SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

ALFA
EUROPEAID
CO-OPERATION OFFICE

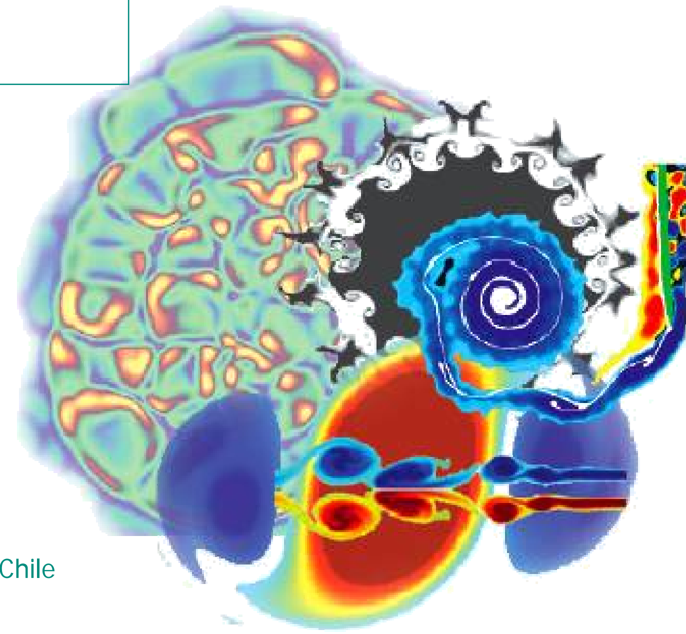
First Latin American SCAT Workshop:
Advanced Scientific Computing and Applications

Introduction to Grid Partitioning

Prof. David Emerson

CCLRC Daresbury Laboratory

University of Strathclyde



CCLRC

University of Strathclyde Glasgow

Overview of presentation

- Basic principles of parallel computing
- Basic principles of grid partitioning
    - A simple problem – calculating Pi in parallel
- Basic principles of explicit and implicit schemes
    - A comparison of typical CFD methods
- Practical example 1 – heat conduction in 1D
- Practical example 2 – the Poisson equation in 2D
- Multiblock, multigrid, and other topics

Brief synopsis

The vast majority of CFD problems are grid-based. To solve these types of problems on a machine with more than one processor it is necessary to split the problem up into smaller sized problems that will fit within the memory limits of the target machine. This is known as "grid partitioning" but is more often referred to as domain decomposition.

In principle, the programmer has complete control over the grid partitioning stage and therefore how the program is distributed to the processors. However, the programmer has no control over the communication network i.e. the latency and bandwidth of the machine. But, the programmer can be aware of any hardware limitations and use this information at the partitioning stage.

This course will explain the principles of how to partition a grid and how to understand what to look for.

Basic reasons for parallel computing

There are two main reasons for using parallelism

1) to reduce the computational time
   - this is usually for fixed-size problems
     e.g. you already have grid independence but need answer faster
2) to increase the accuracy
   - this is usually for a fixed-time problem
     e.g. you need to achieve grid independence but have time constraints
     i.e. industry often driven by *overnight* solutions.

Basic principles of parallel computing – programming model

The basic approach to parallelism is the **S**ingle **P**rogram **M**ultiple **D**ata (SPMD) model.

In this model, every processor sees the same program but operates on different data.

This is probably the most important model in scientific parallel computing.

Basic principles of grid partitioning

Computational Fluid Dynamics (CFD) is all about solving the *Navier-Stokes* equations.

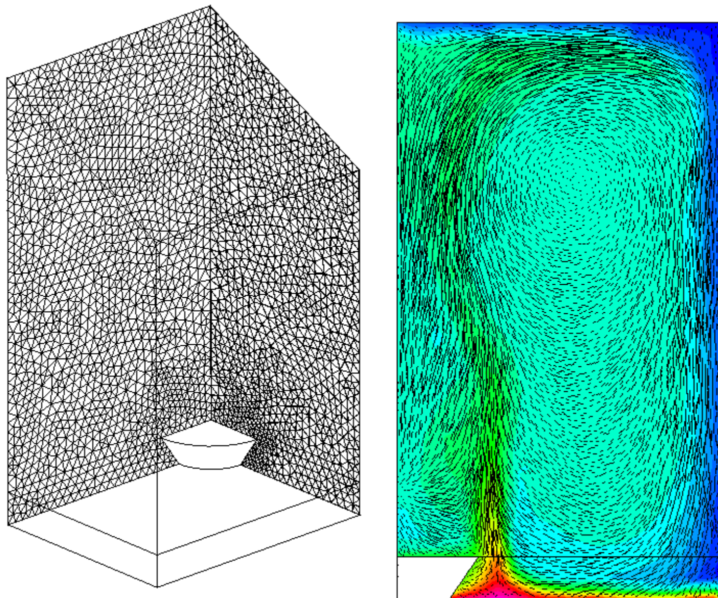
The Navier-Stokes equations are a coupled set on nonlinear partial differential equations (PDEs).

To solve this system of equations, we generally need to discretize the governing PDEs onto a computational grid.

To solve this system in parallel, we need to *partition* the computational grid (note - this is often referred to as *domain decomposition*).

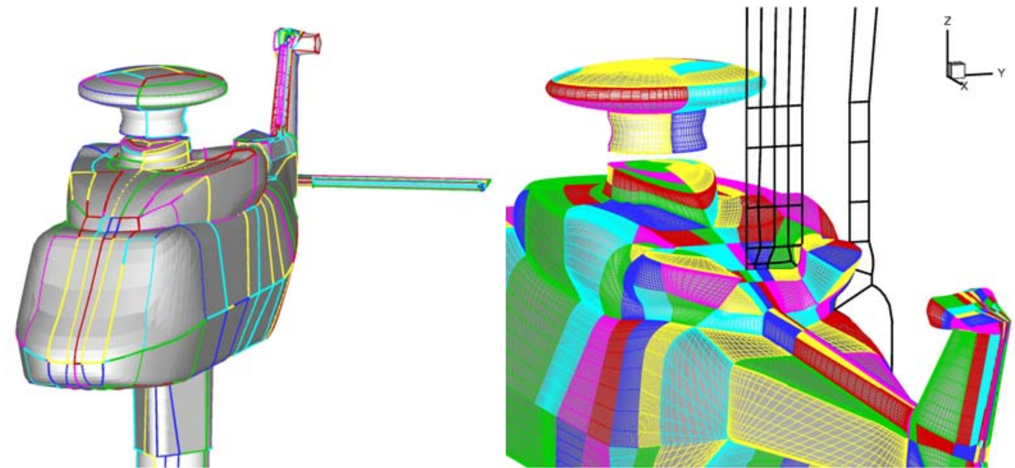
Examples of typical grids used in CFD

unstructured



Thermal flow around the Beagle 2 lander module

structured multiblock



NH19 fuselage
figure courtesy of
Barakos and Badcock (Liverpool)
2,226 blocks, over 12M cells

Basic principles of grid partitioning

A simple model problem – calculating Pi

The starting point is $$\pi = 4\int_0^1 \tan^{-1}(x)\,dx = \int_0^1 \frac{4}{1+x^2}\,dx = \int_0^1 f(x)\,dx$$

The approach to solving this type of problem is known as *functional* decomposition [1].

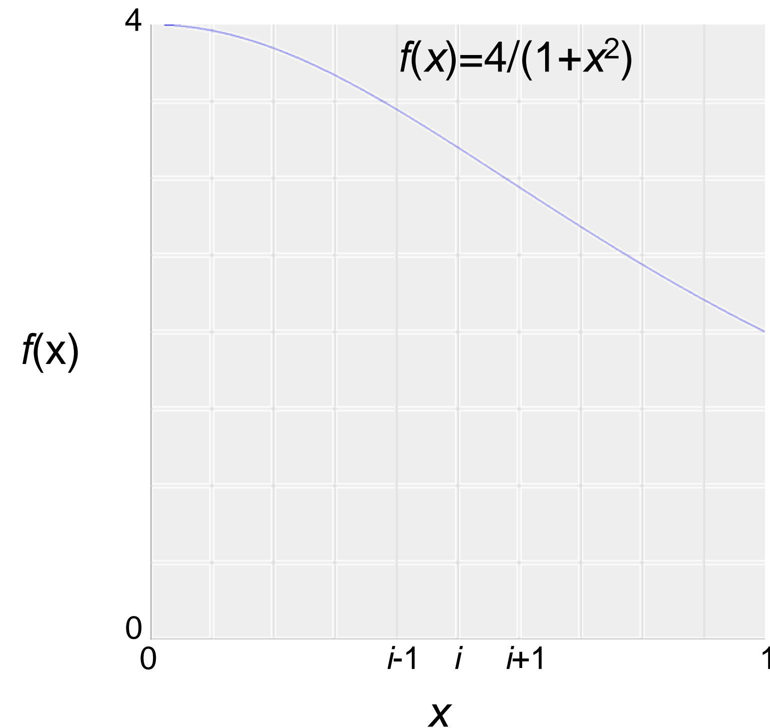
1) Chandy and Taylor, 1992, *An introduction to parallel programming*

To solve this problem, we need to integrate (numerically) the function. We will use the Trapezoidal Rule.

Note: area of trapezoid is $$A = W \times \tfrac{1}{2}\left[f(x_i) + f(x_{i+1}) \right] \approx W \times f(x_{i+1/2})$$

where W is the width of the interval and f is evaluated at the midpoint.

Graphical illustration of the function



Divide the function into *N* equal strips. The width, *W*, is simply 1/*N* and the area under the integral is:

$$A = W \times \left[f\left(x_{1/2}\right) + f\left(x_{3/2}\right) + \ldots. + f\left(x_{N-1/2}\right) \right] = W \times \sum_{i=1}^{N} f\left(x_{i-1/2}\right)$$

Calculating Pi in parallel – some points to note

1) The calculation is like a 1D grid problem (without boundary conditions)
2) The summation does not depend on the order

$$A = W \times \sum_{i=1}^{N} f\left(x_{i-1/2}\right) = W \times \left[\sum_{i=1,3,5}^{N-1} f\left(x_{i-1/2}\right) + \sum_{i=2,4,6}^{N} f\left(x_{i-1/2}\right) \right]$$

How do we solve this in parallel?

To start, let's assume we have *p* processors, where $p = N$.

