Chapter 3

Design for Parallel

Single processor design

fastest; smallest; …. algorithm

You have a single processor so it corresponds to how you might solve the problem.

Multi-processor design: need to design in parallelism.

Simulations are embarrassingly parallel

Communications are the biggest problem.

Bandwidth is much more limited than cpu cycles.

The CPUs are running independently and it is difficult to provide a single consistent time - and yet communication is normally relevant only when both processors have reached a particular point, which requires synchronisation.

May wish to write to a common data structure, but then we need to limit access to the data structure to a single process.

Both problems have solutions, but neither are simple and both tend to slow down the calculation.

Minimal communication – ideally zero between processors.

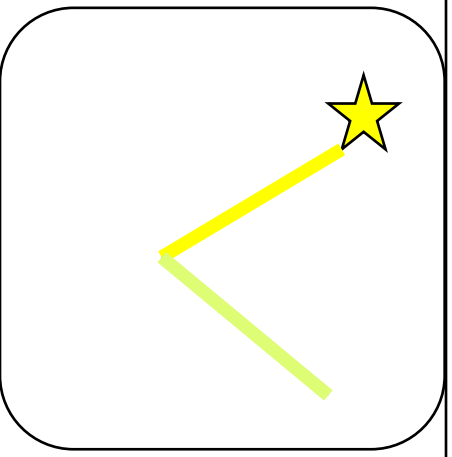
Example

For a film you need 25 frames a second. Each frame takes 4 seconds to generate. Assuming perfect scaling with 100 processors you can generate a frame every 1/25 of a second on average.

Simulations are embarrassingly parallel

Trivial for creating a film. One frame per free processor: first frame takes 4 seconds and after that they arrive at 25 per second.

Put them in order and display



Hard for real time generation in say a game – the image is not ready until the user has made a decision – a frame cannot be generated until user action, thus we need to split the image into 100 bits and generate them all separately.

Light source in one part of the image, reflects off object in a second part modifying colour and illuminates something in a third part.

Communication – potentially non-local

Data Parallelism

different parts of the data are processed on different units. Same code, different data, but not SIMD where the processors are executing instructions in lockstep.

Functional Parallelism

different subtasks are executed on different processors *MPMD* (multiple programme multiple data).

A particular processing step executed for all the data,

Results passed on to further processors for further processing

(Must be possible to pass on part of the data set)

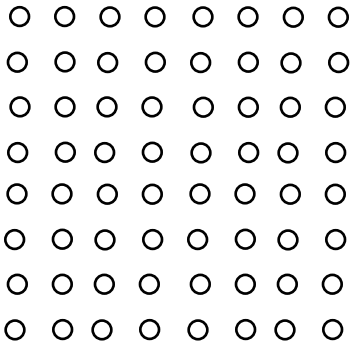
Historical note

Prior to invention of digital electronic computers – "computers" were people who worked together in large rooms doing calculations. On the Los Alamos Project Richard Feynman speeded up calculations by working out a way to parallelise them

Continuous system (differential equation) – replaced by a discrete system – Geometrical Grid – and Finite difference equations.

Not a grid in the module title

Electric fields, magnetic fields, temperature distribution, stress calculations, diffusion, weather, fluid flow.



A cpu calculates the values for a range of grid points.

These require input from a cell and some of its neighbours.

Continuous system (differential equation) – replaced by a discrete system, difference equation.

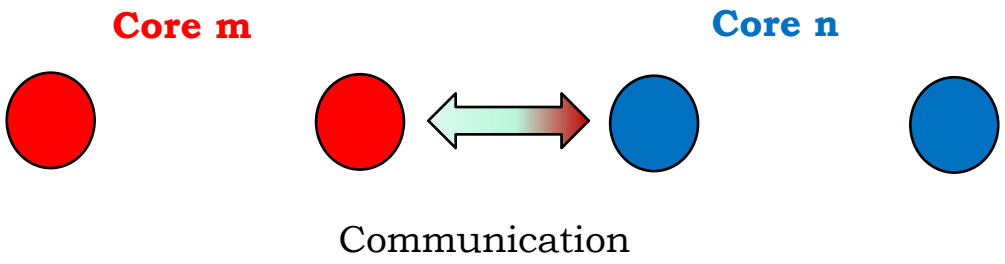
Every step in the calculation requires input from a cell and some of its neighbours. Input changes at every step.

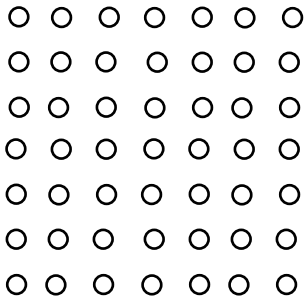
Domain Decomposition

Normally each core will deal with more than one point.

How to decide on the mapping of the points to the cores?

Communication is needed between cores to provide updated values.





Each core needs memory for points not calculated but needed for calculation ***halo*** or ***ghost*** values.

Moving through all points by a core is a ***sweep***.

