Chapter 2: Designing Parallel Algorithms

Got ourselves some

- ✓ parallel machine abstraction
- ✓ parallel programming abstractions

Now need to translate a specification of a problem into an algorithm that displays concurrency, scalability and locality (will do modularity later).

- Requires creativity! ("art")
- There are no simple recipes
- Methodological design can help:
  - ✓ Examine all option, evaluate all alternatives
  - ✓ Reduce backtracking costs

Provide ***FRAMEWORK*** for doing this here. Try and develop:

- ✓Intuition
- ✓experience
- ✓eye for design flaws that compromise efficiency/scalability

Design is basically a highly nonlinear process!

We will try and create some basic principles and some design checklists

2.1 Methodological Design

For our programming problems:

- ✓Solution not generally unique
- ✓Existing sequential solutions may be very misleading

Strategy:

- ✓Consider machine independent issues (e.g. concurrency) ***EARLY***
- ✓Consider machine dependent issues ***LATER***

Four stages (four main weapons of design!): P C A M

- ✓ Partitioning
- ✓ Communication
- ✓ Agglomeration
- ✓ Mapping

2.1 Methodological Design

Partitioning (domain/functional):

Task creation. Decompose computation and data into small tasks. Concentrate on recognising parallel opportunities. Ignore no. of processors and other practical issues.

Communication (local/global, static/dynamic, structured/unstructured, synchronous/asynchronous):

Channel creation. Set up communication structure required to co-ordinate task execution.

Agglomeration (reduce communication and therefore costs):

Evaluate task-channel communication structures with regard to performance and implementation costs. Combine into more efficient (larger?) tasks.

Mapping (load balancing, task scheduling, static or run-time):

Assign tasks to processors. Satisfy competing goals of maximal processor utilisation but minimal communication.

P & C : concentrate on concurrency and scalability

A & M : concentrate on performance, especially due to locality

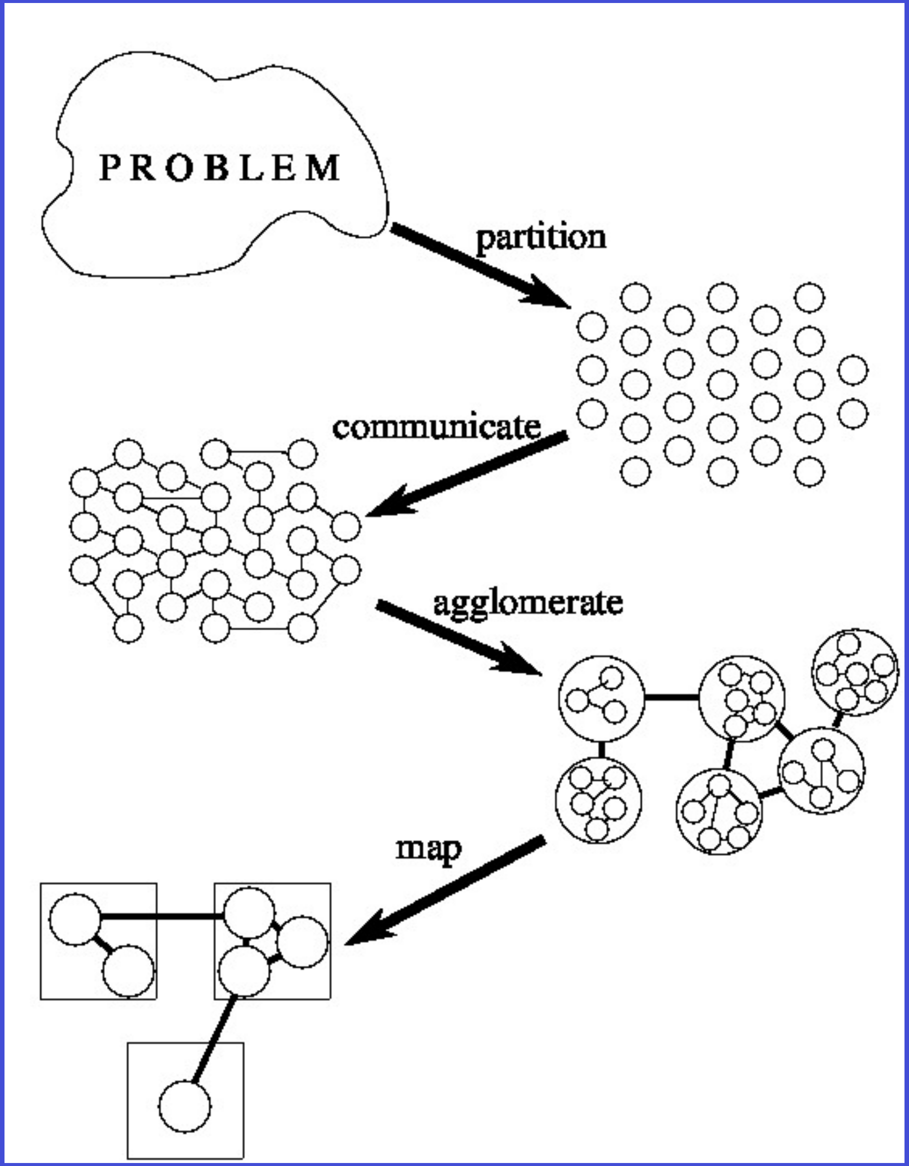
Result can be:

✓Program that creates and destroys tasks dynamically

✓MORE LIKELY FOR US: SPMD - static structure that assigns one (agglomerated) task per processor for all time (=> agglomeration and mapping are basically the same thing)

2.1 Methodological Design

PCAM



2.2 Partitioning

P C A M

- As mentioned earlier when describing task-channel model, at this stage, aim for ***FINE GRAIN*** decomposition of the problem:

  "Fine grained sand is easy to pour ; Bricks are not so easy"

  $\Rightarrow$ fine-grained is more ***FLEXIBLE***

  Agglomerate *LATER* to change granularity

  Take machine/practical details into account at ***THAT*** stage, not here

- Want to divide problem into pieces of computation AND the pieces of data that the computation operates on = tasks

  Obviously want ***DISJOINT*** (non-overlapping) sets of computation/data if possible

- Two ways of doing this:

  ✓ Domain decomposition

  ✓ Functional decomposition

2.2.1 Domain decomposition

P C A M

Seek to decompose the ***DATA*** of the problem by ***GEOMETRY***

We would like ***small pieces of equal size*** => fine-grained and load-balanced

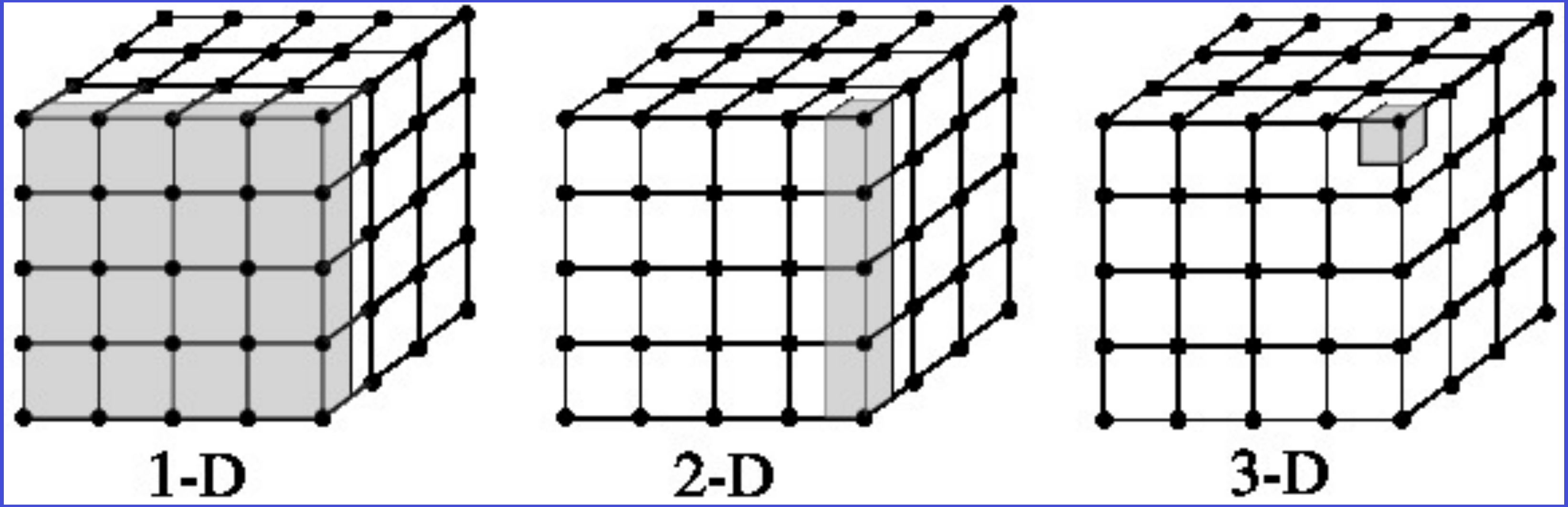
Then associate computations with that data => tasks

There may be different geometrical decompositions at different phases of the program (e.g. input, intermediate computation, some other computation, output etc)

Rule of thumb: start with the largest data structure or the one that is going to be accessed most frequently

2.2.1 Domain decomposition

Domain decomposition example: (The one you all know and love -- Cartesian grids)



1-D data decomposition

1 dimension cropped

2 dimensions full

Task = plane

2-D data decomposition

2 dimensions cropped

1 dimension full

Task = column/pencil

3-D data decomposition

3 dimensions cropped

0 dimension full

Task = grid point (or few)

Aggressive!