Calculating Pi in parallel

Assign strip 1 to processor 1,

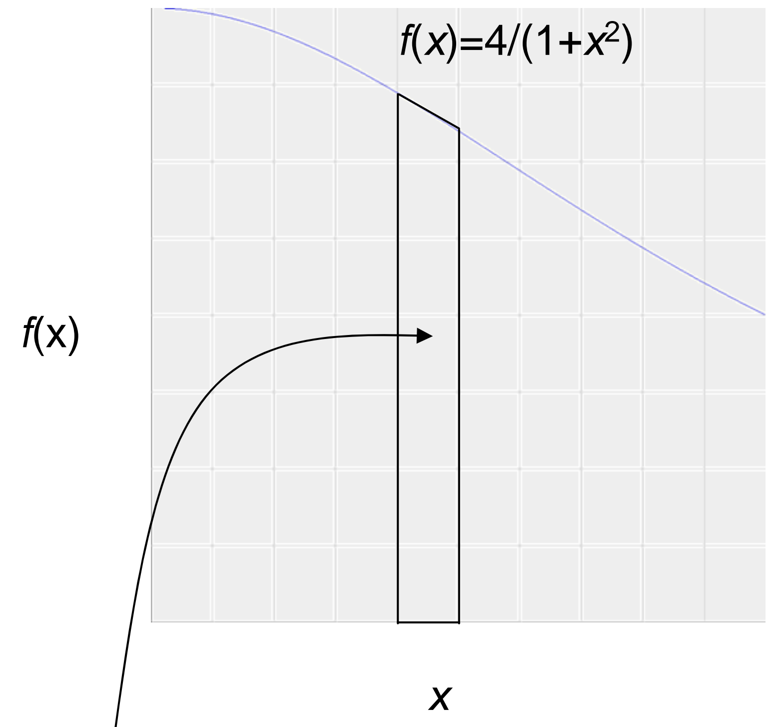
strip2 to processor 2, etc.

Assume each processor knows its value for *x*

Processor 1 knows $A_1 = W \times f\left(x_{1/2}\right)$

Processor 2 knows $A_2 = W \times f\left(x_{3/2}\right)$

Processor *i* knows $A_i = W \times f\left(x_{i-1/2}\right)$



strip *i* to the *i*'th processor

Each processor now "owns" a part of the solution.

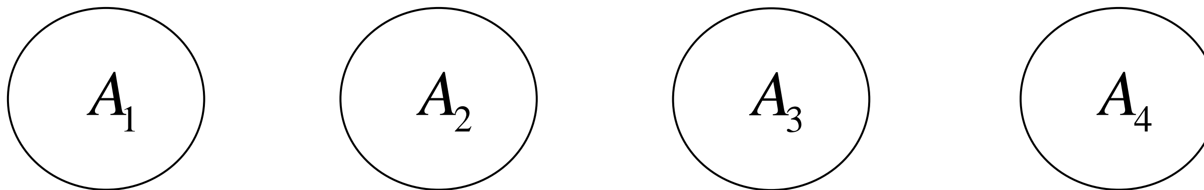
Calculating Pi in parallel

Each processor must now communicate to determine the value of Pi.

Using MPI, this can be done quite efficiently.

Before doing that, let's examine some possible ways of calculating the value of Pi by communicating each value.

Assume we have 4 processors, each holding a value as in figure below.



Calculating Pi in parallel – a linear chain

Denote processor 1 as P_1 etc. and start by doing the following

$$P_1(A_1) \rightarrow P_2(A_2^* = A_1 + A_2) \rightarrow P_3(A_3^* = A_2^* + A_3) \rightarrow P_4(A_3^* + A_4)$$

The notation should be read as processor 1 sends to processor 2, the two values are added, processor 2 sends the result to processor 3, the two values are added etc.

Some important points to note:

1) the communication time scales with the number of processors

2) only the final processor (in this case P_4) knows the value of Pi.

3) for all processors to know Pi, communication time scales as $2P$

Clearly there are better ways to perform the communication step.

Calculating Pi in parallel – a reduction operation

What if all processors send their data to P_1 and the data is added?

$$\begin{array}{rl} & \swarrow P_2(A_2) \\ P_1(A_1 + A_2 + A_3 + A_4) & \leftarrow P_3(A_3) \\ & \nwarrow P_4(A_4) \end{array}$$

Some important points to note:

1) only processor 1 knows the value of Pi

2) communication time considerably reduced

3) the bad news is we have created a major bottleneck

A simple model problem – calculating Pi in parallel

The best approach is based on the following

$$P_1\left(A_{12}^* = A_1 + A_2\right) \rightleftarrows P_2\left(A_{12}^* = A_1 + A_2\right)$$

$$P_3\left(A_{34}^* = A_3 + A_4\right) \rightleftarrows P_4\left(A_{34}^* = A_3 + A_4\right)$$

Here, all processors exchange data concurrently.

The next step is a further exchange

$$P_1\left(A_{12}^* + A_{34}^*\right) \rightleftarrows P_3\left(A_{12}^* + A_{34}^*\right)$$

$$P_2\left(A_{12}^* + A_{34}^*\right) \rightleftarrows P_4\left(A_{12}^* + A_{34}^*\right)$$

This is a *hypercube* summation and is highly efficient – the summation is completed after $\log_2 p$ operations. All processors know the answer.

A simple model problem – calculating Pi in parallel

- In practice, the safest strategy for communicating is as follows:
  - If processor number is odd, send data to even processor number
    If processor number is even, receive data from odd processor number
- After step is complete, you need to “reverse” the procedure i.e.
  - If processor number is even, send data to odd processor number
    If processor number is odd, receive data from even processor number
- The commands to do this operation will be covered in detail in the MPI course by Dr. Mike Ashworth

A simple model problem – calculating Pi in parallel

What if $p < N$?

This is always the case for practical problems.

We need to *distribute* the data to get a good *load balance.*

Let $p = 4$ and $N = 40$. Clearly work is balanced if each processor has 10 strips i.e.

$$P_1\left(\sum_{i=1}^{10} A_i\right) \quad P_2\left(\sum_{i=11}^{20} A_i\right) \quad P_3\left(\sum_{i=21}^{30} A_i\right) \quad P_4\left(\sum_{i=31}^{40} A_i\right)$$

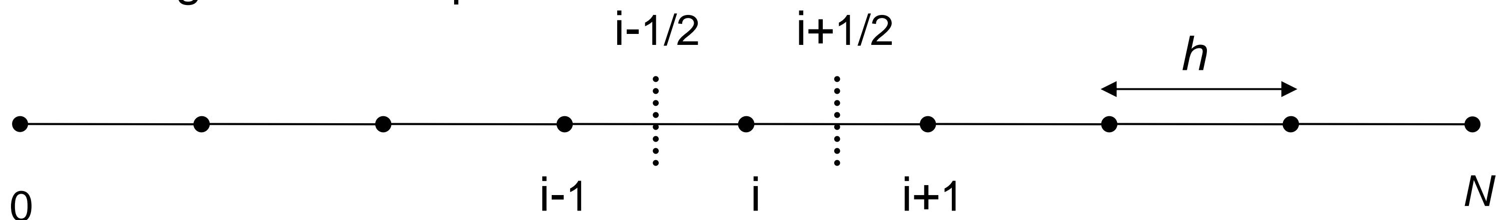
We can now use our favourite summation routine to find the answer.

The foregoing is really the basis for grid partitioning.

Basic principles of explicit and implicit schemes

Basic principles of explicit and implicit schemes

Consider a grid of $N + 1$ points



The numerical approaches to solving grid type problems are generally based on explicit or implicit solvers. The methods can be described as :

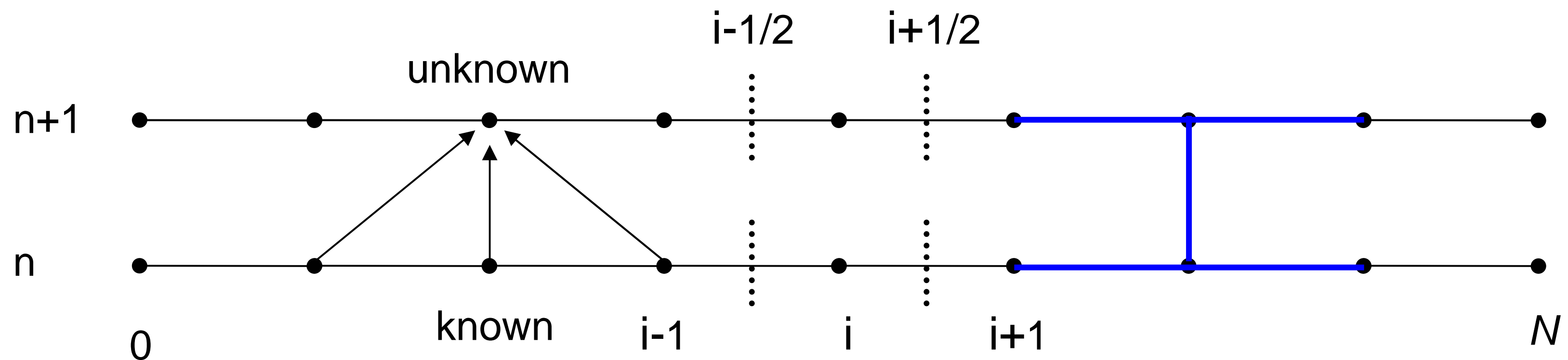
Explicit $$f_{n+1} = f_n + h\, f\left(t_n, y_n\right)$$

Implicit $$f_{n+1} = f_n + h\, f\left(t_{n+1}, y_{n+1}\right)$$

Explicit schemes depend on known data whilst implicit schemes require the solution of an equation because more than one unknown appears.

Basic principles of explicit and implicit schemes

Consider the 1D Poisson model equation $$L(\phi) = \frac{\partial^2 \phi}{\partial x^2} - g(x) = 0$$



Explicit $\phi^{n+1} = f\left(\phi^{n}\right)$

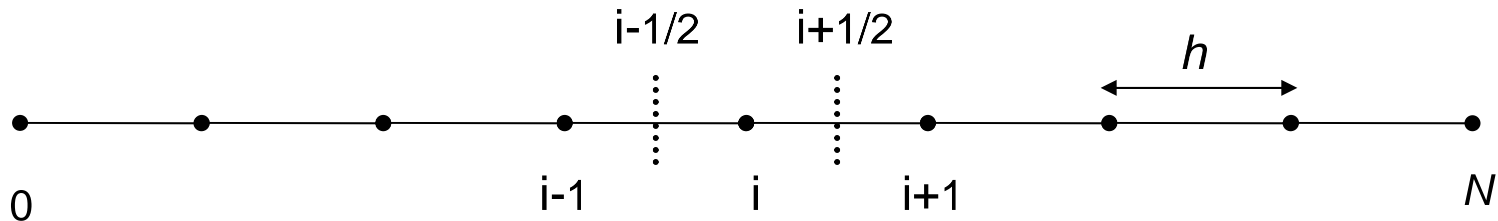
Implicit $\phi^{n+1} = f\left(\phi^{n+1}\right)$

e.g. Jacobi, Runge-Kutta

e.g. Crank-Nicolson, Stone

Finite difference approximation to model problem

Consider model equation $$\frac{\partial^2 \phi}{\partial x^2} = g(x)$$



Subject to boundary conditions $\phi_0 = \phi(0) \qquad \phi_1 = \phi(1)$

in the domain $\Omega^h = \{x : 0 \le x \le 1\}$

In the above, $h = 1/N$ and $x_i = ih$ and right hand side values are known.