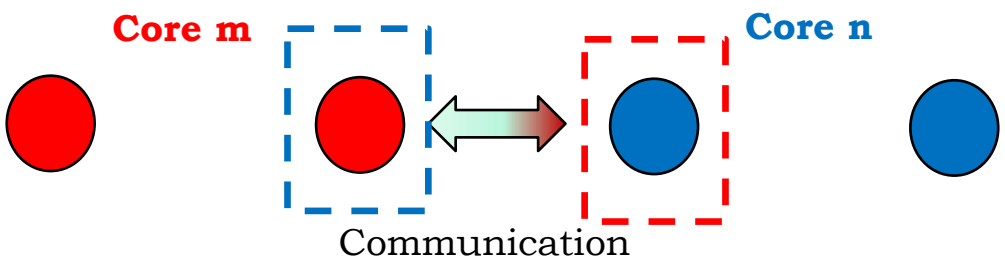
Updating occurs when all cores have finished a sweep.

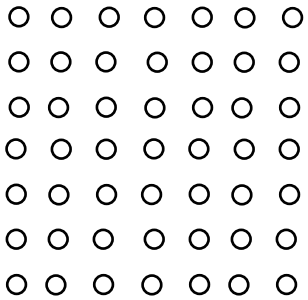
More complex update schedules may be possible

Communication is much slower than calculation

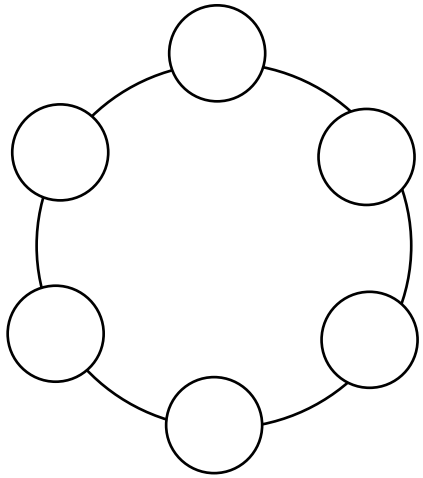
Loads must be balanced across processors – waiting is inefficient (and wastes power).

Communication overheads depends on the size of the ***stencil.*** Value, first derivative and second derivative depend on nearest neighbour. Higher derivatives require longer range communications.

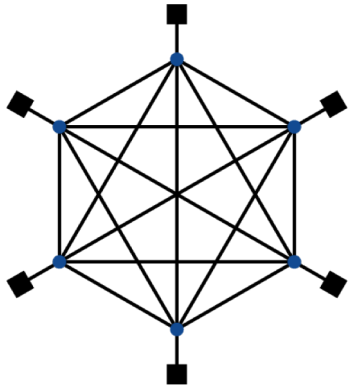




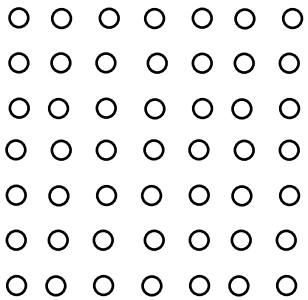
Overhead on longer range interactions depend on topology.



Ring



Fully connected



Master Slave

Trivially parallelisable

Break the problem into distinct independent problems.

CGI movie – each frame

Weather predictions – each starting point.

Master knows about the work units.

Distributes to slaves.

Collects and collates results.

Problems – bandwidth/congestion/single point of failure.

Lost jobs

This works especially well with clusters – collections of independent processors, rather than the tightly connected special purpose systems.

So cloud – although there are still significant problems to an efficient solution.

Going to talk about a way of approaching the task of solving a problem on a distributed system.

Might think that defining how a distributed system performs in relation to a single processor system is easy.

There are subtleties …

In order to quantify improvements you need to understand metrics – both to "sell" any solution that you come up with and to understand the way other people might describe their solutions.

Design always needs intelligent application

Parallel design: hard.

An approach – ideas to bear in mind.

Partition

Communication

Concurrency and scalability

Agglomeration

Mapping

Locality

11 Partition

Looking for possibilities for parallelism.

Maximum number – smallest unit of work.

Clearly increases communication

Fine grained decomposition.

May need to agglomerate later

Greatest flexibility in terms of building solution.

Easier to consolidate than divide.

Divide computation and data

OO design naturally produced good targets for parallelism.

Object includes data and code.

Objects should be as small as possible (they do one task) and with minimal coupling.

One object per processor is always possible.

Domain decomposition: break up the data and associate the code with it.

Old technique

Functional decomposition: break up the operations and associate the data.

12 Partition Check

Following checklist

Does your decomposition have at least an order of magnitude more tasks than processors?

Need flexibility

Does it avoid repeated computation and multiple storage?

Will not scale

Are the tasks equal size?

Allocation hard

Does number of tasks should scale with problem size?

If tasks get larger then solution will not scale

Do you have alternatives?

A large task which cannot be decomposed behaves like a serial part of the execution.