2.2.2 Functional decomposition

Seek to decompose the *COMPUTATION* of the problem into *disjoint tasks*

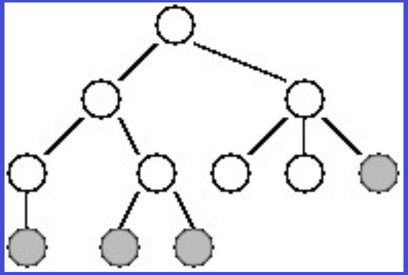
Then add data:

- If data also disjoint, good!
- If not, and LOTS of communication is required, ABORT and try domain decomposition!

e.g. 1: Earlier "tree search" problem:

No obvious data decomposition

New searches (=computation) are obvious tasks



e.g. 2: Complex system modelling e.g. climate model

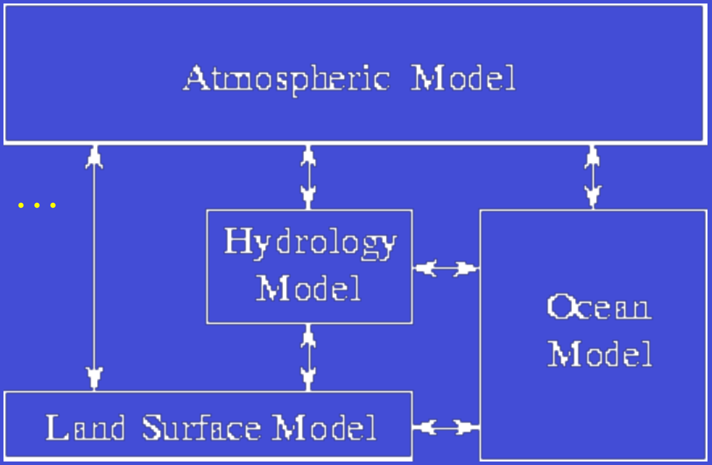
Divide into tasks by *physics*

Sections for atmosphere, ocean, ice, carbon dioxide, …

Each "model" = separate task

Works if little communication between tasks

Tasks can be further decomposed (by domain!)



2.2.3 Partitioning design checklist

P C A M

✓ Does the partition define at least an order of magnitude more tasks than there are processors on the target machine? If not, you may not have flexibility in later design stages.

✓ Does the partition avoid redundant computation (and storage)? If not, algorithm may not scale well to larger problems.

✓ Are tasks of comparable size? If not, may be load balancing issues -- hard to allocate equal work to processors.

✓ Does the number of tasks scale with the problem size? Increase in problem size should ideally also increase the number of tasks rather than increase the size of individual tasks. If not, may not be able to solve really big problems even when more processors are available.

✓ Have you at least identified the alternatives (domain vs functional)? May help design issues down the road …

✓ Have you looked for some changes to the original problem that might make life a lot easier in parallel? E.g. Switch numerical methods from spectral to finite-differences! Sometimes need to examine pre-history, because things were designed for sequential programming and there are different trade-offs. Due to parallel issues, may be better to start again from scratch!

2.3 Communication

P C A M

So we have ***TASKS***

- ✓ Tasks should be able to run concurrently
- ✓ BUT tasks seldom completely independent

⇒ Data must be transferred between tasks for computation to proceed

Information flow => communication => ***CHANNELS***

Two conceptual stages:

1. Define channels between tasks that need to communicate
2. Specify messages to be sent/received on those channels

(May not actually do anything for 1. - just a concept!)

We need to consider:

- ✓ Sending messages incurs physical costs => we want to avoid unnecessary communication
- ✓ We want to distribute communication over tasks – don't do all in one task - load balancing
- ✓ We need to organise to allow concurrency

2.3 Communication

Domain decomposed problem:

- ✓ Even simple partitions can have complex communication structures
- ✓ Tasks (data in a partition) may have many operations NOT needing communication and some sprinkled around that do
- ✓ => hard to agglomerate communication due to it's irregularity.

Functional decomposition problem:

- ✓ Often straightforward
- ✓ Data flow is between *coarse-grained* tasks e.g. wind data from atmospheric model is used to drive ocean model.
- ✓ => Easier to agglomerate

2.3 Communication

Four basic types of communication: (our four main weapons of comm …!)

Local/global

Local - communication with a few other neighbouring tasks

Global - communication with lots of other tasks

Structured/unstructured

Structured - regular grid communication, obvious patterns

Unstructured - arbitrary structure

Static/dynamic

Static - communication partners do not change over time (channels fixed)

Dynamic - comm partners determined by data at runtime

Synchronous/asynchronous

Synchronous - senders and receivers co-ordinate

Asynchronous - no co-operation between senders and receivers necessary (just let fly!)

Talk about these in detail …

2.3.1 Local communication

Data required only from a small number of other tasks

Usually simple to define channels and send/receives

Illustrate with an example - Jacobi iteration:

Update of a grid by weighted sum of neighbouring grid points

Which neighbours are used = stencil

Try 5-point stencil:

$$X_{i,j}^{(t+1)} = (X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + 4X_{i,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)})/8$$

Task structure?

Data decomposition

Assign the update of a single x(i,j) for each time t=1,2,3,… to a task

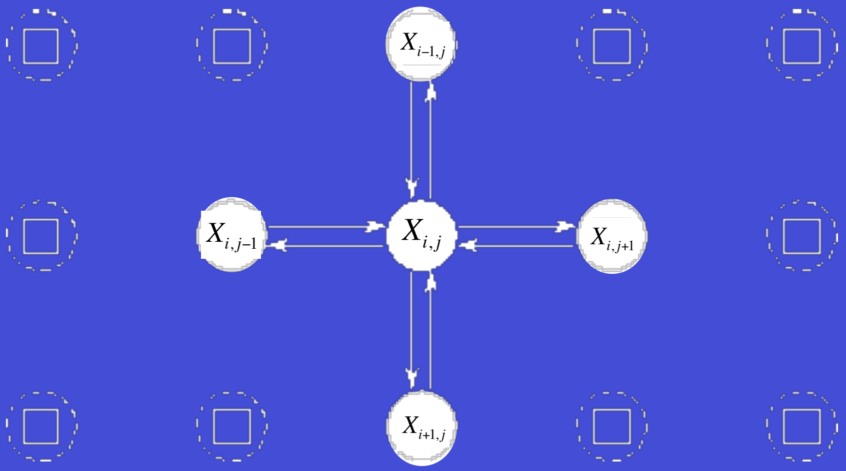
Communication structure?

Each task for an x(i,j) requires knowledge of 4 neighbours:

x(i+1,j), x(i-1,j), x(i,j+1),x(i,j-1)

2.3.1 Local communication (cont)

Envisage channels:



Each task needs to receive a value from it's four neighbours in order to update it's own value

Each task must also send it's own current value to each of it's four neighbours for their computation.

Task logic:

```
For t=0,T-1
send current t x(i,j)   ->   4 neighbours
receive current t from each neighbour    <- x(i+1,j),x(i-1,j),x(i,j+1),x(i,j-1)
compute x(i,j,t+1) from these values
end for
```

2.3.1 Local communication (cont)

This is also a ***great example*** of where

<u>optimal sequential and parallel computational techniques may be different:</u>

In regular computing, would you use Jacobi iteration?

No! There are better strategies, e.g. Gauss-Seidel iteration:

Better in the sense that the iteration converges faster (less iterations)

Minor change:

$$X_{i,j}^{(t+1)} = (\boxed{X_{i-1,j}^{(t)}} + X_{i+1,j}^{(t)} + 4X_{i,j}^{(t)} + \boxed{X_{i,j-1}^{(t)}} + X_{i,j+1}^{(t)})/8$$

If you are updating lexicographically, then x(i-1,j) and x(i,j-1) have already been updated to new information, so should use that (each iteration should be an improvement so use best available).