The ODE is now replaced by a finite difference approximation

Finite difference approximation to model problem

The finite difference approximation at the *i*'th grid point is given by

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{h^2}$$

In 2D, the finite difference approximation at the *ij*'th grid point is given by

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \approx \frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{h_x^2} + \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{h_y^2}$$

These approximations are 2nd order accurate. Higher order schemes are clearly possible. We will now look at solving these problems in parallel.

Solution algorithms to model problem

The first thing to be aware of is that the best sequential algorithm may not be the best parallel algorithm.

For elliptic problems, like Poisson's equation, there are very fast and efficient direct methods. These methods, e.g. Gaussian elimination, solve the system of equations exactly (to machine accuracy) in a finite number of steps. There are also very efficient Fast Fourier Transform (FFT) methods and cyclic reduction. Parallel versions of such schemes already exist but they are often for specialised problems. For example, Direct Numerical Simulation (DNS) solves the Navier-Stokes equations without approximations and resolves all turbulent length scales. The problems are often assumed to be periodic and the FFT algorithm is ideal.

Solution algorithms to model problem

Although direct methods can be considered “optimal”, they suffer from a number of important limitations:

A rectangular domain is required;

The size of the coefficient matrix;

A large storage requirement;

Boundary conditions - in practice, very few problems are periodic in all spatial directions. Most problems have solid walls and complex geometries making the FFT and direct methods difficult algorithms to use.

We will therefore focus on iterative methods.

Solution algorithms to 1D model problem – Jacobi iteration

Jacobi's method is explicit. For the 1D problem with $g = 0$ it can be written as

$$\phi_i^{k+1} = \frac{1}{2}\left[\phi_{i-1}^k + \phi_{i+1}^k\right]$$

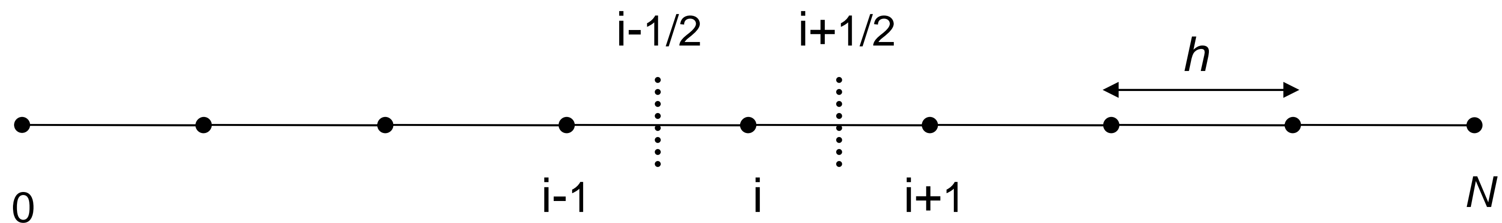
For the 2D problem with $g = 0$ it can be written as

$$\phi_{i,j}^{k+1} = \frac{1}{2\left(1+\beta^2\right)}\left[\phi_{i-1,j}^k + \phi_{i+1,j}^k + \beta^2\left(\phi_{i,j-1}^k + \phi_{i,j+1}^k\right)\right]$$

$$\beta^2 = h_x^2 / h_y^2$$

Jacobi's method in parallel

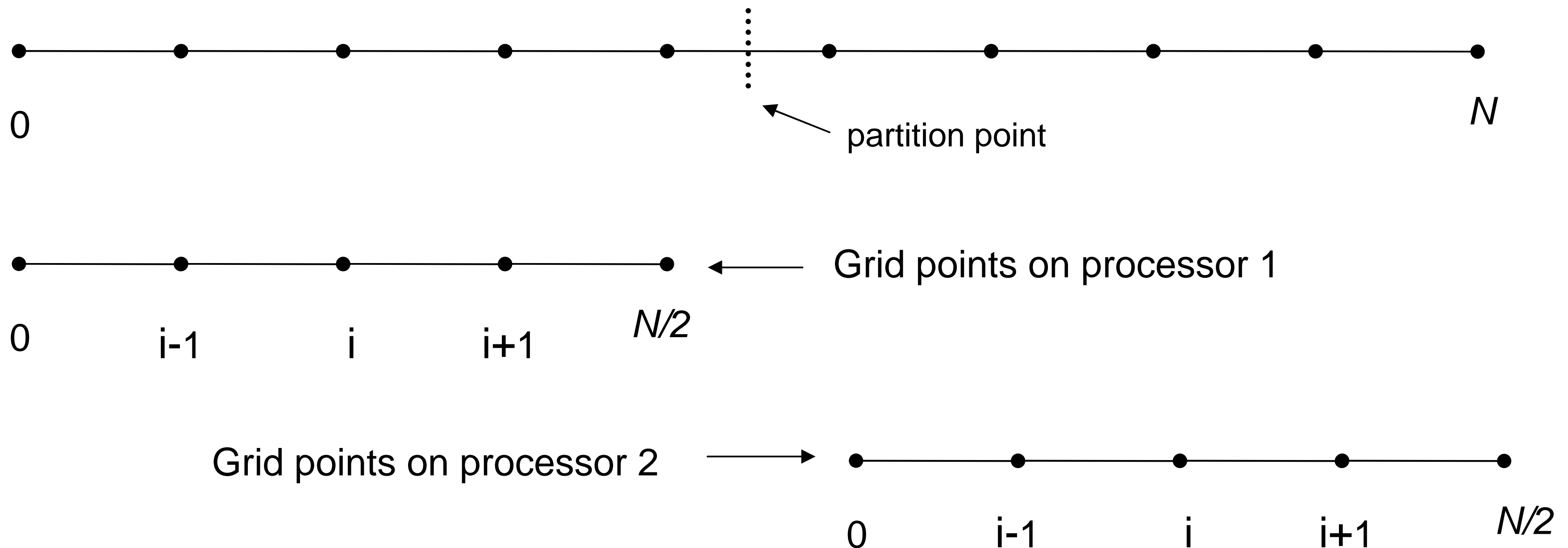
How does this problem differ from the functional decomposition?



In the case of functional decomposition all data was local and communication was only needed to get the complete answer. However, we now have derivatives to calculate and this requires data from neighbouring points.

Jacobi's method in parallel

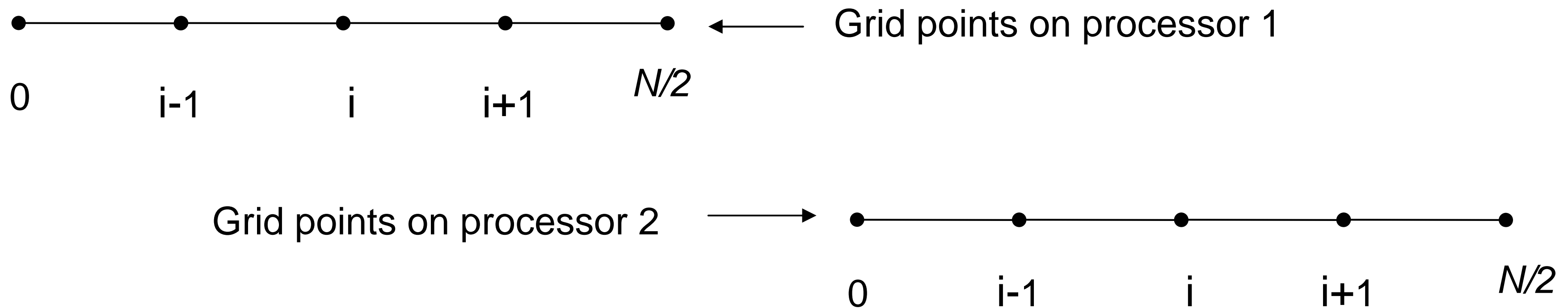
Consider our model problem grid being *partitioned* into two and each domain containing N/2 grid points:



Jacobi's method in parallel

To save storage, each processor contains arrays with size N/p and NOT N. In this case, $p = 2$.

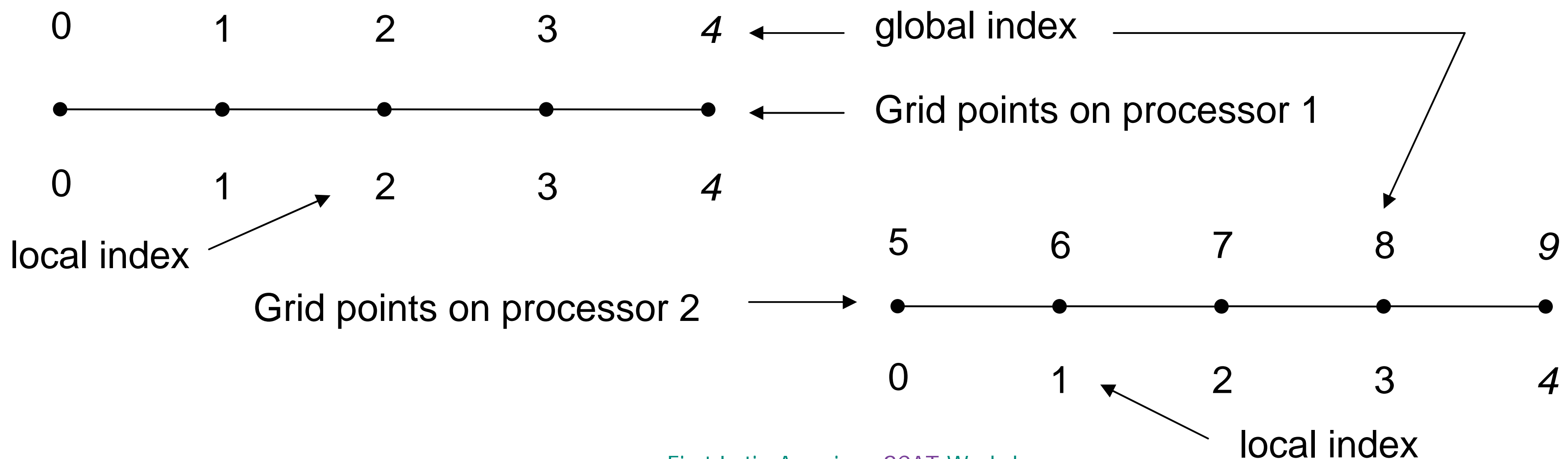
What about boundaries? Processor 1 knows about x = 0 but no longer knows anything about x = 1 (i.e. at N). Similarly, processor 1 doesn't know anything about the boundary at x = 0 (remember, x = 0 on processor 2 actually corresponds to the physical location $x = N/2*h$)