Communication is via a **channel**

One end sends, other end receives

Send is asynchronous

Receive is synchronous

Define

channels – links between producers and consumers.

Messages – data which travels down the channel.

Communication should be spread widely

A central communication hub is a bottleneck

Many messages should be possible *concurrently*

Communication modes

Local: only with "near" neighbours

For grid "near" means high bandwidth, low latency.

Structured: communicating nodes for a regular structure.

Static: does not change with time. Might be known at compile or submit time. *Dynamic* changes.

(A)Synchronous: both parties have to be on the line at the same time.

Look at an example of a problem.

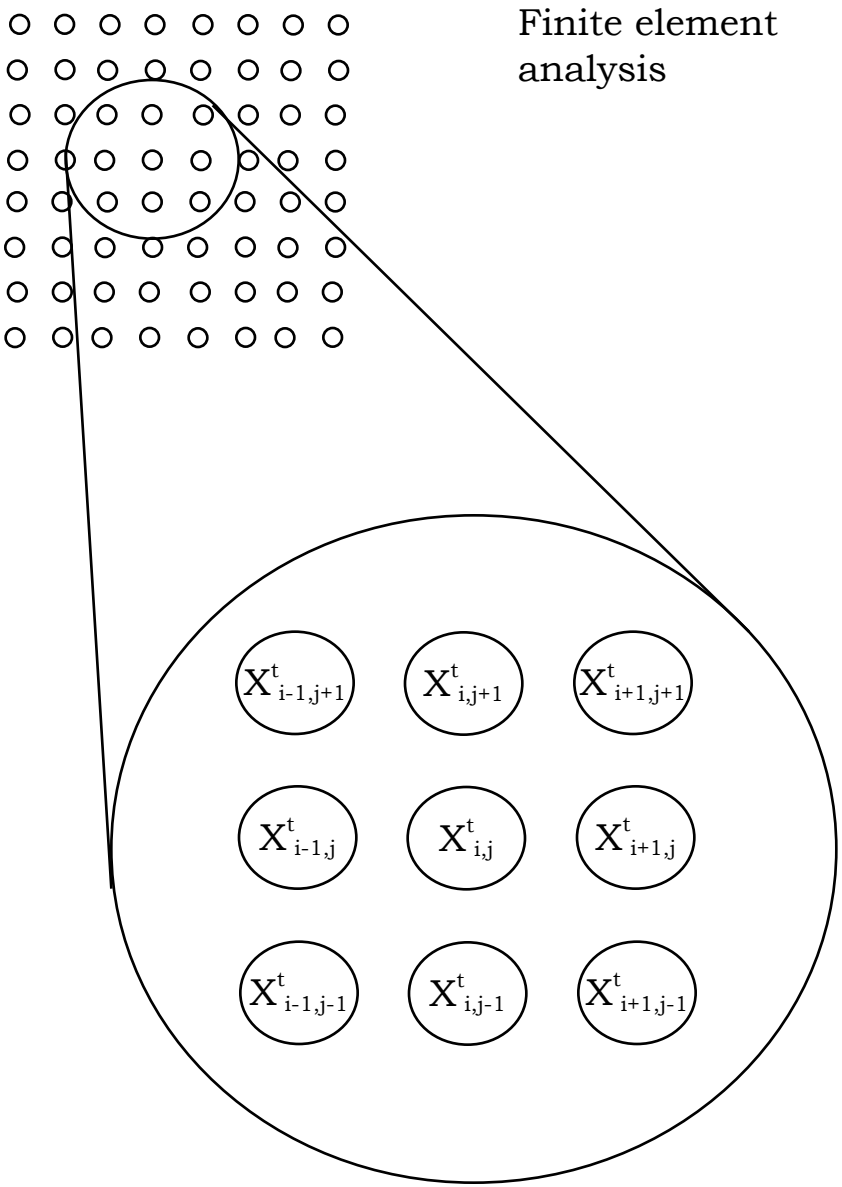
Definitely suitable for distributed solution

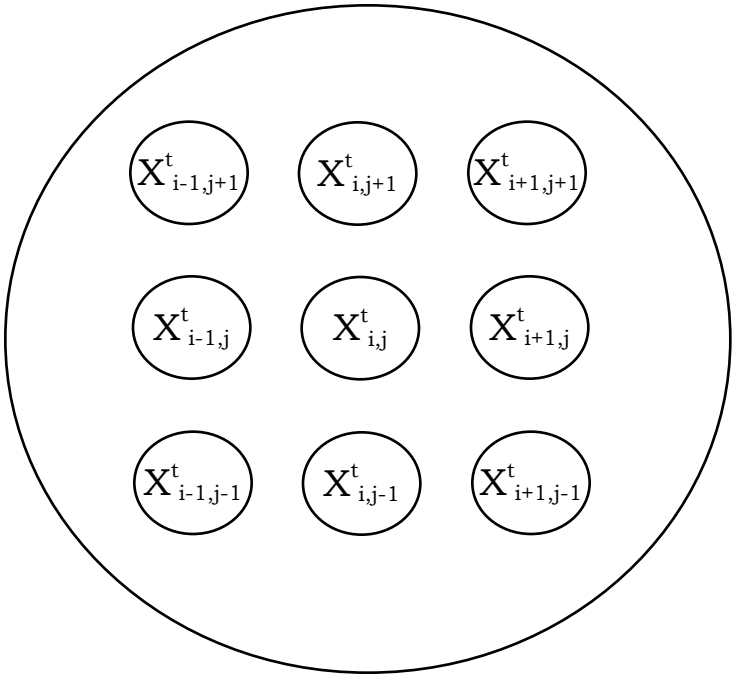
Consider communication patterns and their implication

Consider a calculation done on a rectangular grid.