Programming-wise, it just means update in-place:

```
x(i,j) = x(i-1,j)+x(i+1,j)+4x(i,j)+x(I,j-1)+x(i,j+1)
```

rather than

```
x_new(i,j) = x_old(i-1,j)+x_old(i+1,j)+4x_old(i,j)+x_old(i,j-1)+x_old(i,j+1)
```

In sequential programming, Gauss-Seidel is faster than Jacobi

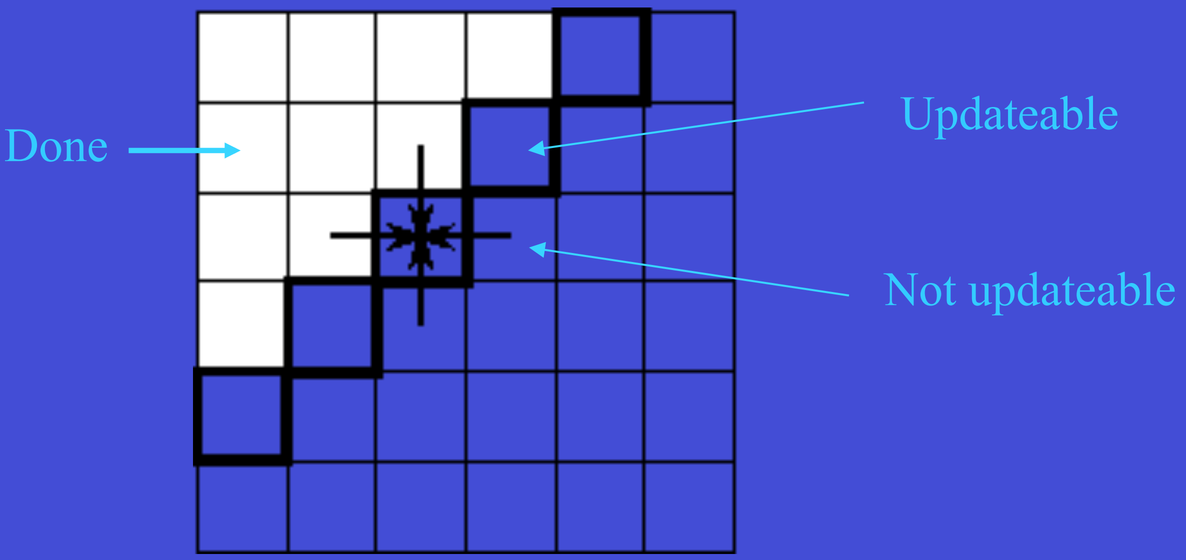
2.3.1 Local communication (cont)

BUT …

Jacobi iteration can have all NxN grid updated simultaneously (since all previous times exist on whole grid before update)

Gauss-Seidel iteration *cannot* update all simultaneously:

Cannot update a point until the points above and to the left are finished



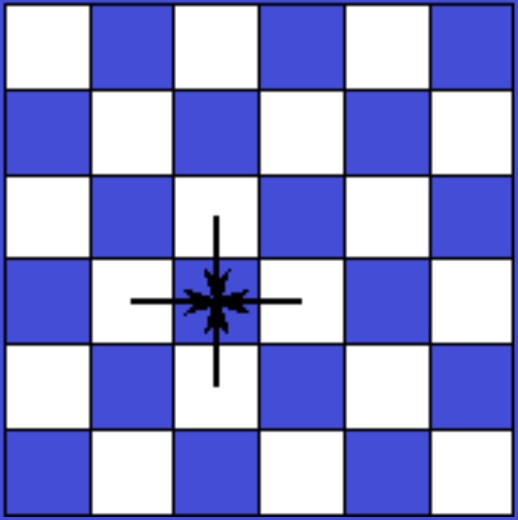
No. of points available for concurrent update starts at 1, goes up to N and back down to 1

⇒ on average can ~ N/2 concurrently. Not so good (compared to N^2 for Jacobi!)

FORTUNATELY … There are other strategies that converge faster than Jacobi and yet are more parallelizable …

2.3.1 Local communication (cont)

e.g. Red-Black (checkerboard) iteration:



Don't think lexicographically -- too sequential!

Update all the odd-numbered elements concurrently

Then update all the even numbered elements concurrently

~ N^2/2 concurrency

- ✓ faster convergence than Jacobi
- ✓ more parallel than Gauss-Seidel
- ✓ best of both worlds!

So, the important point …

Choice of solution strategy plays important role in determining performance of parallel program.

What may be the best choice for sequential program, may not be ideal for the parallel program.

2.3.2 Global communication

P C A M

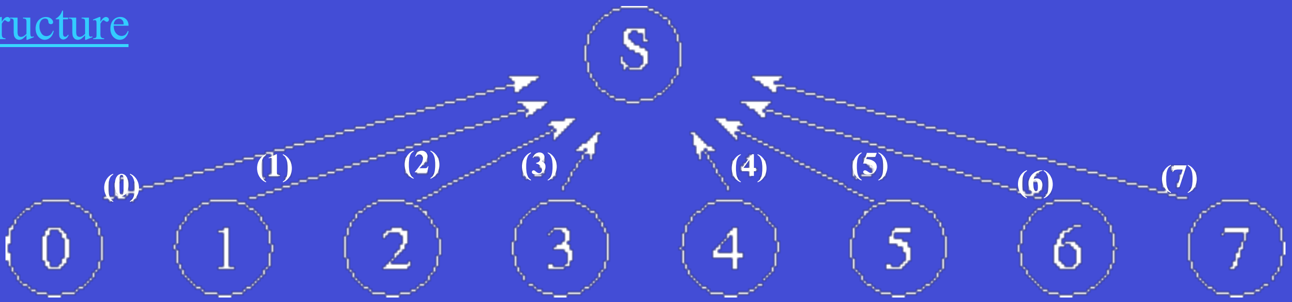
Global communication = a communication in which many tasks must participate

e.g. parallel reduction operation

Reduces N values distributed over N tasks e.g. summation $S = \sum_{i=0}^{N-1} X_i$

Assume a single manager task does the job:

Communication structure



Task logic:

Workers send their numbers to central manager

Central manager sums the numbers

Good algorithm?

No! Since central accumulator can only receive and sum one value at a time, it takes O(N) time to sum N numbers = not very parallel!

2.3.2 Global communication (cont)

P C A M

Problems of this algorithm:

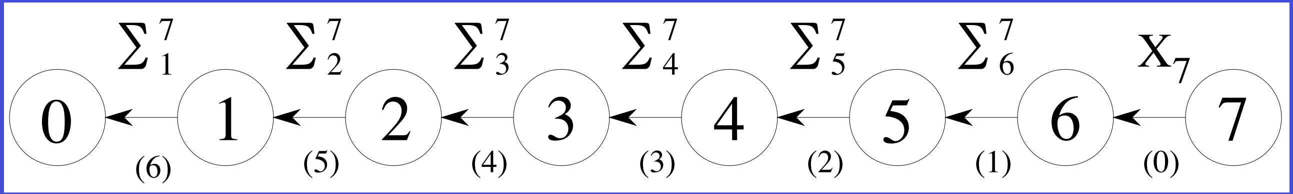
- Algorithm is centralized: it does not distribute computation and communication evenly; the manager task is involved in *every* operation.
- The algorithm is sequential: it does not allow computations and communications to proceed concurrently

Must address these problems to develop good parallel algorithm for this simple function …

Distributing computation and communication:

e.g. make a task-channel structure which is

Task i = sum my x(i) with current sum and send to the left



Now have nicely *distributed computations and communications* (one comp, one comm per task)

BUT STILL NO CONCURRENCY! *Still O(N) steps …*

(but could be good if doing lots of sums: pipeline from right, so eventually pipe full = concurrent comp and comm)

2.3.2 Global communication (cont)

To achieve concurrency … DIVIDE AND CONQUER!

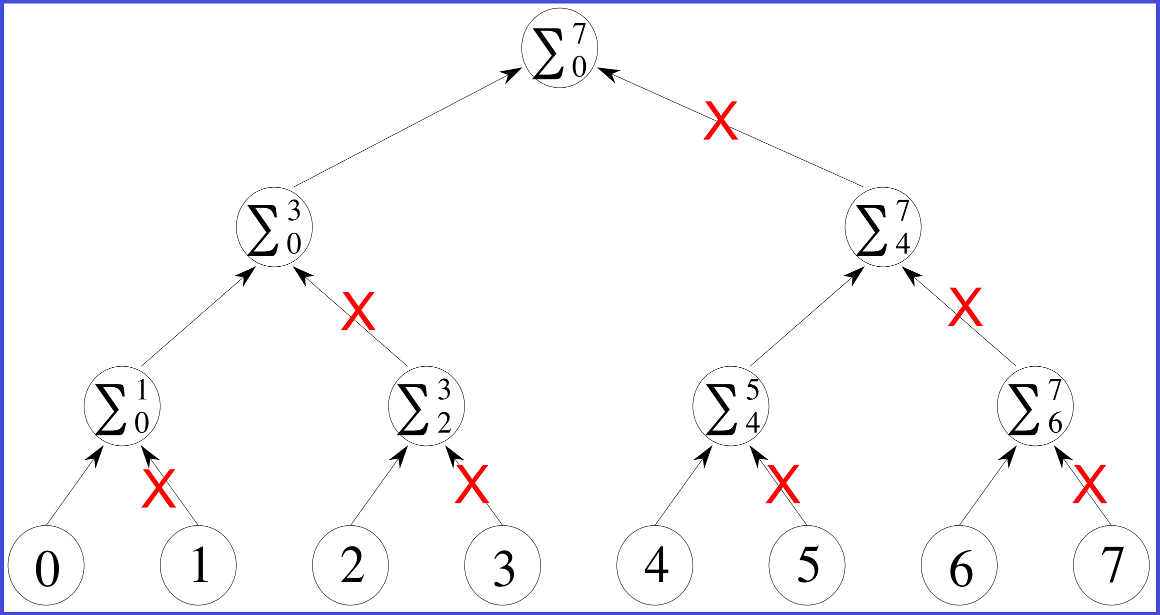
As always, try and divide problem into sub-problems.

In this case, assume N=2^n and divide sum in half

$$\sum_{i=0}^{2^n-1} = \sum_{i=0}^{2^{n-1}-1} + \sum_{i=2^{n-1}}^{2^n-1}$$

These two sums can be performed concurrently.