Jacobi's method in parallel

What happens at the interface? Recall, we have to solve

$$\phi_i^{k+1} = \frac{1}{2}\left[\phi_{i-1}^{k} + \phi_{i+1}^{k}\right]$$

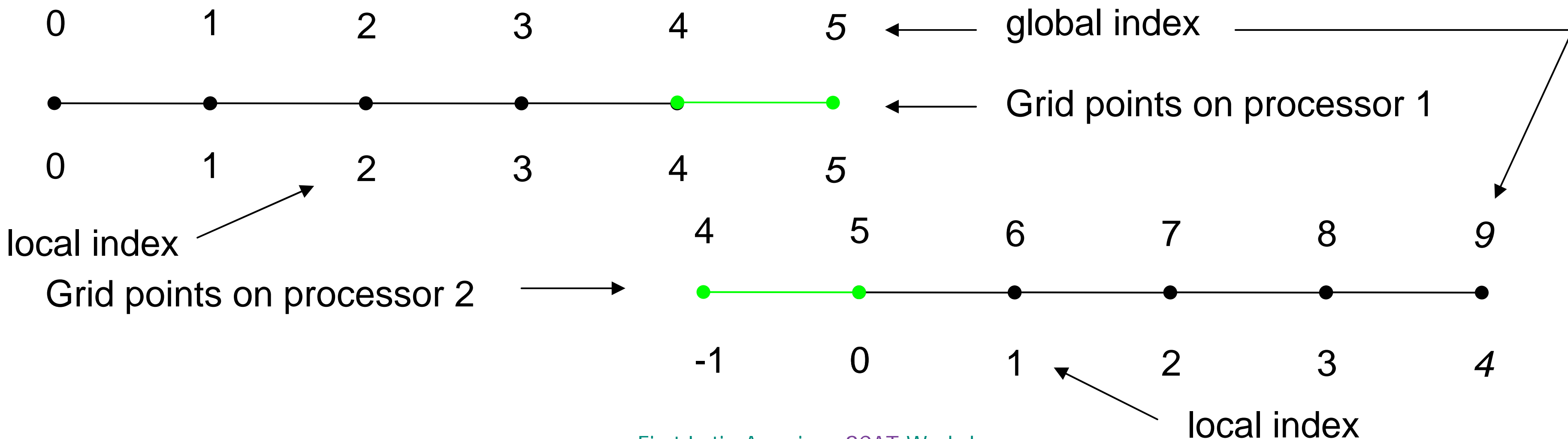


Jacobi's method in parallel – solving the interface

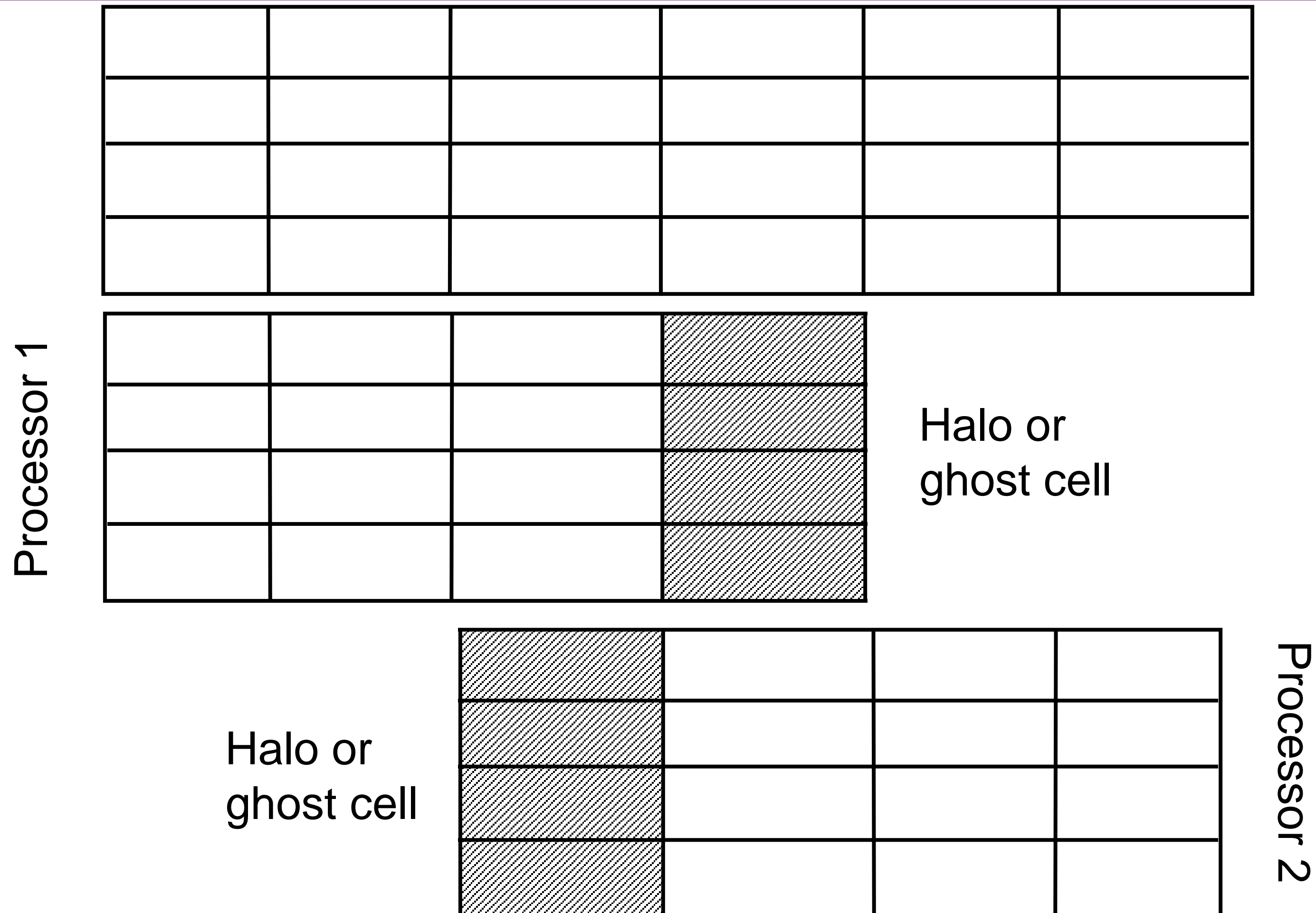
On processor 1 and 2 (using global notation) we have to solve

$$\phi_4^{k+1} = \frac{1}{2}\left[\phi_3^k + \phi_5^k\right] \qquad \phi_5^{k+1} = \frac{1}{2}\left[\phi_4^k + \phi_6^k\right]$$

This is achieved by introducing a *halo* region.



The interface in 2D for a single block grid partition



Jacobi's method in parallel – solving the interface

The halo region allows us to calculate derivatives at the partition interface.

The number of halo points required usually depends upon the order of the derivative i.e. second order accuracy will require 2 halo points for the computational stencil i.e. consider a simple upwinding scheme

$$\frac{\partial f}{\partial x} \approx af_i + bf_{i-1} + cf_{i-2}$$

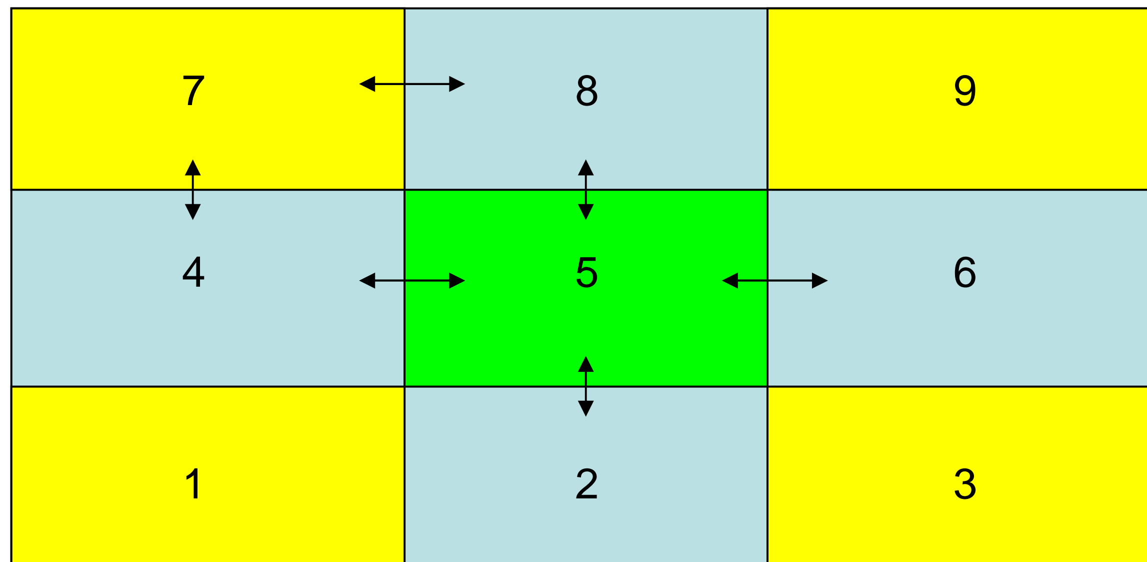
The derivative requires information from two grid points away. If higher order accuracy is required, more halo points can be added.

As with the functional decomposition, the "missing" data is transferred using communication primitives e.g. MPI_SEND/MPI_RECV, which will be explained in detail in the MPI course. Note, we clearly have to send just the required data and place it in the appropriate array location.

Jacobi's method in parallel

If we have more than two processors, only the processors related to the physical boundary will have one interface. All other processors will have two partition interfaces and a halo region at each end.

In 2D, there could be a mix 2, 3 and 4 interface regions that will depend upon the partitioning strategy employed. In figure, procs 1, 3, 7, & 9 have two interfaces, procs 2, 4, 6, 8 have three, and proc 5 has four. 3D is similar.



Jacobi's method in parallel – slight recap

We can now, in principle, partition our computational grid into any number of separate domains (depending on how many processors we have, the problem size etc.). The solution will be converged when we achieve some specified tolerance e.g.

$$\varepsilon = \sum_{i=1}^{N-1} \left| \phi_i^{k+1} - \phi_i^k \right|$$

Other criteria are also possible, such as L_2 norm.

Note: analogous to the functional approach, each processor only has a partial residual. A global residual is needed and this is done using a global sum.

Some questions we still need to answer:

How many iterations will it take to converge?

Is the number of iterations affected by the partitioning?

Jacobi's method in parallel

How many iterations will it take to converge?

This will depend on the problem and the grid size.

In fact, Jacobi's method is a poor algorithm and it frequently fails to converge because it is not good at removing the low frequency errors.