(stick to 2d)

Another meaning of grid computing





Another meaning of grid computing

Used when reality corresponds to a continuous field.

At every point there is a value of a variable, which may be scalar, vector or even tensor.

Variable, can be

electric field, temperature, current water, air,…
stress, strain, gravitational field, magnetic field,
Anything described by a differential equation.

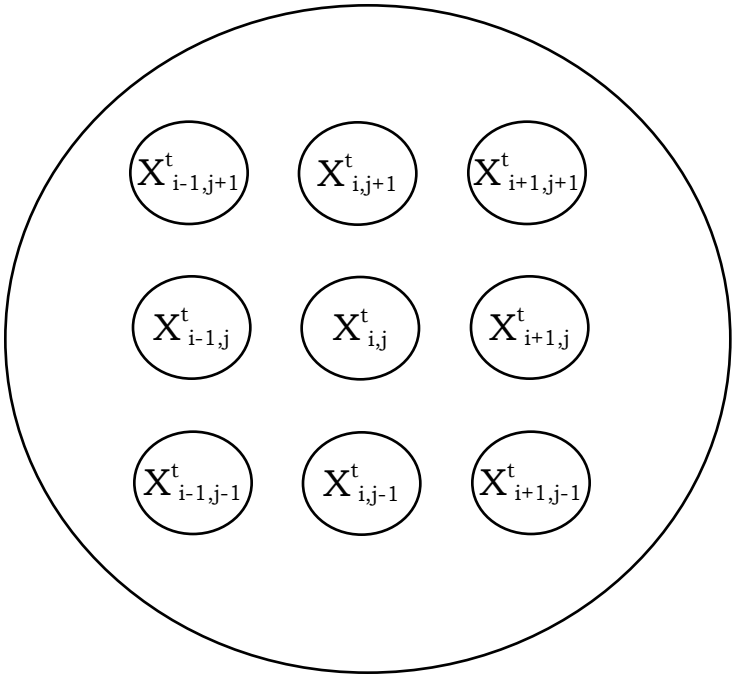
Many applications

Many problem domains

Applications

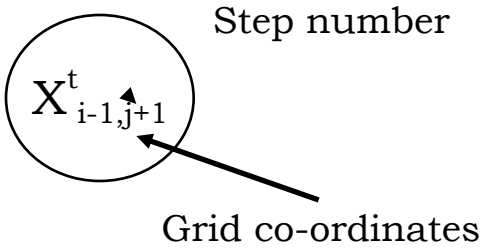
meterology, civil engineering, mechanical engineering, chemical engineering, aero engineering, …..

17 Difference from differential



Another meaning of grid computing

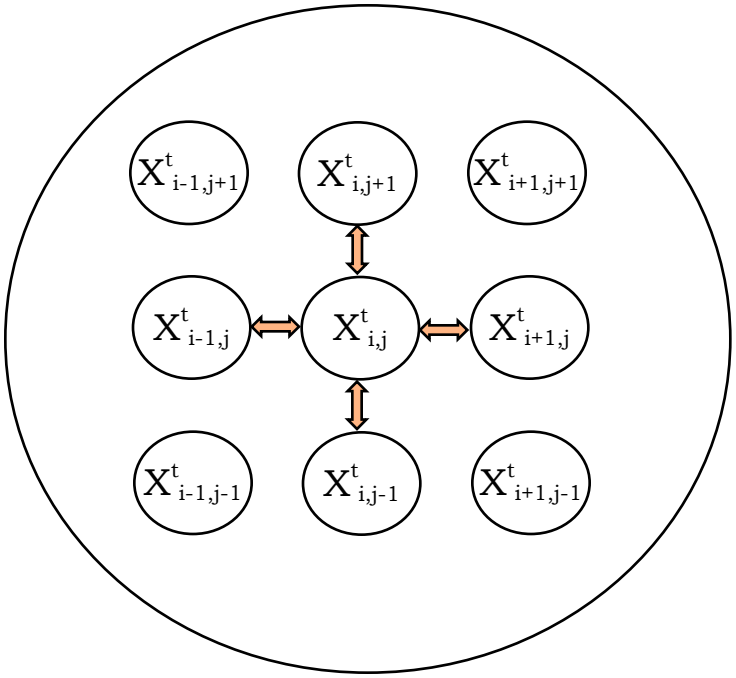
Basic idea is to use a difference equation (*derived from the differential equation)* to calculate the solution.