Extrapolate this …



Number of levels to achieve sum = Height of tree = $\log_2 N$ (= n)

⇒ O(log N) not O(N) steps to complete => concurrency!

Still O(N) communications …

2.3.2 Global communication (cont)

Success! Efficient parallel algorithm!

- ✓ Distributed the N-1 computation and communication operations
- ✓ Modified the order to allow concurrency
- ✓ Regular communication structure (balanced)
- ✓ Each task communicates mostly with a small set of neighbours (locality)

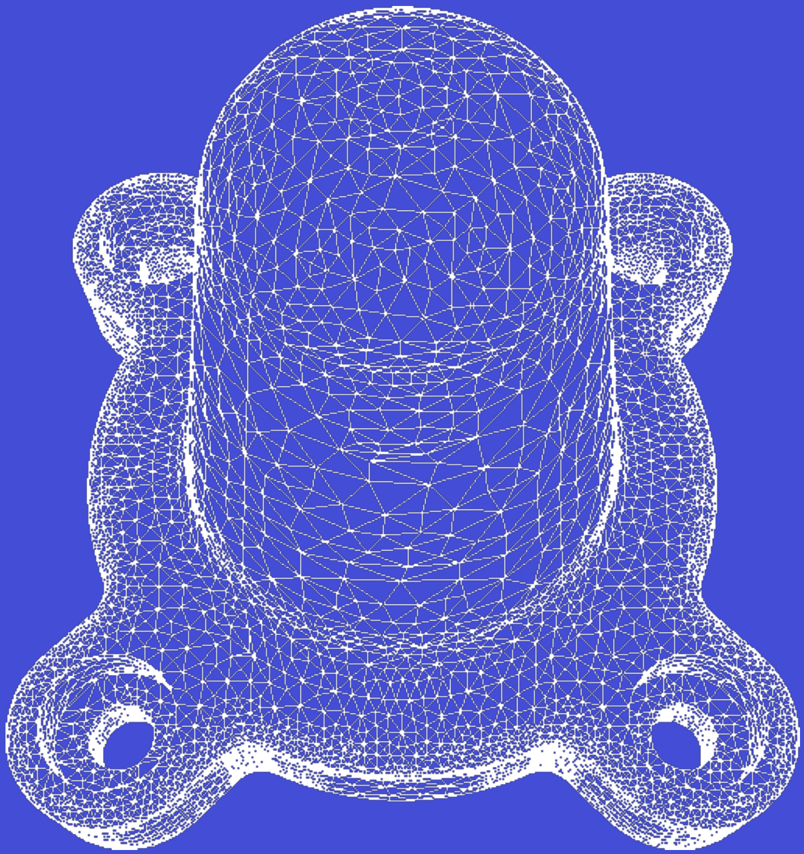
2.3.3 Unstructured and dynamic communication

Previous examples all static and structured => tasks communication structure forms a regular pattern (grid, tree) and does not change over time.

Can be much more complex!

- ✓ grids may take the shape of a complex object or be refined in critical regions
- ✓ Refinement may vary with time (AMR = adaptive mesh refinement)

e.g. finite-element grids in engineering



Does not necessarily prove to be a problem at early stages of design (partition, communication)

e.g. easy to still define task for each vertex in finite-element grid and to require communication at each edge

Will however cause problems at later stages of design - agglomeration and mapping

Need tasks of roughly equal computation and communication and this is difficult with such irregular objects

Agglomeration may need to be highly dynamic => costs

2.3.4 Asynchronous communication

P C A M

Assumed previously, synchronous => all senders and receivers are aware of comm event and senders explicitly send to receivers

Asynchronous => tasks that produce data (producers) are not able to determine when other tasks require data (consumers); consumers must request data from producers.

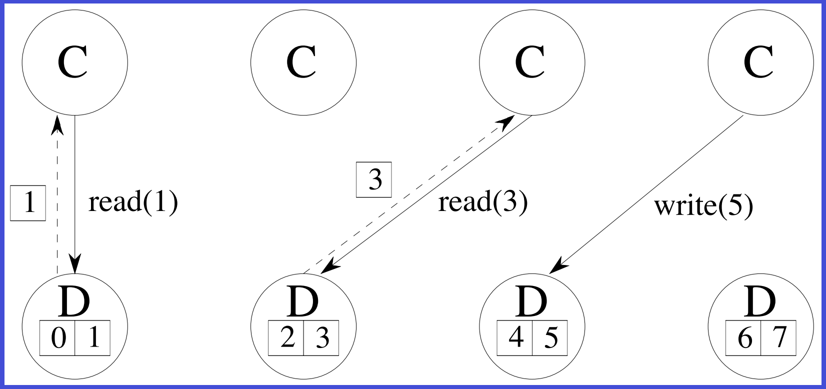
Commonly occurs when tasks periodically read/write a shared data structure.

Assume data structure too large (or too frequently accessed) to be encapsulated in a single task. Require it to be distributed and yet supporting asynchronous read/writes on its components.

Possible ways to do this:

1. Data structure distributed amongst computational tasks. Each task performs its own computation and requests data located in other tasks. Also periodically halts computational task and ***POLLS*** for pending requests on it's own data. (Convoluted programming, polling expensive => trade: expense of frequent vs. rapid response to poll events)

2. Distributed data structure encapsulated in a ***SEPARATE*** set of tasks, responsible only for read/write requests (More modular, loose locality - there is no local data => extra comm)



3. In a shared memory paradigm: no problem! All done automatically. However, must watch for order of operations (synchronisation), cache coherence etc

2.3.5 Communication design checklist

- ✓ Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggests non-scalable. Revisit design to distribute more equitably. e.g. if there exists a frequently-accessed data structure encapsulated in one task, consider distributing it (if large) or replicating it (if small).
- ✓ Does each task only communicate with a small number of neighbours? If each task communicates with many, see if it is possible to reformulate global communication into local (e.g. like "pairwise interactions" exercise or "divide and conquer").
- ✓ Are communication operations able to proceed concurrently? If not, algorithm is likely inefficient and non-scalable. Try "divide and conquer" again to uncover concurrency?
- ✓ Can the computation proceed concurrently or is it blocked by the communication (e.g. Gauss-Seidel)? If not, losing efficiency and scalability again. Can you re-order the computation and communication and "unblock" it? Notice, this might require a complete revision of the algorithm in use (e.g. change Gauss-Seidel to red-black).

2.4 Agglomeration

P C *A* M

We have

- ✓ partitioned computation into a set of tasks
- ✓ introduced communication to provide the required data for the tasks

Implementation is currently ***abstract*** - not specialised for efficient execution on any particular parallel machine.

In fact, ***may be highly inefficient*** if, e.g., there are many more tasks than procoessors if machine is not designed for efficient execution of small tasks

Therefore, we begin a 3rd stage where we move from the abstract towards the more concrete.

We will revisit decisions made about partitioning and communication with a view to obtaining an algorithm that will execute efficiently on some class of parallel machine.

In particular:

- ✓ Consider if it is useful to combine, or *AGGLOMERATE*, tasks identified in the partitioning phase so as to provide a smaller number of tasks each of greater size.
- ✓ Determine whether it is worthwhile to ***replicate*** some data and/or computation.

2.4 Agglomeration (cont)

Agglomeration may *reduce* the number of tasks, but we may still end up with more tasks than processors. In this case, our design remains somewhat abstract still, and the *MAPPING* of tasks to processors is still required.

Alternatively, at this stage we may aim for one task per processor if we are intending to work in SPMD mode in which case the mapping is largely done at the same time.

So here we will concentrate on the *GRANULARITY* issue …

There are *THREE (somewhat) CONFLICTING GOALS*:

- Reduce communication costs by increasing computation and communication granularity
- Retain flexibility with respect to scalability and mapping decisions
- Reduce software engineering costs

We will now discuss these three issues ….

2.4.1 Increasing granularity

In partitioning phase, focused on defining *as many tasks as possible*. Useful because it forces us to consider wide range of possibilities for parallel execution.

However, defining a large number of fine-grained tasks is *not necessarily efficient*.

Generally have to stop computation to communicate. We would rather be computing than communicating! Therefore, try and reduce amount of time spent communicating.