Is the number of iterations affected by the partitioning?

As Jacobi's method is explicit there is NO dependence on the partitioning. It will converge in the same number of iterations. If it doesn't, you've done something wrong!

This makes it an ideal test case for developing your parallel computing skills.

A slight digression - measuring parallel performance

Let’s assume we have our parallel Jacobi method working. We need to get some idea of how the parallel version performs. This is measured by the speed-up which is defined as

$$S_p = \frac{t_1}{t_p}$$

where t_1 is the computational time on one processor and t_p is the time on p processors.

The parallel execution time depends on two factors – the processor speed (which will be the same for both single and multi-processor runs)* and the communication performance.

* processor performance is strongly affected by cache utilisation.

Communication performance

There are two aspects to communication performance

$$T_{comm} = t_0 + N / M$$

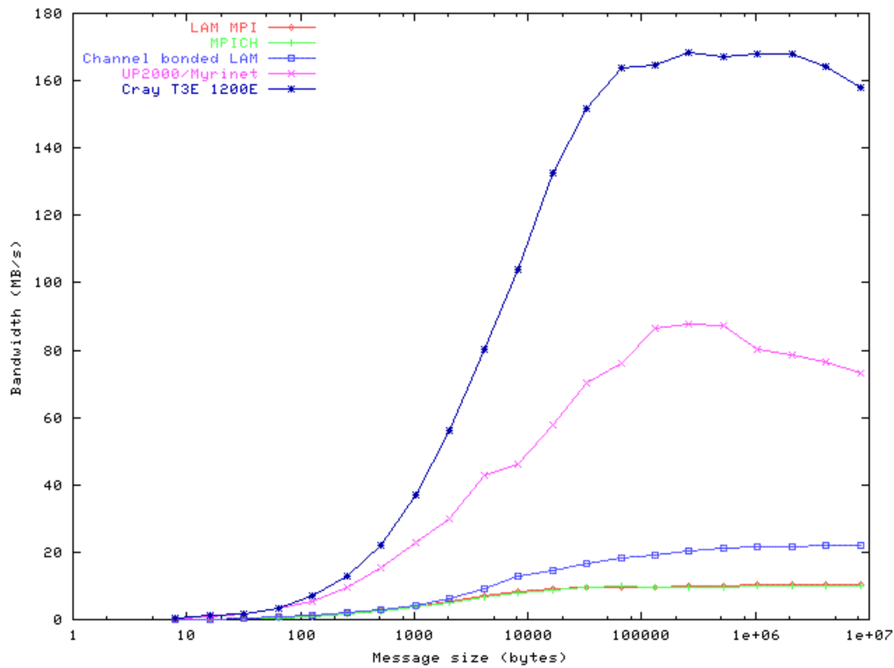
where N is the message length, t_0 is the start-up time (or *latency*), and M is the bandwidth. The user can only affect the message size (i.e. packing data together) – the latency and bandwidth are a property of the interconnect.

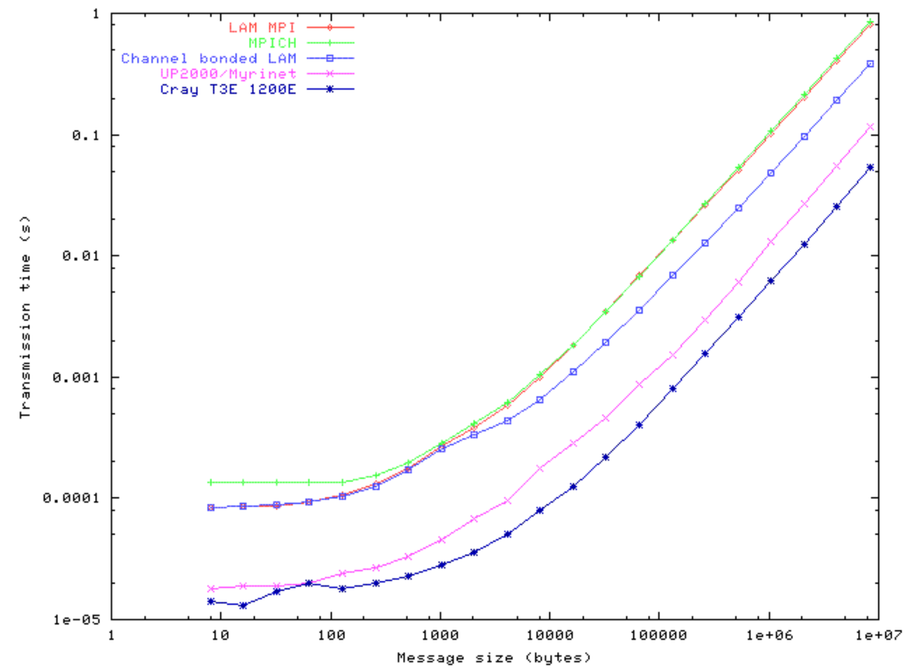
The latency will dominate for short messages e.g. global residuals.

The bandwidth will dominate for long messages.

Communication performance – graphical plot

$$T_{comm} = t_0 + N / M$$





A slight digression - measuring parallel performance

The speed-up is therefore

$$S_p = \frac{t_1}{t_p} = \frac{t_1}{T_{comm} + t_1 / p} = \frac{p}{p\left(T_{comm} / t_1\right) + 1}$$

To obtain a linear speed-up, the communication time must be << computation time.

If the problem size is fixed, the communication time will eventually start to dominate. This is known as Amdahl's law.

If the problem size is scaled with the number of processors, the scaling is usually very good. This is known as Gustafson's law.

Alternative methods in parallel – Point Gauss-Seidel

Let's now consider the Point Gauss-Seidel iteration method.

In this approach, the current values of Φ are used as soon as they are available. The algorithm for the model problem looks like:

$$\phi_{i,j}^{k+1} = \frac{1}{2\left(1+\beta^2\right)}\left[\underline{\phi_{i-1,j}^{k+1}} + \phi_{i+1,j}^{k} + \beta^2\left(\underline{\phi_{i,j-1}^{k+1}} + \phi_{i,j+1}^{k}\right)\right]$$

Consider the grid location (1,1) followed by (2,1):

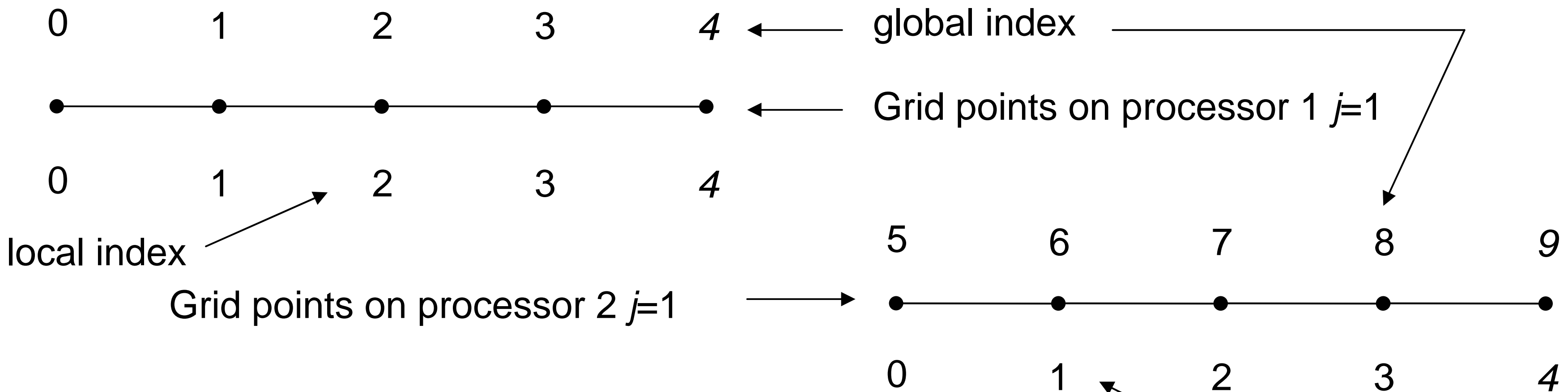
$$\phi_{1,1}^{k+1} = \frac{1}{2\left(1+\beta^2\right)}\left[\underline{\phi_{0,1}^{k+1}} + \phi_{2,1}^{k} + \beta^2\left(\underline{\phi_{1,0}^{k+1}} + \phi_{1,2}^{k}\right)\right]$$

$$\phi_{2,1}^{k+1} = \frac{1}{2\left(1+\beta^2\right)}\left[\underline{\phi_{1,1}^{k+1}} + \phi_{3,1}^{k} + \beta^2\left(\underline{\phi_{2,0}^{k+1}} + \phi_{2,2}^{k}\right)\right]$$

Alternative methods in parallel – Point Gauss-Seidel

Unlike Jacobi's method, the Point Gauss-Seidel is recursive.

What does this mean in parallel? Consider interface region along j = 1.



$$\phi_{5,1}^{k+1} = \frac{1}{2\left(1+\beta^2\right)}\left[\underline{\phi_{4,1}^{k+1}} + \phi_{6,1}^{k} + \beta^2\left(\underline{\phi_{4,0}^{k+1}} + \phi_{4,2}^{k}\right)\right]$$

On proc 2, the solution requires the value of (4,1) at iteration level k+1.

Alternative methods in parallel – Point Gauss-Seidel

In parallel, all processors start solving the problem at the same time.

For the Point Gauss-Seidel method on 2 processors, this means:

Processor 1 has the correct value at k+1 (i.e. same as sequential version)

Processor 2 starts with the "wrong" value but after the interface data is transferred, it has a better estimate. Processor 2 therefore *lags* processor 1 in the solution.

Unlike Jacobi's method, the number of iterations for the Point Gauss-Seidel method to converge will change with the number of processors!