Many problem domains

The improved value at a grid point is derived from the values at the point coupled to values at other points.

We are looking for a method which *rapidly converges* to the correct answer.



Shown are the channels.

Each arrow is two channels

Boundary conditions

The **stencil** is the pattern of communication channels to the neighbours.

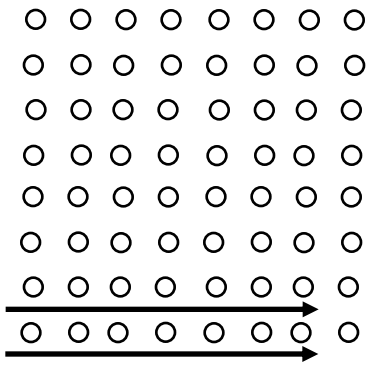
Rectangular grid and nearest neighbour leads to the **5 point stencil**.

$$\psi^t(i,j)=1/8\left[\psi^{t-1}(i,j-1) + \psi^{t-1}(i-1,j) +\psi^{t-1}(i+1,j) + \psi^{t-1}(i,j+1)+4*\psi^{t-1}(i,j+1)\right]$$

The important thing to look at here is that the values of all the points depend on the values of the four nearest neighbour at the previous step.

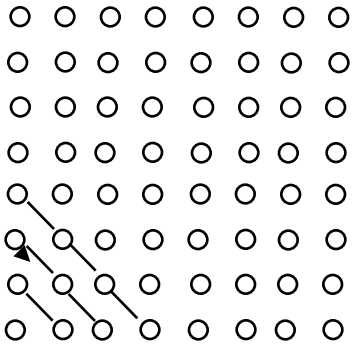
The initial update strategy seems to be a sensible one.

We update all nodes in turn. We can of course think of various orders in which to do the update.C

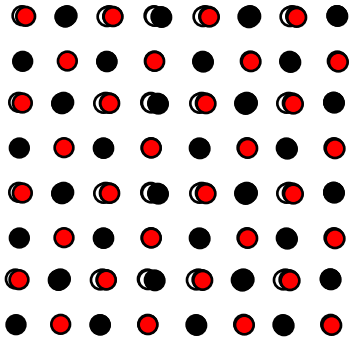


Obvious is x then y (or y then x)

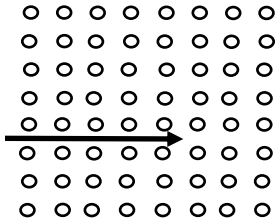
Trivial extrapolation to higher dimensions



A wavefront travelling from bottom leftt to top right



Red-black or *checkerboard*

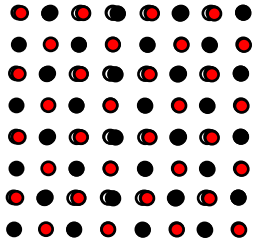


Which ever order we calculate the nodes for the simple replace *Jacobi* algorithm.

We conceptually just replace all elements at step t, with all elements at t+1 *simultaneously.*

Some coordination needed.

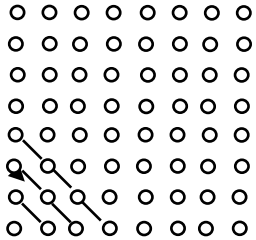
Increment a flag when you complete and look for a value. *Global.*

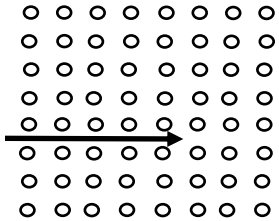


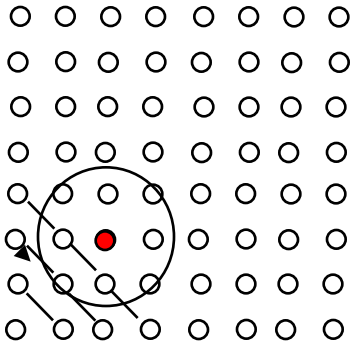
Send to the output channels when complete and listen on input channels for neighbours to finish. *Local.*

Are we guaranteed not to get out of step?

Convergence is slow.