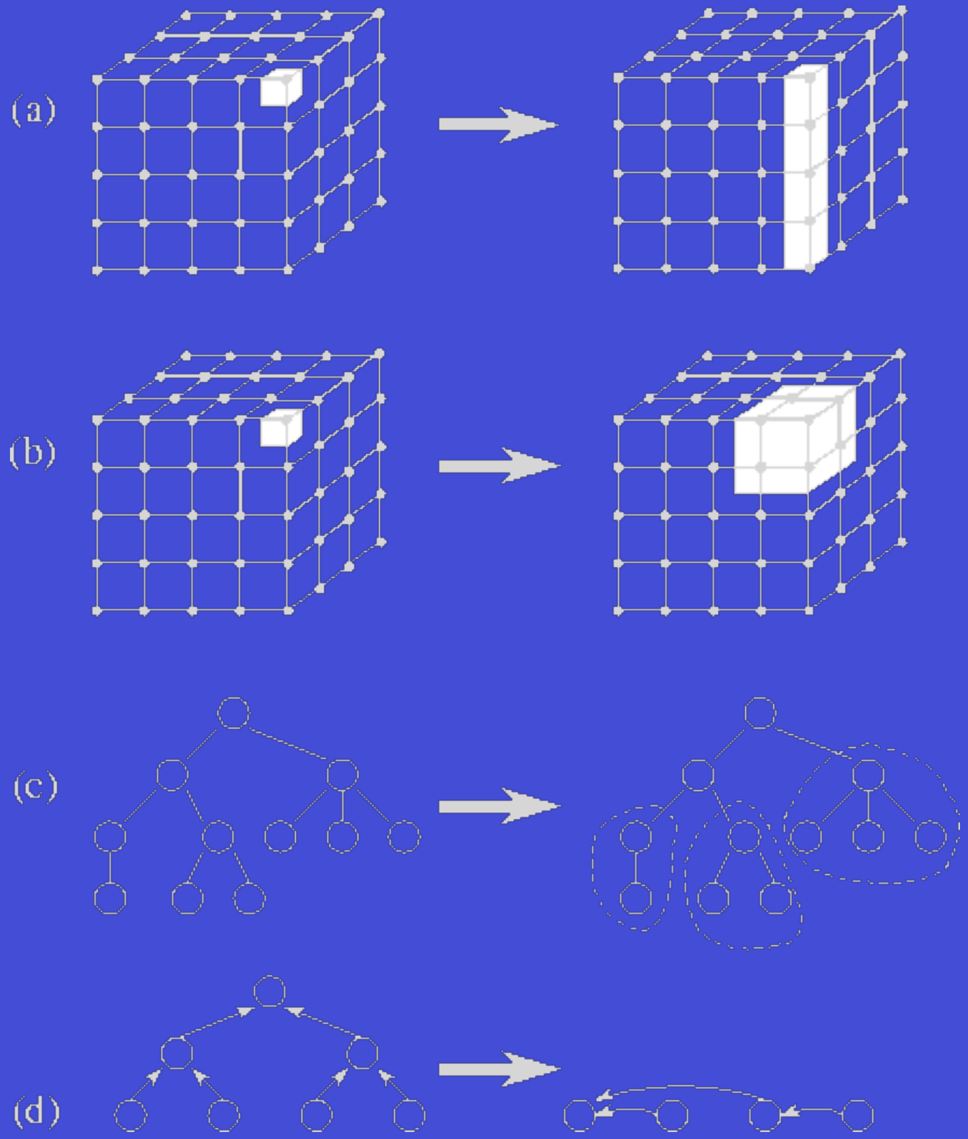
<u>How can we reduce communication time/costs?</u>

- ✓ Send less data: cost is proportional to data length
- ✓ Send same amount of data, less messages: also a fixed startup cost incurred sending messages
- ✓ We may also want to reduce the task creation costs.

If the number of communication partners is small, we can often reduce both the number of communication operations and the total communication data volume by increasing the granularity of our tasks => agglomerating several tasks into one

2.4.1 Increasing granularity (cont)

Examples of agglomerating tasks:



2.4.1 Increasing granularity (cont)

P C A M

Really want to reduce the amount of communication ***PER*** computation i.e. reduce the communication/computation ratio

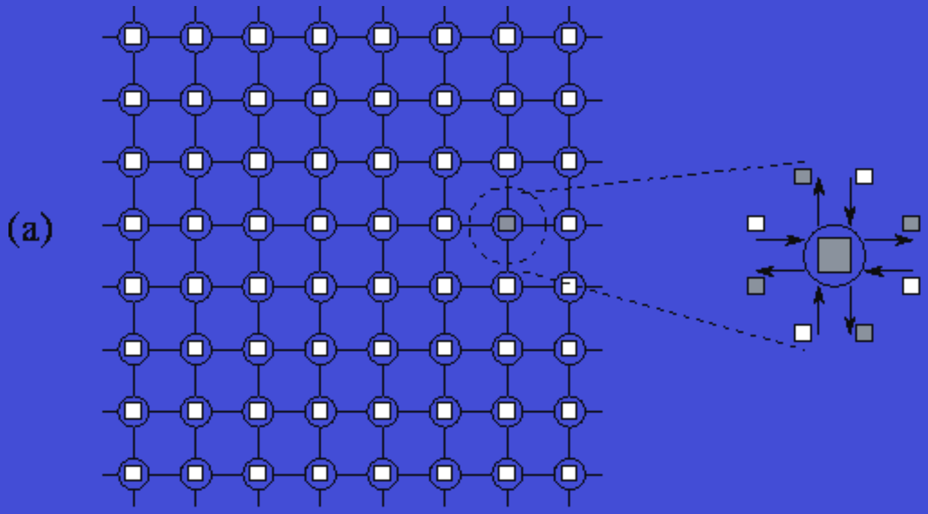
This is a "surface-to-volume" effect: (clear in domain decomposition but true for all)

- Communication costs are proportional to the "surface area" of the sub-domain.
- Computation costs are proportional to the "volume" of the sub-domain.
- => ratio ~ N^2/N^3 = 1/N => decreases with increasing task size
- => agglomerate to increase sub-domain size and reduce ratio

2.4.1 Increasing granularity (cont)

P C A M

Example: 5-point finite-difference/iteration stencil



Costs: Original fine-grained 1x1 tasks

8x8 = 64 tasks

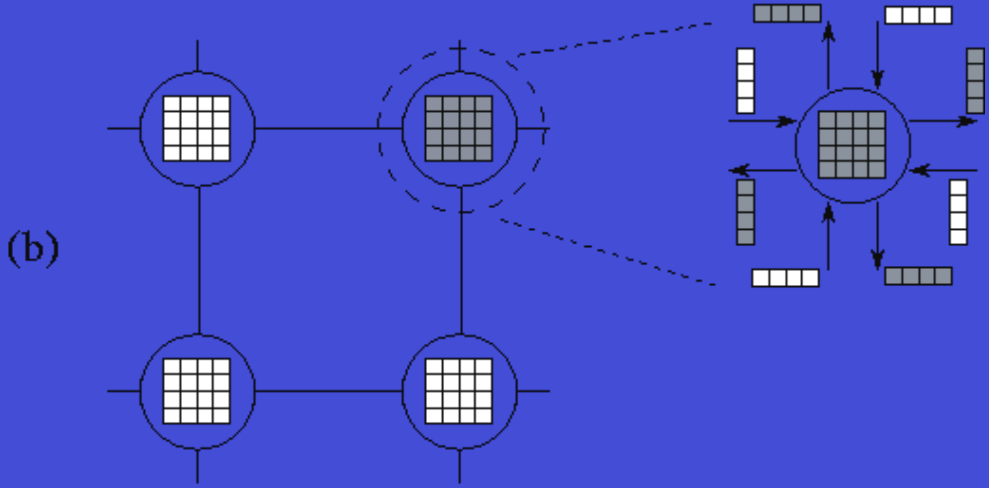
4 communications per task

⇒Total 256 communications

1 datum per comm =>Total 256 data transferred

1 computation per task => Total 64 computations

=> comm/comp = 256/64 = 4/1 = 4



Costs: Agglomerated tasks to 4x4

8x8 = 2x2 array of agglom = 4 tasks

4 communications per task

=> Total 16 communications

4 data per comm => Total 64 data transferred

16 computation per task => Total 64 comps

=> comm/comp = 64/64 = 1

2.4.1 Increasing granularity (cont)

Important consequence: *higher dimensional decompositions are more efficient* since they have lower surface to volume ratios.

⇒ Usually best to agglomerate in ALL directions rather than reducing the dimension of the decomposition

Obviously, this is a much more complicated affair on unstructured algorithms (see later).

2.4.1 Replicating computation

Can sometimes trade-off replicated computation for reduced communication

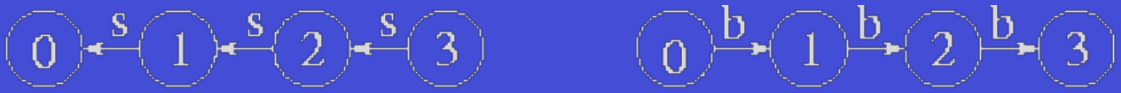
Example: Summation problem (see earlier)

Variation: Say we need the result in every task.

We had devised techniques that used a array or a tree structure.