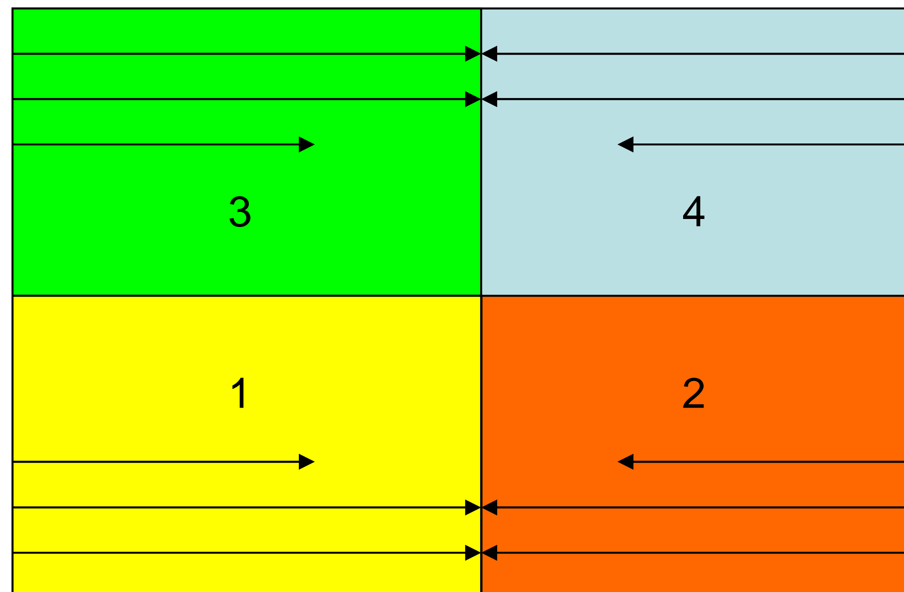
$$\phi_{2,1}^{k+1} = \frac{1}{2\left(1+\beta^2\right)}\left[\underline{\phi_{1,1}^{k+1}} + \phi_{3,1}^{k} + \beta^2\left(\underline{\phi_{2,0}^{k+1}} + \phi_{2,2}^{k}\right)\right]$$

Alternative methods in parallel – Point Gauss-Seidel

What other technique can be used?

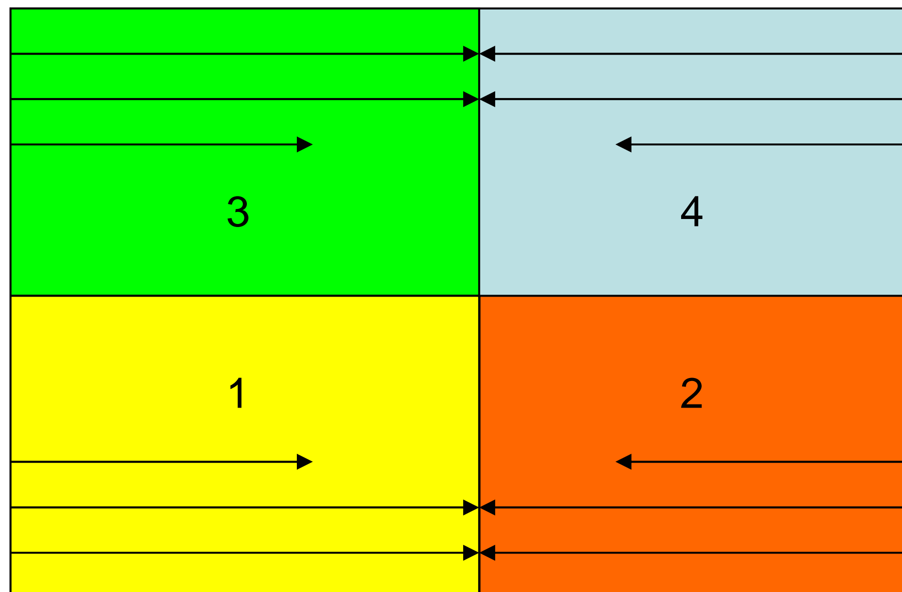
Consider splitting a 2D problem for 4 processors. Values (boundary points) are known along global locations $i = 0$, $j = 0$, $i = N$, $j = M$.

We could therefore sweep left to right on procs 1 & 3 and right to left on 2 & 4.



Alternative methods in parallel – Point Gauss-Seidel

Although offering some attractive features (probably improved convergence), is has several disadvantages. It will be more awkward to program and will not scale to many processors.



The Point Gauss-Seidel method in parallel

In practice, the Point Gauss-Seidel will converge but it generally requires more iterations than the sequential version. As a rule, expect about 10% more iterations.

What about other related methods? In principle, all other implicit schemes will exhibit similar trends so the Gauss-Seidel method provides an ideal prototype.

Alternative implicit methods

Line Gauss-Seidel iteration method.

$$\phi_{i-1,j}^{k+1} - 2\left(1+\beta^2\right)\phi_{i,j}^{k+1} + \phi_{i+1,j}^{k+1} = \left[-\beta^2\left(\underline{\phi_{i,j-1}^{k+1}} + \phi_{i,j+1}^{k}\right)\right]$$

This results in a tridiagonal matrix to solve.

Point Successive Over-Relaxation method.

$$\phi_{i,j}^{k+1} = \left(1-\omega\right)\phi_{i,j}^{k} + \frac{\omega}{2\left(1+\beta^2\right)}\left[\phi_{i-1,j}^{k+1} + \phi_{i+1,j}^{k} + \beta^2\left(\phi_{i,j-1}^{k+1} + \phi_{i,j+1}^{k}\right)\right]$$

When omega = 1, the Point Gauss-Seidel method is recovered. The trick is to find the “optimum” value of omega.

There are many variants on the above, including Alternating Direction Implicit (ADI), “red-black” Gauss-Seidel,…….

Conjugate Gradient methods

- There are many flavours of conjugate gradient methods but the system we are looking at is symmetric positive definite. The basic method is ideal for our purpose.
- In a similar manner to the various iterative schemes, understanding how the basic conjugate gradient scheme works in parallel means that other conjugate gradients methods can be understood e.g. conjugate gradient squared.
- We now re-write our system of equations in matrix form i.e. $Ax = b$

Conjugate Gradient methods – matrix form

$$\begin{bmatrix} \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 & 0 \\ 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 & 0 \\ 0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 & 0 \\ 0 & 0 & 1 & \alpha & 1 & 0 & 0 & \beta^2 \\ \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 & 0 \\ 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 & 0 \\ 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha & 1 \\ 0 & 0 & 0 & \beta^2 & 0 & 0 & 1 & \alpha \end{bmatrix} \begin{bmatrix} \phi_{1,1} \\ \phi_{2,1} \\ \phi_{3,1} \\ \phi_{4,1} \\ \phi_{1,2} \\ \phi_{2,2} \\ \phi_{3,2} \\ \phi_{4,2} \end{bmatrix} = \begin{bmatrix} b_{1,1} \\ b_{2,1} \\ b_{3,1} \\ b_{4,1} \\ b_{1,2} \\ b_{2,2} \\ b_{3,2} \\ b_{4,2} \end{bmatrix}$$

$$\alpha = -2\left(1+\beta^2\right)$$

Note, full system not shown!

Conjugate Gradient methods – basic pseudocode

$x_0 = 0; \quad r_0 = b - Ax_0$

$p_{-1} = 0; \quad \beta_{-1} = 0$

solve w_0 from $Kw_0 = r_0$

$\rho_0 = (r_0, w_0)$

For $i = 0, 1, 2, ...$

$$p_i = w_i + \beta_{i-1} p_{i-1}; \quad q_i = Ap_i; \quad \alpha_i = \frac{\rho_i}{(p_i, q_i)}$$

$$x_{i+1} = x_i + \alpha_i p_i; \quad r_{i+1} = r_i - \alpha_i q_i$$

If $\| r_{i+1} \|_2 < \varepsilon$ stop

$$w_{i+1} = K^{-1} r_{i+1}; \quad \rho_{i+1} = (r_{i+1}, w_{i+1}); \quad \beta_i = \rho_{i+1} / \rho_i$$

end

In the above, *K* is a preconditioning matrix and (…,…) are dot product operations.

Conjugate Gradient methods – basic pseudocode

For convenience, the preconditioning matrix is set to the identity matrix.

There are several dot product operations and a global residual test for convergence. This will test the latency of the network.

Data will need to be transferred at the interface regions and will test the bandwidth of the network.

When $K = I$, the parallel algorithm performs analogously to the Jacobi explicit scheme i.e. the number of iterations required to converge does not depend on the partitioning.

In practice, choosing a good preconditioner is needed to obtain good convergence and this remains an active area of research. For many CFD type applications, incomplete *LU* factorisation works very well.

A model test problem

The governing equation for 2D heat conduction is
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

The convergence criteria was set to
$$\varepsilon = \sum_{i=1,\, j=1}^{N-1, M-1} \left| T_{i,j}^{k+1} - T_{i,j}^{k} \right| = 0.01$$

The number of grid points used was 21 in the x-direction and 41 in the y-direction. The computational domain is $0 \le x \le 1$ and $0 \le y \le 2$ with boundary conditions: T(x,0) = T1 and T(0,y) = T(1,y) = T(x,2) = 0, with T1 = 100.

$$T(x,y) = T_1 \left[2 \sum_{n=1,3,5}^{\infty} \left(\frac{1}{n\pi} \right) \frac{\sinh\left(n\pi\left(H-y\right)/L\right)}{\sinh\left(n\pi H/L\right)} \sin\left(\frac{n\pi x}{L} \right) \right]$$

For this problem, $L = 1$ and $H = 2$.

Iteration count for Poisson equation in parallel

Method	NPX	NPY	NP	Iterations	Time (s)
J	1	1	1	6949	31.74
GS	1	1	1	3642	19.36
CG	1	1	1	150	2.90
J	2	1	2	6949	40.72
GS	2	1	2	3726	20.10
CG	2	1	2	150	2.41
J	1	2	2	6949	25.85
GS	1	2	2	3680	15.57
CG	1	2	2	150	2.63
J	2	2	4	6949	29.46
GS	2	2	4	3763	17.18
CG	2	2	4	150	2.34

Multiblock in parallel

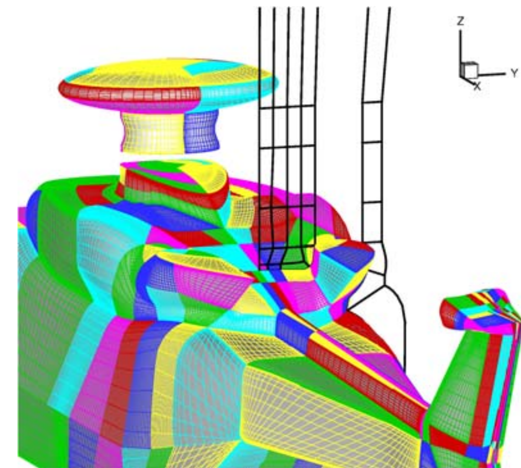
Many advanced CFD codes use multiblock to model complex geometries.

However, this approach was developed for sequential execution.

What happens in parallel?

From the previous model problems it should be clear that the convergence properties are going to be affected and the number of iterations or time-steps will change.

structured multiblock



NH19 fuselage
figure courtesy of
Barakos and Badcock (Liverpool)
2,226 blocks, over 12M cells

What about unstructured problems?