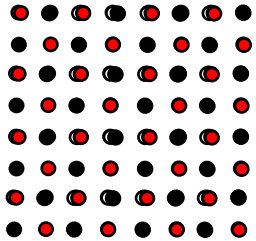


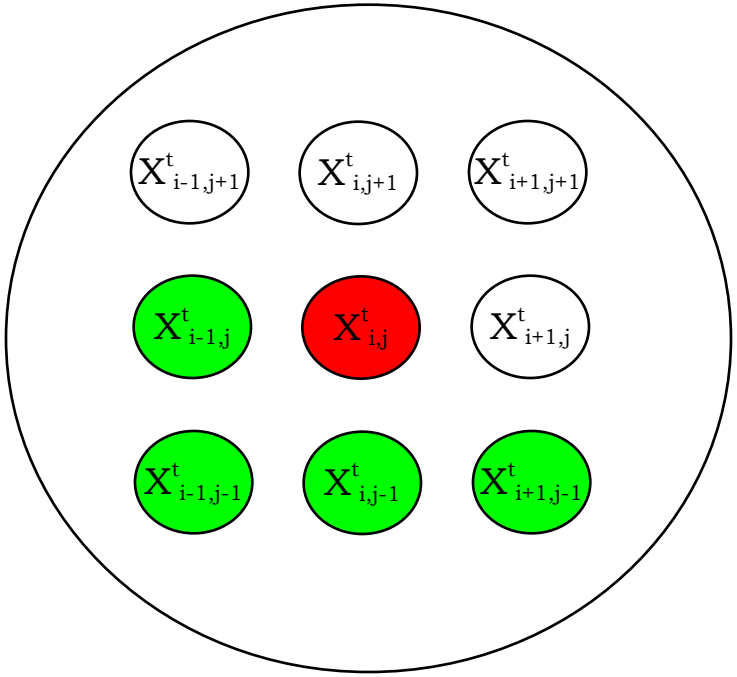




Consider a single processor *(serial)* solution.

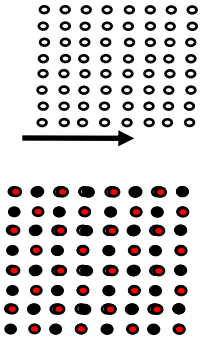
Concentrate on the marked node.

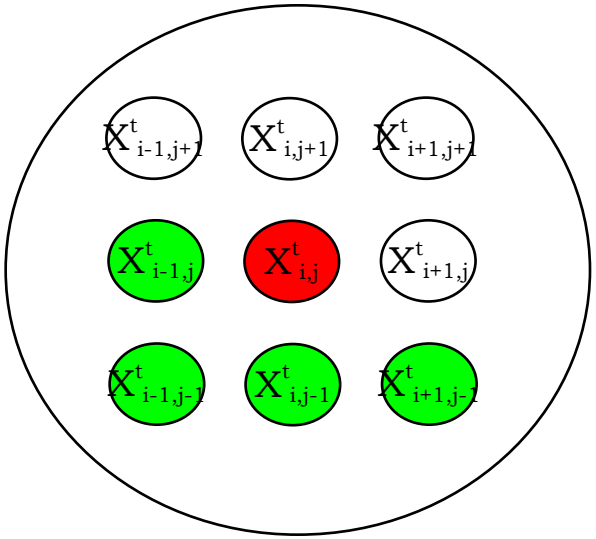




The nodes marked in green have already reached step t+1.

The other nodes are still at step t.





Can we use the *green values?*

If we do what happens

Do we get the correct answer?

$$\psi^t(i,j)=1/8\left[\psi^t(i,j-1) + \psi^t(i-1,j) +\psi^{t-1}(i+1,j) + \psi^{t-1}(i,j+1)+4*\psi^{t-1}(i,j+1)\right]$$

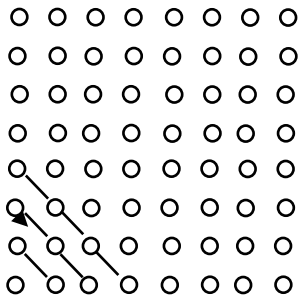
Called Gauss-Seidel

The answer is correct – and it needs **less** steps to arrive at a particular accuracy.

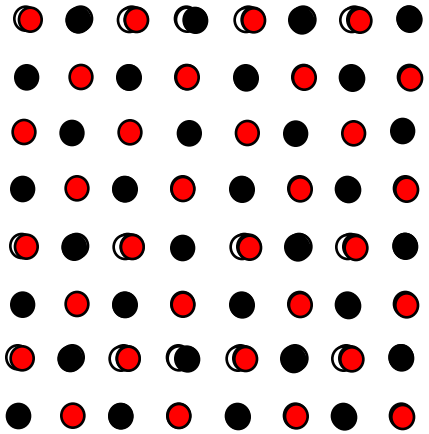
But i,j for a particular step cannot be calculated until values in the diagonal below have been calculated.

At the start very few processors can be used.

Second wave can start before first wave completes



Parallel Design



No red node depends in value on any other red node
No black node depends in value on any other black node.

Update all red, then update all black. Like a leapfrog update.

Need to look at – efficiency of speed-up.

Communication overhead.

Efficiency of algorithm.

If A speeds up less than B, but B needs more step to produce the answer. The correct algorithm is not trivial.

Trivial extrapolation to higher dimensions

Looking for answer not efficiency

What if communication is asynchronous?

Producer doesn't know when result is required.

FEA result is always required as produced.

Large data structure – all of which can be written by many tasks.

Data structure I/O can be structured as a single task. Bottleneck

Structure distributed among computational tasks.

> Requirement to check for I/O requests complicates code

Separate set of communication tasks. No data locality

25 Communication check

Checklist for communication.