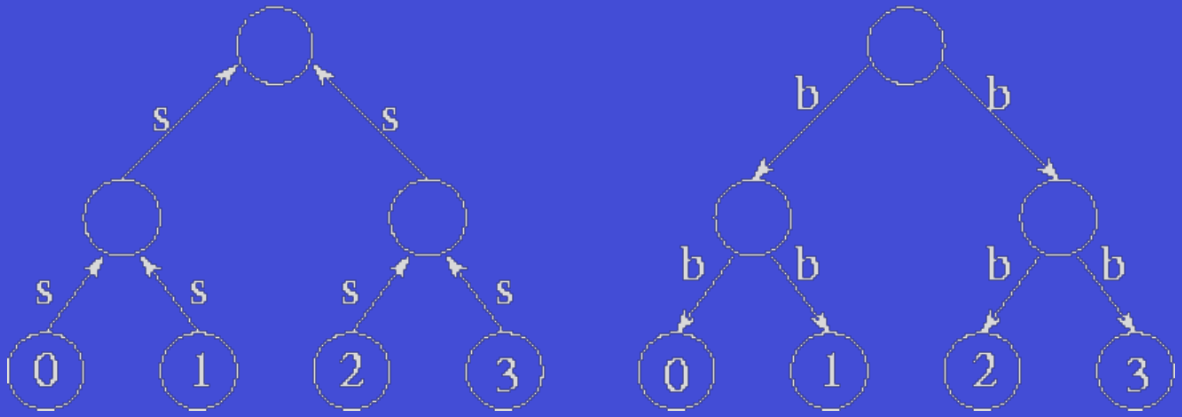
We could achieve our object by broadcast the result back using the same communication structure:

ARRAY



2 * (N-1) steps

TREE



$2 * \log_2(N)$ steps

2.4.1 Replicating computation (cont)

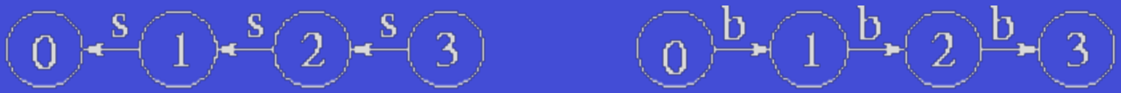
These algorithms are optimal in the sense that no computation is repeated : minimal computations

However, alternative algorithms execute in less elapsed time at the expense of unnecessary computations

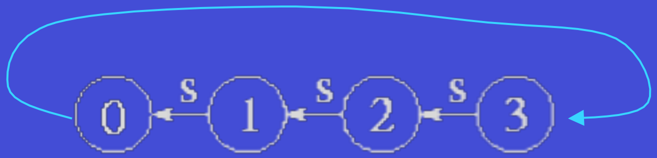
e.g. change array summation to a ring

Perform same algorithm in each processor (add my_number and pass to the left)

ARRAY



RING



In (N-1) steps, all N tasks have computed the full sum! ***HALF THE TIME!***

No broadcast required BUT

Redundant computations (additions): OLD: (N-1) NEW: N*(N-1) => $(N-1)^2$

Redundant communications : (n-2)*(N-1) ~ $(N-1)^2$

2.4.1 Replicating computation (cont)

P C A M

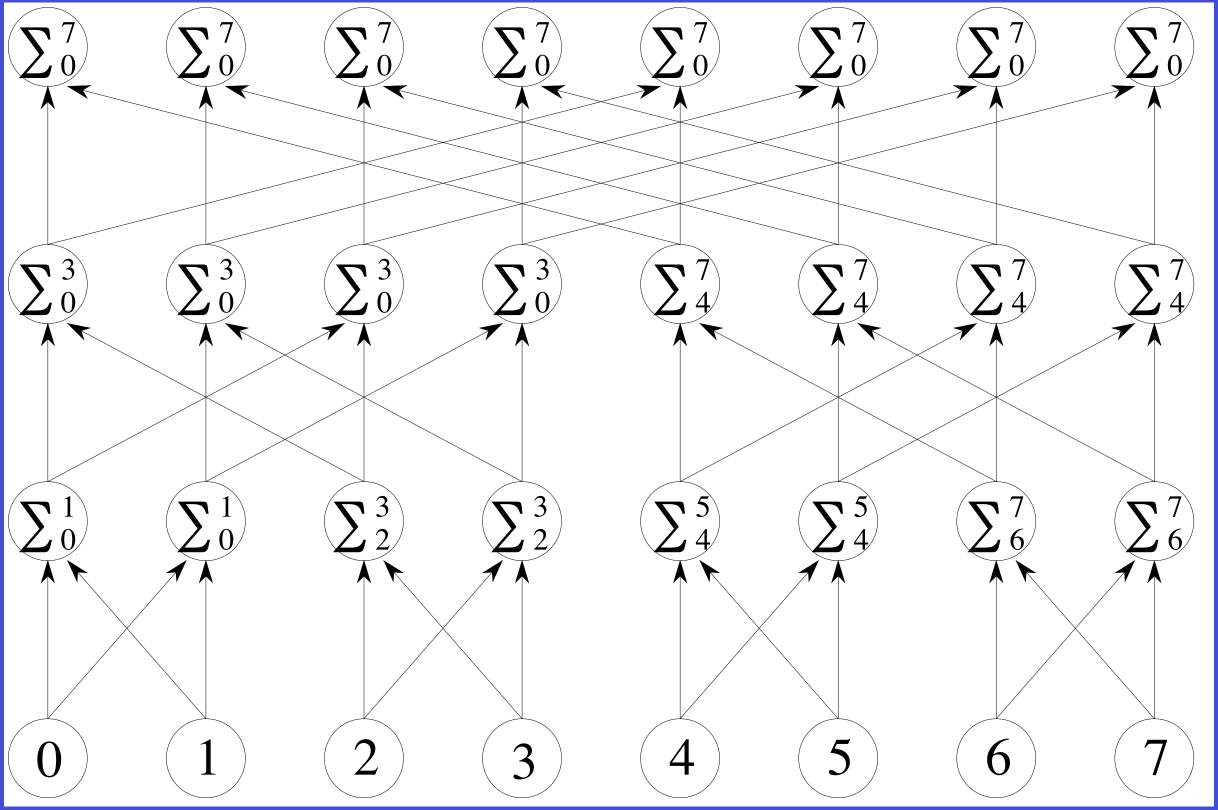
The **TREE ALGORITHM** can be modified in a similar way to avoid the need for a separate broadcast. Multiple tree summations are performed concurrently so that after $\log_2(N)$ steps, each task has a copy of the sum

BUTTERFLY: Each task that computes a sum, keeps the sum ("sends up") and sends to the other half of the pool

Step 3

Step 2

Step 1


$\sum_0^7$ $\sum_0^7$ $\sum_0^7$ $\sum_0^7$ $\sum_0^7$ $\sum_0^7$ $\sum_0^7$ $\sum_0^7$
$\sum_0^3$ $\sum_0^3$ $\sum_0^3$ $\sum_0^3$ $\sum_4^7$ $\sum_4^7$ $\sum_4^7$ $\sum_4^7$
$\sum_0^1$ $\sum_0^1$ $\sum_2^3$ $\sum_2^3$ $\sum_4^5$ $\sum_4^5$ $\sum_6^7$ $\sum_6^7$
0 1 2 3 4 5 6 7

$\log_2(N)$ steps

N comms per step

=>

$O(N \log_2 N)$ comms

$O(N \log_2 N)$ comps

<< $O(N^2)$!!

Less steps, less redundancy than ring

2.4.1 Replicating computation (cont)

P C A M

Agglomeration of tree/butterfly:

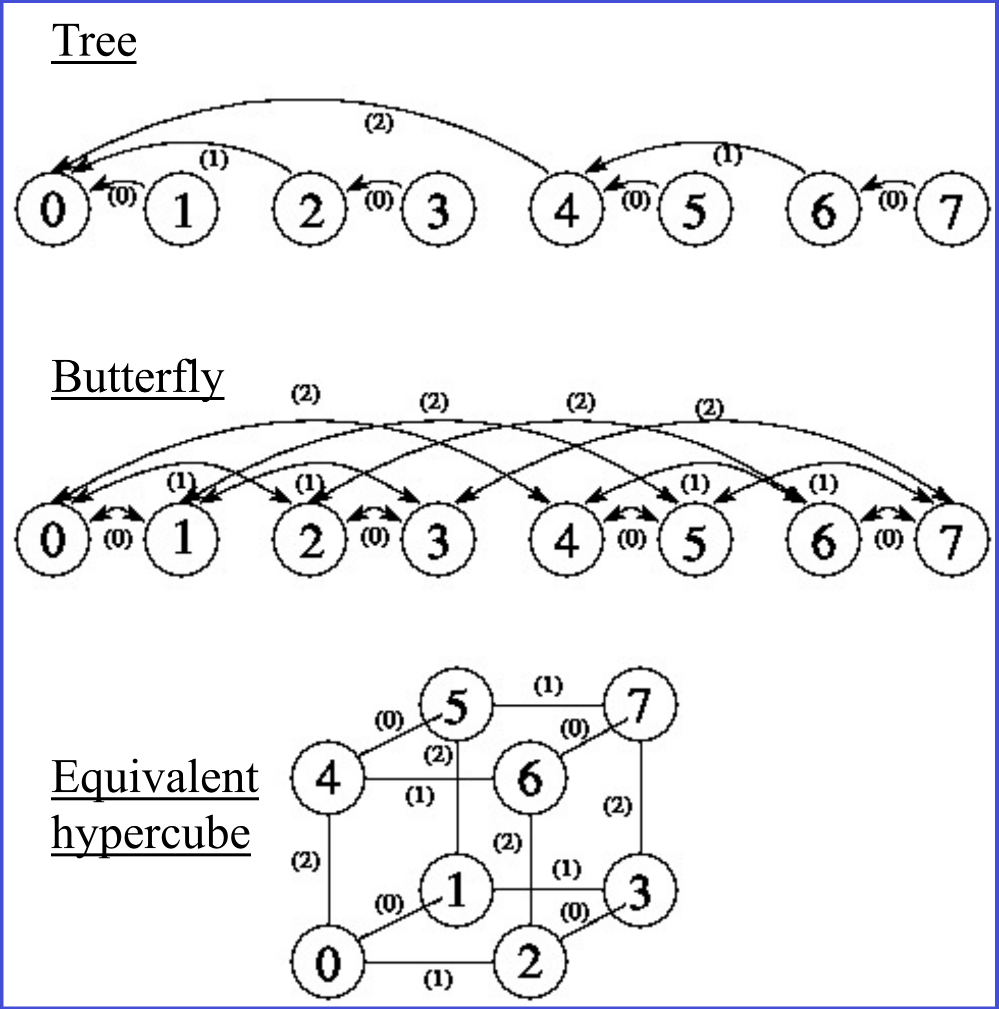
Agglomeration almost always useful if analysis of communication reveals that tasks cannot execute concurrently

Notice that for the tree or butterfly, only tasks at the SAME level can execute concurrently

Hence tasks at DIFFERENT levels can be agglomerated without changing the level of concurrency. i.e. task structure really looks like …

(bracketed numbers = step no.)