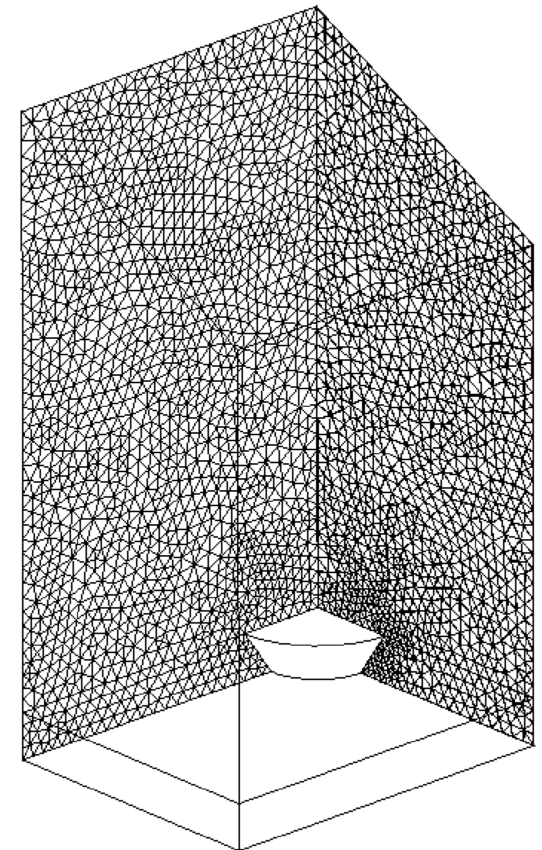
Unstructured grids have many advantages e.g. ability to handle extremely complex geometries, relatively easy to introduce grid refinement/adaptation, easier to couple with structural problems.

However, partitioning is not easy!

Fortunately there is a very efficient software package (Metis) that will do this for you.

For very, very large grids e.g. 120M+ nodes, there is a parallel version!

The major issues are related to load balancing, efficient cpu utilisation (see PETSc) presentation, …….



Unstructured grid around the Beagle 2 lander module

Multigrid methods

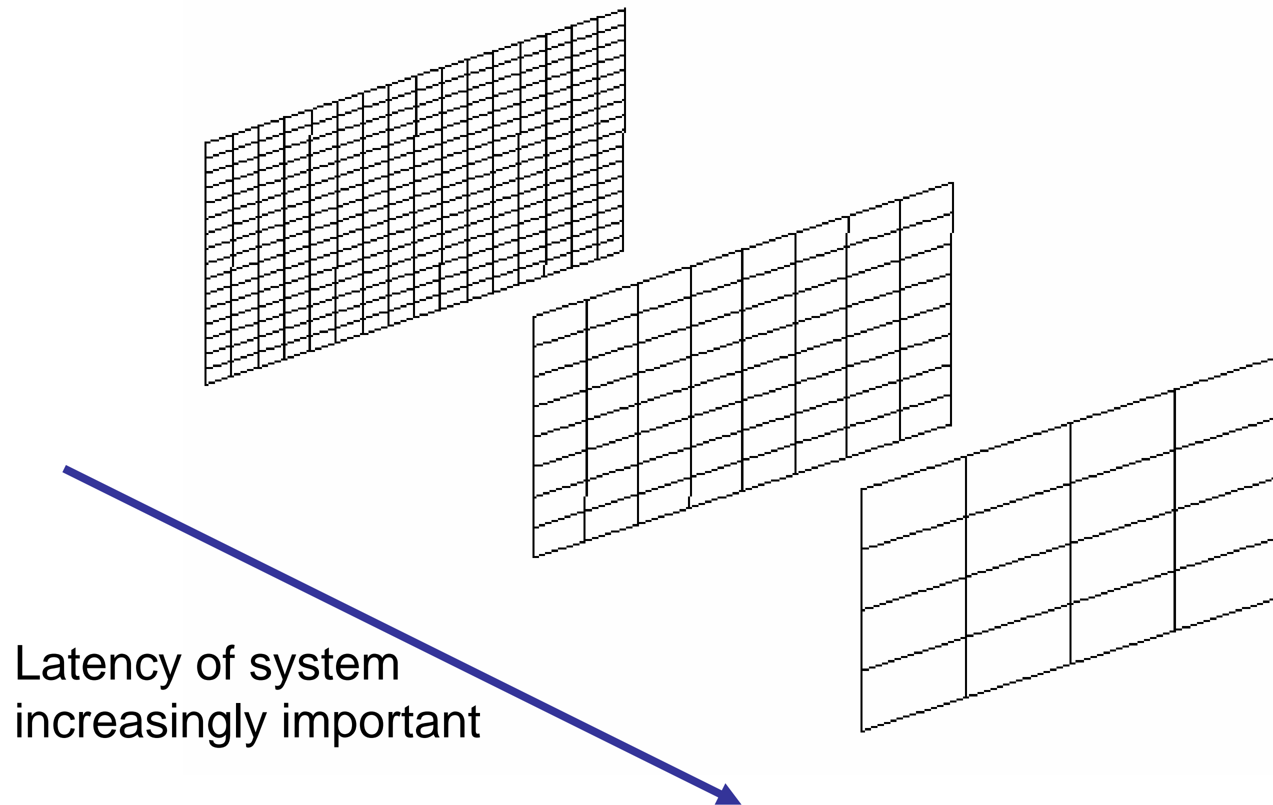
Multigrid principle:
initial mesh → create successive meshes (coarse → fine, fine → coarse)

Structured grids: relatively straightforward

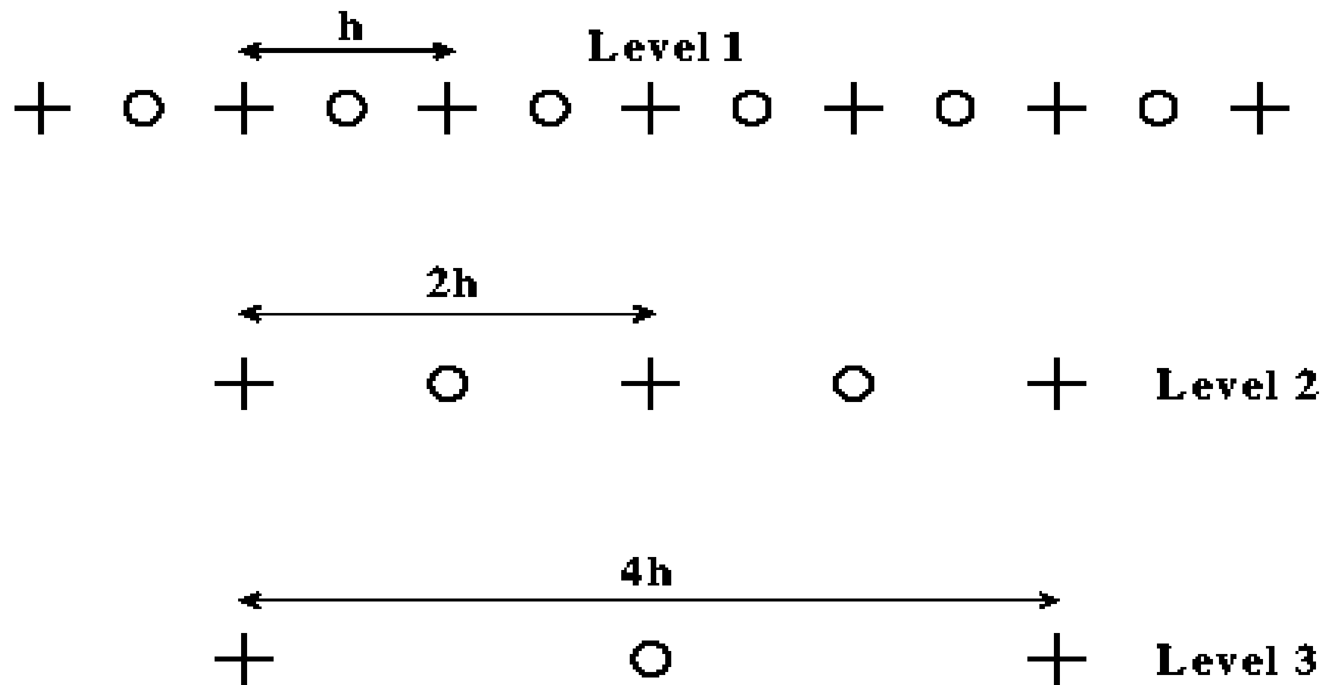
Unstructured Grids: much more challenging
Nested (derived from initial mesh) and non-nested methods
Agglomeration (coarse mesh level derived by merging cells)

Partitioning unstructured meshes: this **was** a major bottleneck and suffered severely from disconnected domains. The problem was solved in early 90s (H. Simon). Today, partitioning unstructured grids is fairly routine and the best package available is METIS (free!)

Multigrid: effect on communications



Finite Volume Multigrid

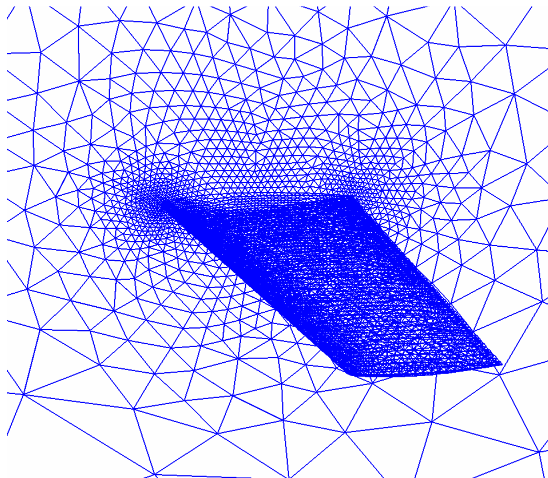


SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

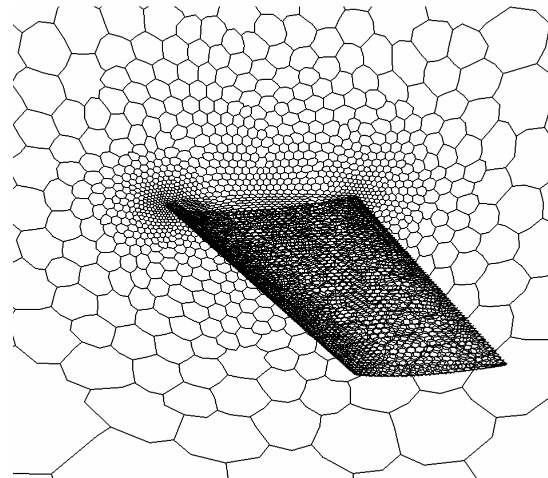
ALFA
EUROPEAID
CO-OPERATION OFFICE

Agglomeration multigrid of M6 wing

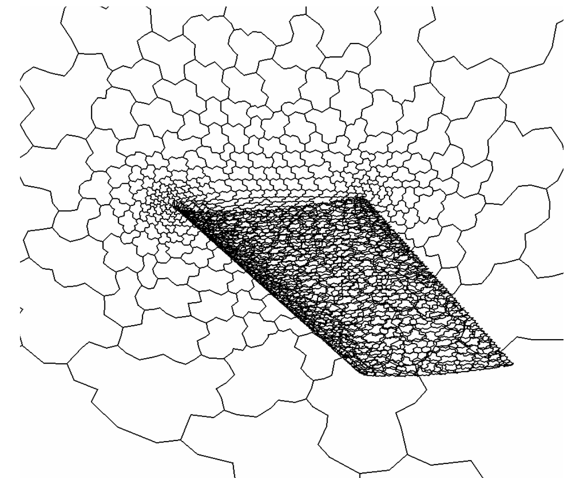
The figure shows the fine mesh (a) and its dual mesh (b)\(which shows the communication path). The final figure shows the next level using an agglomeration technique.