All tasks perform the same number of communication operations. If unbalanced worry – can we duplicate data to reduce number of operations.

Are communication partners local or global. Local is best.

Possibility of concurrent communications?

Can the computation in different tasks proceed concurrently. Consider order of communication and computation tasks.

Agglomeration

Stages 1 and 2 produce an abstract solution.

Stages 3 and 4 look at the particular architecture.

How do we put the bits together to create an efficient and scaleable solution?

Do we need to duplicate data?

Do we aim at one task per processor?

Alternately leave to mapping

Consider *granularity, flexibility, engineering costs*

Balance sometimes conflicting requirements.

Granularity

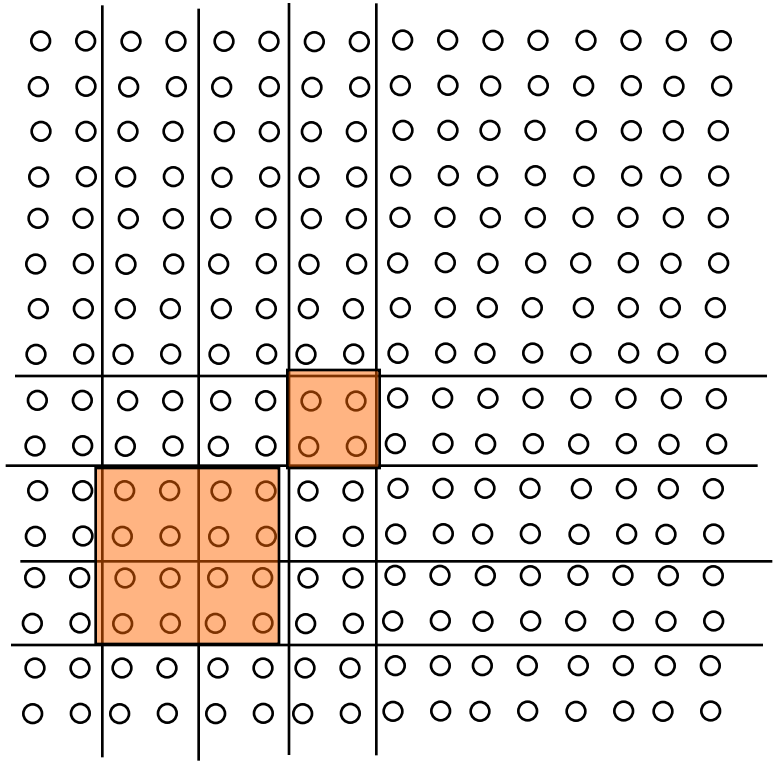
Communications can form a big part of parallel computation.

Reducing granularity may reduce message volumes beneficially.

It will reduce total task creation time.

May be constant message volume but reduced message number is desirable.

When message start up is a significant overhead



FEA again.

1 node communicates with 4 external nodes.
4 per node

2*2 node communicates with 8 external nodes.
2 per node

4*4 node communicates with 16 external nodes

1 per node.

Increase number of tasks per processor reduce amount of communication per calculation

Only true for local communications

Agglomeration on the Grid

Traditional parallel asks about multiple tasks on one machine versus spreading through a cluster.

Parallel v. communication.

Cloud is normally in one data centre - may be communication problems – is bandwidth in a rack as good as communication between racks – can you specify location?

What if we want to run on several sites.

Communication overheads are large between sites.

Problem of ensuring co-ordination between sites.

Booking bandwidth between sites.

Remember looking for optimum solution. Not the most scalable algorithm. (Normally)

Allocating tasks to processors

Common sense.

Place tasks which need to communicate close.

Separate tasks which can run concurrently

Can you specify with Cloud provider?

Mapping problem is NP complete.

In effect it is unsoluble

Of then the load per task is stochastic and some sort of balancing during execution is required.

Not enough to send 100 tasks to 100 processors, one may end up taking 10 time as long as the others.

Load balancing should not use more time than it saves!

Solution then arrives much later.

In most complex cases

number of tasks changes dynamically

size of tasks appears dynamically.

Multiple operations – local algorithms cause less overhead.

Recursive bisection

Divide the problem into subdomains

Aims

Equal computational cost

Minimum communication overhead.

Send part of problem out and sub-divide later.

Repeat – the algorithm can itself be executed in parallel.

Bisection may not give best results in complex geometries. *Unbalanced recursive bisection.* Break into P parts. Same aim.

Balance computational cost, minimise communications.

Other more complex "*recursive spectral bisection"*

Pothen et al SIAM J. Mat. Anal. Appl. 11(3) 1990

Reviews

H. Simon. Computing Systems in Engineering 2(2/3) 135-148 1991

R. Williams. Concurrency: Practice and Experience 3(5)457-481 1991

Local algorithms

For instance compare load with other local processors

Transfer load to more lightly loaded processors.

Multi-level – distribute loads at high level to local "controllers". Controllers communicate with "local" nodes to balance load.