(Notice we could pipeline multiple summations too …)



2.4.2 Preserving flexibility

P C A M

When agglomerating, must try not to limit the scalability.

e.g. agglomerating finite-difference grid and lowering the dimensionality of the partition may be bad for a very large number of processors (limited to $\text{Nprocs}_{\text{max}}$ ~ no of planes)

The ability to create a ***varying*** number of tasks is critical if a program is to be portable and scalable.

e.g. if tasks block (wait) for remote data, it would be nice to have some "spare" tasks to map to the same processor so that

- ✓ maybe task would not block (remote data becomes local)
- ✓ processor could compute whilst first task blocked (overlap comp and comm)

More tasks provides more flexibility in the mapping process.

May help in load balancing too

Rule of thumb: order of magnitude more tasks than processors

Optimal number of tasks determined by combination of analytical modelling and empirical studies

Flexibility does not require large no. of tasks. What is required is that the design does not limit the no. of tasks that can be created (=> scalability)

2.4.3 Reducing software engineering costs

So far we have assumed everything is driven by efficiency and flexibility.

Should take into account relative development costs too.

Particularly important when parallelizing an existing sequential code.

Maybe avoid extensive code changes = unnecessary programming! Re-use existing code!

e.g. Finite-difference grid: maybe actually worth partitioning only in 1D if means can use a lot of the existing coding.

e.g. Often codes need different data layouts in different parts of the code. Need to weigh up whether to

- ✓ re-organise the data structure between sections
- ✓ use one data structure throughout even if less efficient.
- ✓ if so, which one?

Each has different performance characteristics that must be examined (more later).

2.4.4 Agglomeration design checklist

P C A M

We have revised partitioning and communication decisions made earlier. We have used agglomeration to *increase concurrency, increase granularity to reduce cost, decrease software engineering costs.*

- ✓ Has agglomeration reduced communication costs by increasing locality? If not, re-examine agglomeration strategy to see if this is possible.
- ✓ If agglomeration led to replicated communication, have you verified that the benefits outweigh the costs of the redundancy for a range of problem sizes and processor counts?
- ✓ If agglomeration replicates data, have you verified that this does not impact scalability? (i.e. does it restrict the range of problem sizes or processor counts that can be used?)
- ✓ Has agglomeration yielded tasks with similar computation and communication costs? (The larger the tasks, the more important this is. One task per processor => need nearly identical costs)
- ✓ Does the no. of tasks scale with the problem size? If not, cannot solve larger problems on a larger machine
- ✓ If agglomeration removed some concurrency, is there sufficient concurrency left for current and future target machines? (An algorithm with insufficient concurrency may still be the most efficient, if other algorithms have excessive communication costs)
- ✓ Can the no. of tasks be reduced further without introducing load imbalances, software engineering costs, reduced scalability etc? Fewer large-grained tasks often simpler and more efficient (due to surface-to-volume effect)
- ✓ Can you re-use existing code by choosing an appropriate agglomeration strategy? There is a cost trade-off: software engineering vs. efficiency

(Recap from Tues)

(Recap from Tues)

Partitioning (domain/functional):

Task creation. Decompose computation and data into small tasks. Concentrate on recognising parallel opportunities. Ignore no. of processors and other practical issues.

Communication (local/global, static/dynamic, structured/unstructured, synchronous/asynchronous):

Channel creation. Set up communication structure required to co-ordinate task execution.

Local communnication – may need to redesign serial algorithm!

Global communicaton – divide-and-conquer => tree/Butterfly

Agglomeration (reduce communication and therefore costs):

Evaluate task-channel communication structures with regard to performance and implementation costs. Combine into more efficient (larger?) tasks.

Mapping (load balancing, task scheduling, static or run-time):

Assign tasks to processors. Satisfy competing goals of maximal processor utilisation but minimal communication.

P & C : concentrate on concurrency and scalability

A & M : concentrate on performance, especially due to locality

SPMD - static structure that assigns one (agglomerated) task per processor for all time (=> agglomeration and mapping are basically the same thing)

2.5 Mapping

P C A M

Finally we need to *specify where each task is going to execute =* ***MAPPING***

Not a problem on uniprocessors and shared memory machines (automated task scheduling)

No general purpose mapping mechanism for scalable parallel computers!

Goal: minimise total execution time

Two basic strategies:

- ✓ Place tasks that can execute concurrently on different processors, enhancing concurrency
- ✓ Place tasks that communicate frequently on the same processors, increasing locality

Clearly, sometimes these two strategies will conflict, requiring some trade-offs

Also, resources may restrict the number of tasks that can be placed on one processor.

No tractible way of evaluating the general case. Therefore present rough classification of problems and some representative techniques …

2.5 Mapping (cont)

P C A M

1. Many algorithms developed using domain decomposition feature

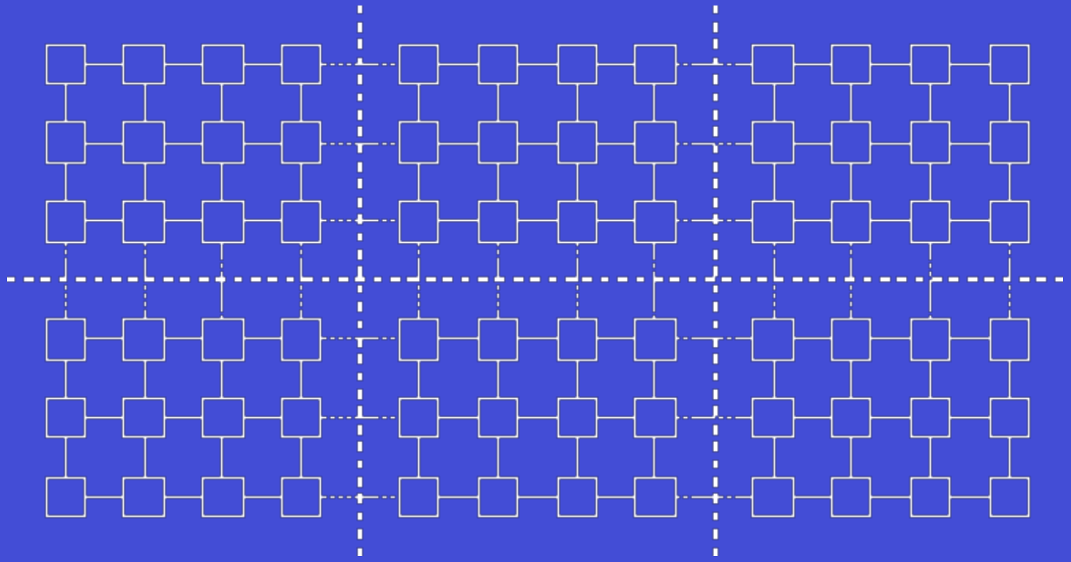
   ✓ Fixed number of equal sized tasks

   ✓ Structured local and global communication

⇒ Mapping pretty easy:

   ✓ Map to minimise communication

   ✓ Agglomerate tasks mapped to same processor (if not already done) to yield P coarse-grained tasks, one per processor.



Tasks have regular comp and comm

Grid and assoc comp is divided s.t.

✓ Same amount of comp per proc

✓ Minimise comm out of proc

(fill proc => min comm)

2.5 Mapping (cont)

P C A M

2. In more complex domain decomposition algorithms featuring
   - ✓ Variable amount of work per task
   - ✓ Unstructured local/global communication

   Use load-balancing techniques to identify agglomeration and mapping strategies.
   - ✓ Must evaluate trade-off: time required to execute load-balancing algorithm vs. decrease in total execution time.
   - ✓ Probabalistic load-balancing often has less overhead (i.e. random assignements)
3. The most complex tasks have
   - ✓ No. of tasks changes dynamically, and/or
   - ✓ Dynamics changes of amount of comp or comm per task

   Use dynamic load-balancing
   - ✓ Load-balancing performed periodically
   - ✓ Local load-balancing may be better (don't need to know global comp state)

2.5 Mapping (cont)

4. Algorithms based on functional decomposition
   - ✓ Many short-lived tasks
   - ✓ Co-ordinate with other tasks only at beginning and end of execution

   Use task-scheduling algorithms
   - ✓ Allocate tasks to processors that are idle or soon to become idle

Look at all these load-balancing techniques briefly …

2.5.1 Load balancing algorithms

Wide variety of load-balancing algorithms for domain decomposition techniques.