a

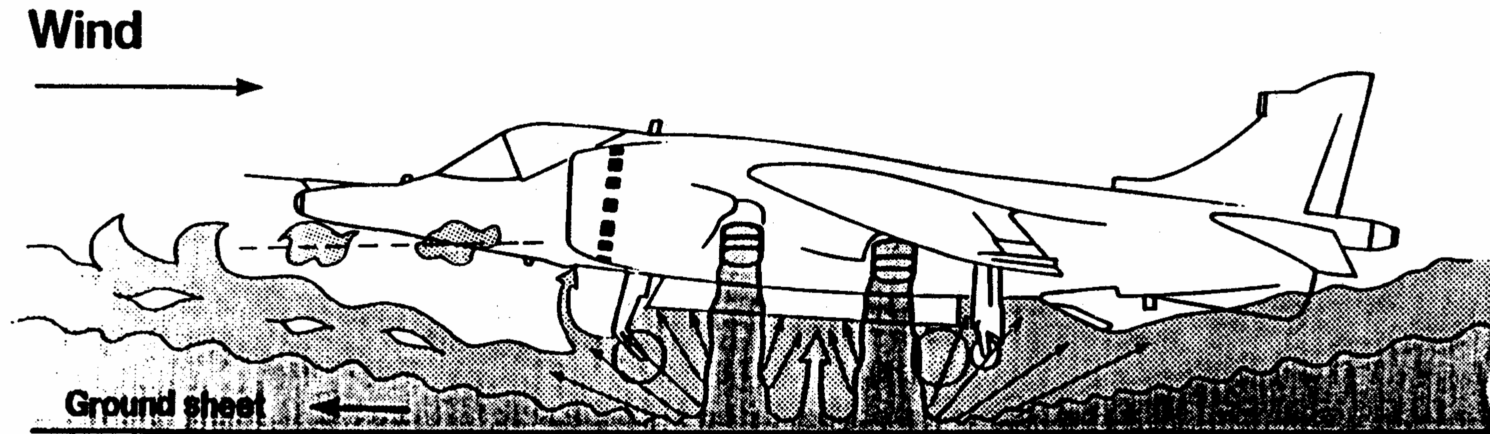


b



c

Grand challenges of the 1990s

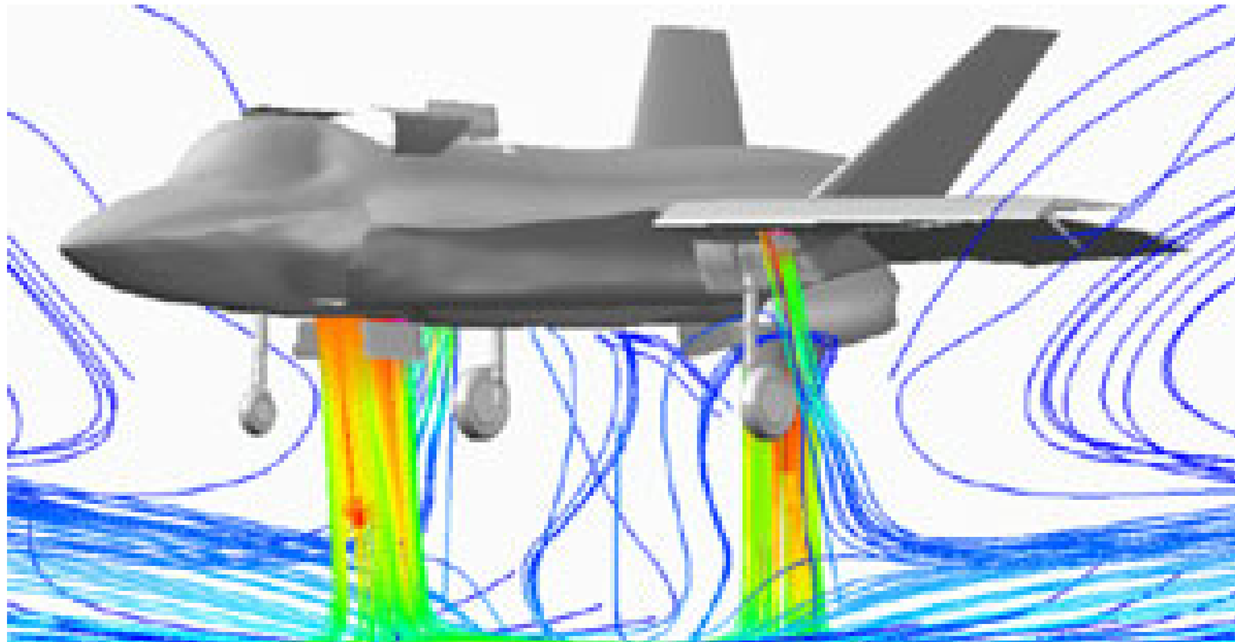


Simulation of the landing for a V/STOL aircraft including the prediction of:

- Forces
- Thermal loads (on aircraft and landing surface)
- Propulsion system interaction etc.

Goal: turn-around time low enough to enable design studies.

Challenges for 2000 and beyond



A recent simulation of a V/STOL aircraft in hover using a commercial CFD code (CFD-ACE+). These calculations are now within the grasp of engineers.

Future challenges: helicopter simulation

Helicopter simulations are among the most challenging problems facing CFD.

Recent work on HPCx has required 32M grid points around the rotor blade to capture the tip vortices.

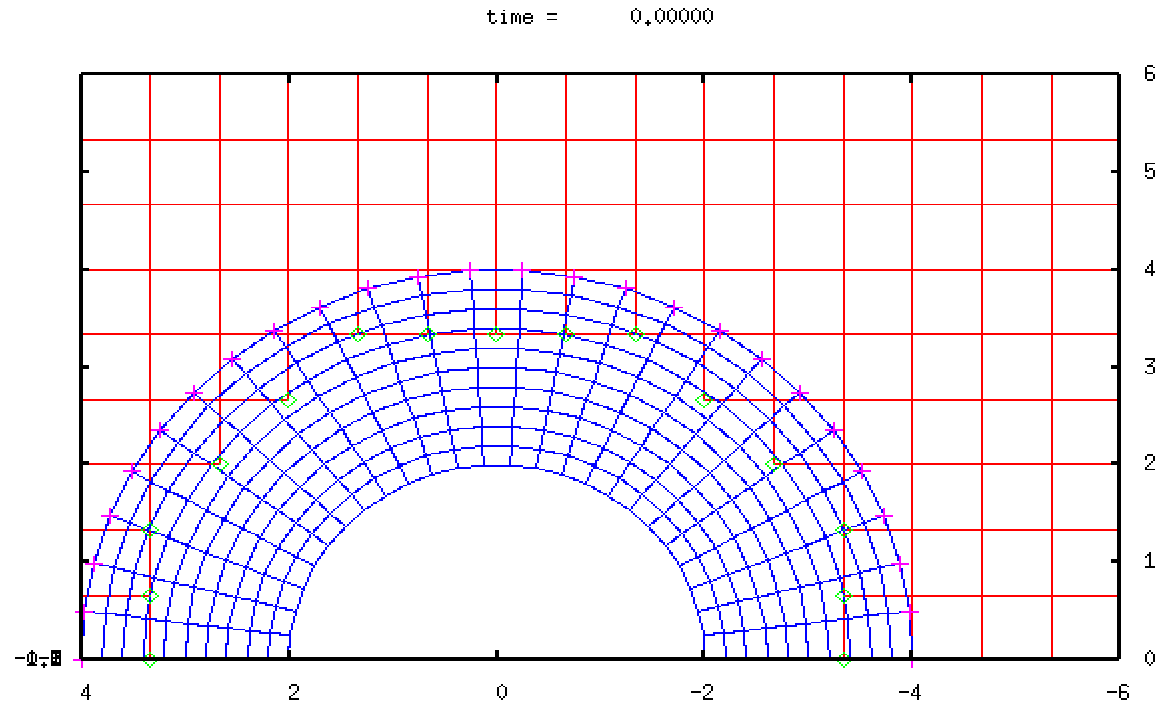
Simulating tThe landing of a H-60 helicopter on a ship may have to wait!

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

ALFA
EUROPEAID
CO-OPERATION OFFICE

Future challenges: moving/overlapping grids

This type of grid would be used for very complex intersections or for moving bodies but remain difficult problems to solve in parallel.



SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

ALFA
EUROPEAID
CO-OPERATION OFFICE

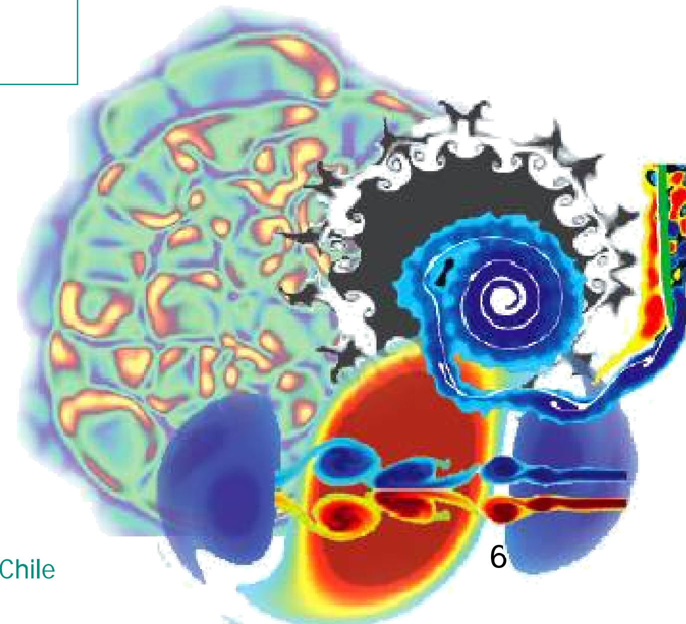
First Latin American SCAT Workshop:
Advanced Scientific Computing and Applications

Introduction to Grid Partitioning

Prof. David Emerson

CCLRC Daresbury Laboratory

University of Strathclyde



CCLRC

University of Strathclyde Glasgow