(For some problems "Controllers" can "request" more load after start.)

Reduces communications overheads.

Random

Allocate tasks at random.

Many tasks and expect the load to be equal (probably) computationally very light.

Scheduling

If we have only weak locality constraints.

Chop the task up into a pool of tasks. Free processors are allocated tasks.

Works on the grid for particle physics.

Works on cloud for similar problems

Easy for suitable problems

Manager - worker. Very simple to implement.

Managers simpler than controllers.

LHC use “pilot” jobs – investigate the performance of nodes and adjust submission appropriately.

A single manager is susceptible to failure.

“Elections”

Hierarchical manager-worker. Looks like distributed controllers. Controllers can re-allocate after start. Managers allocate to free.

Split the task into tasks with guaranteed “longest time” – manager works.

How does a processor know when it is finished?

Central co-ordination is easy – ask the boss.

Completely independent tasks simple – finished the allocated task and that is your job done.

Decentralised – much harder …

Particle Physics problems – are completely independent and so finish and exit works.

Some of the LHC experiments have found the overheads in sitting in the queue unacceptable.

Run a task – which contacts central server and asks for events for this node.

Blocks a slot in a processor farm – causes problems.

- Local management lose control of their resources.
- Tasks creating other tasks have security implications

Have you considered both a static solution when all tasks are created and then run

and

a solution which includes dynamic task creation and deletion

If you have centralised load balancing schemes have you ensured the central point is not a bottleneck.

For dynamic load balancing have you considered several different strategies.

If using probabilistic or cyclic methods do you have enough tasks to distribute. At least an order of magnitude more tasks than processors

Designing for Failure

Distributed Computing

"Where your can fail due to the action(s) of a computer of which you have no knowledge."

Failures are inherent in solving large problems on Cloud

For instance paper on RGMA testing

And is …

Infrastructure needs to be resilient – and is designed as such.

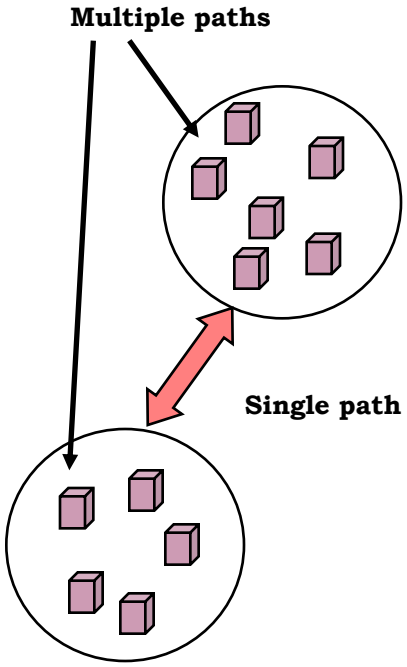
Need to confirm cloud infrastructure of the provider will is resilient. Connections to the cloud are as resilient as the connections to the internet.

Need to ensure workflow system can cope with failures

The infrastructure does not protect individual computing tasks.

Failures can be classified according to their effect.

> Understand the morphology and protect your jobs.
>
> Protection may involve manual intervention.



Crash

processor halts. Does nothing else *in particular no illegal messages.* Never restarts.

Crashes are not detectable in asynchronous systems *crashes are indistinguishable from slow connections.*

Link failure

A communication link fails. It stays failed. Link failure may reduce communication bandwidth **OR** it may partition the network, some pairs of nodes can never communicate

NETWORK PARTITONED

Omission

Only a proper subset of the required messages is *sent* or *received.*

Byzantine fault

System fails with arbitrary behaviour.

A system which can tolerated Byzantine behaviour can tolerate any faults.

THAT is no country for old men. The young
In one another's arms, birds in the trees
- Those dying generations - at their song,
The salmon-falls, the mackerel-crowded seas,
Fish, flesh, or fowl, commend all summer long
Whatever is begotten, born, and dies.
Caught in that sensual music all neglect
Monuments of unageing intellect.

Link failures

Some analyses assume that there are no link failures on the basis that they can always be modelled by other forms of failures.

Useful in GC especially if we are looking at failure probabilities.

Heterogeneous processor/network – failure probabilities can be shuffled between sources.

NETWORK PARTITONED

GC with multiple CPU and link types the allocation can be made but would be rather artificial.

Two identical CPUs – different sites so different failure probabilities.

Byzantine fault

Hardest to protect against. Various theorems.

N processors with t failures need to run correctly.
$t < N/2$ for benign failures

$t < N/3$ for malign failures

Synchronous v Asynchronous

What is the difference?

Asynchronous means that a process can ask for some communication and then go away and do something else.

Synchronous means that having asked the process waits.

Implication – synchronous system there is a maximum wait time at which point the end-point is assumed to have failed in some way.

Asynchronous – there is no maximum time.

> A slow asynchronous system is indistinguishable from a system where one part has died.

Computer theory tells you reasoning about asynchronous systems is much harder.

Making a reliable system is likely to be easier if you insist on synchronicity.