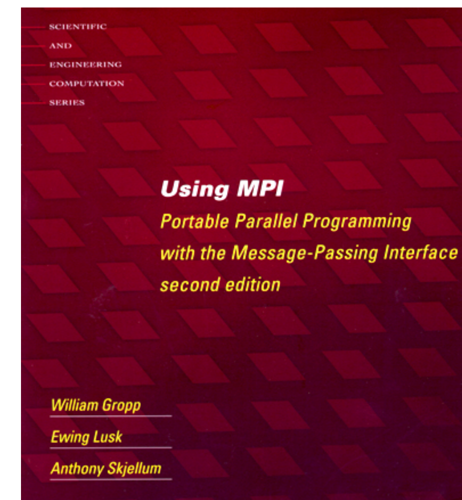
  http://www.mpi-forum.org/

  http://www-unix.mcs.anl.gov/mpi/

  - Contains MPI documentation (english)
- Google “MPI exercises” leads to several sites
- “The Book”

Using MPI-2 - Advanced Features of the Message Passing Interface, William Gropp, Ewing Lusk and Rajeev Thakur





Example programs available on the Argonne website

Tomorrow Thursday 11th January there will be two sessions on MPI including practical exercises on the UTFSM cluster

Starting 12:10

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

The End

If you have been … thank you for listening