We review a few:

- ✓ Recursive bisection
- ✓ Local algorithms
- ✓ Probabilistic methods
- ✓ Cyclic mapping

Need to agglomerate tasks to one per processor *OR* can think of it as partitioning domain into one sub-domain per processor (hence often known as partitioning algorithms)

Recursive bisection

e.g. complex finite-element grid.

Used to partition a domain into sub-domains of approx equal computation cost while minimising communcation costs (i.e. no. of channels crossing domain boundaries)

Basic idea: Cut domain into two sub-domains. Then cut each sub-domain recursively until have the required number. A divide-and-conquer approach => algorithm itself can be performed in parallel!

2.5.1 Load balancing algorithms (cont)

P C A M

Most straightforward: Recursive co-ordinate bisection

e.g. irregular grids that have mostly local communication structure

- ✓ Makes cuts along physical co-ordinates of domain.
- ✓ Subdivide longest co-ordinate direction (moves towards better comm/comp balance?)

Advantages and disadvantages:

- ✓ Simple
- ✓ Inexpensive
- ✓ Partitions computation well
- × Does NOT optimise communication
- × Can end up with long skinny subdomains => can generate more messages than square subdomains (surface-to-volume)

2.5.1 Load balancing algorithms (cont)

P C A M

Improvement: Unbalanced recursive co-ordinate bisection

Attempts to reduce communication costs by forming subgrids with better aspect ratios

Instead of dividing in HALF, considers the P-1 partitions obtained by forming unbalanced subgrids with

1/P and (P-1)/P of the load

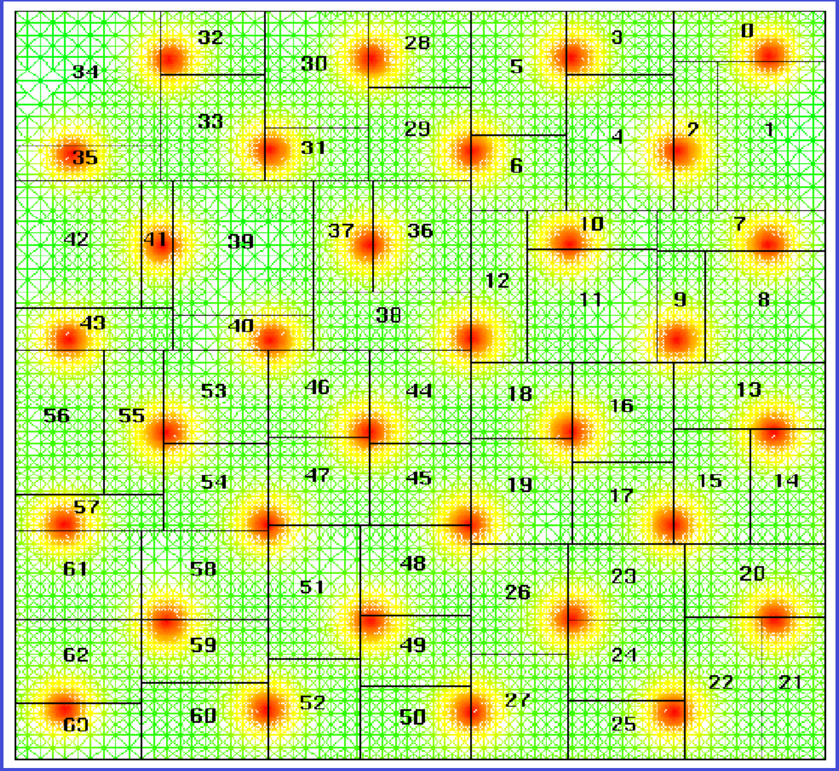
2/P and (P-2)/P of the load

… etc

and chooses the partition that minimizes the partition aspect ratio

Increases cost of computing the partition

BUT reduces communication costs ultimately



64 subdomains

2.5.1 Load balancing algorithms (cont) *PCAM*

And many more … (complex unstructured meshes) (divides vertices by distance, edges)

- ✓ Recursive graph bisection
- ✓ Recursive spectral bisection
- ✓ …

2.5.1 Load balancing algorithms (cont)

Local algorithms

The preceding methods require a ***GLOBAL*** knowledge of the computation state

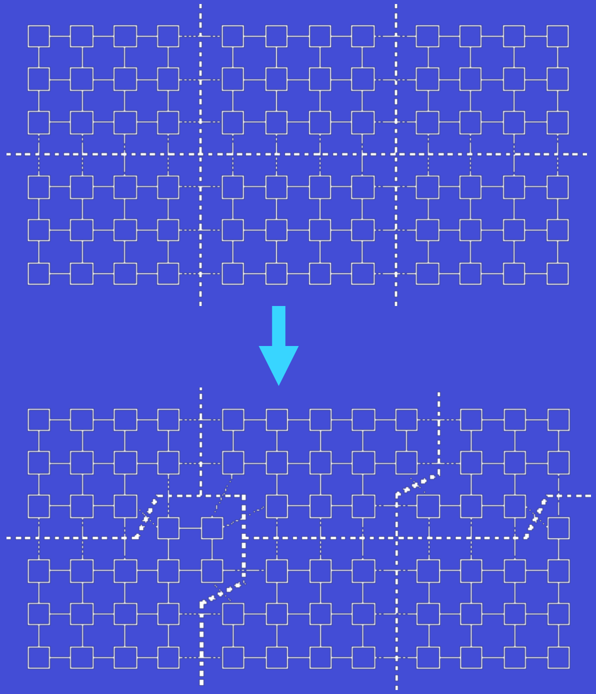
⇒ Bad … Expensive …

LOCAL algorithms compensate for change in computational load using only information from a small number of neighbouring processors

e.g. If processors organised into a logical mesh:

Periodically, processors check computational load with neighbours.

Transfer computation if the difference in load exceeds some threshold.



Pros/Cons:

✓ Good if the load is constantly changing, since CHEAP

× Less good at load balancing than global in general

× Can be slow to adjust to major load disturbances (takes many load balancing adjustments to "diffuse" away problem)

2.5.1 Load balancing algorithms (cont)

Probabilistic Methods

Allocate (fine-grained) tasks randomly!

If the number of tasks is large, we may expect that each processor will be allocated roughly the same amount of work.

Pros/Cons:

- ✓ Very simple
- ✓ Low cost
- ✓ Very scalable
- ✗ Off-processor communication required for likely every processor
- ✗ Acceptable load distribution only if there are lots more tasks than processors
- ✗ Most effective when there is relatively little communication between tasks and/or little locality in the communication structure anyway. Otherwise end up with more communication in general.

2.5.1 Load balancing algorithms (cont)

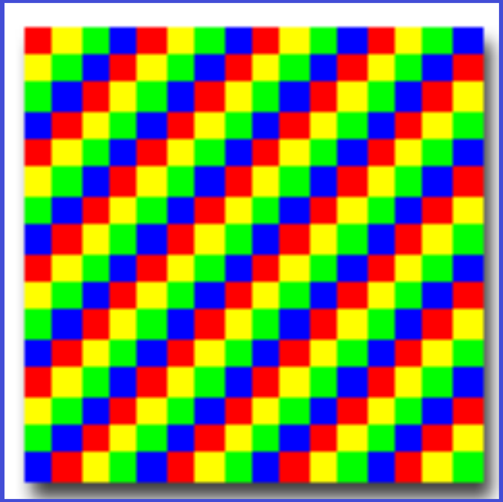
Cyclic Mapping

If

- ✓ variable computational load per grid point
- ✓ significant spatial locality to levels of load

Then maybe good to use *cyclic (or scattered) mapping*:

Assign each of the P processors with every Pth task according to some numbering of the tasks.

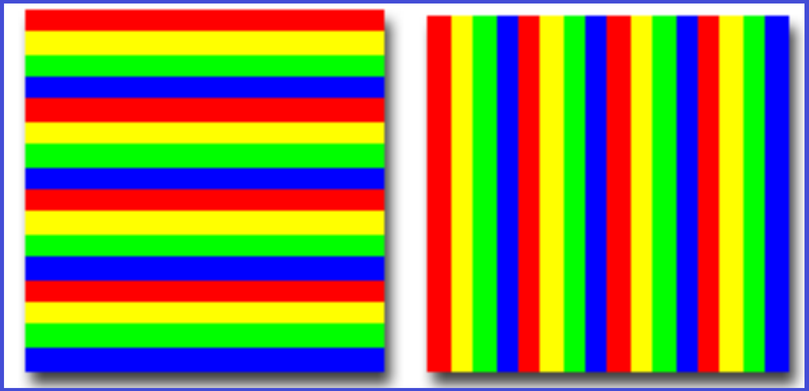


Pros/cons:

✓ Notice is really a form of probabilistic since it depends on the enumeration of the processors. Therefore again, on average, roughly equal load.

✗ May have reduced locality (e.g. if this was a 5-point stencil)

=> Weigh improved load balance vs. increased communication costs



Can of course do block cyclic distributions etc

2.5.2 Task scheduling algorithms

Can be used when a functional decomposition yields many tasks, each with weak locality.

A centralised or distributed ***TASK POOL*** is maintained :(cf. taxicabs with dispatcher)

- New tasks are placed in the pool.
- Tasks are taken from the pool and allocated to processors for processing. Obviously, hard part is strategy for assigning tasks to worker processors (manager)

Compromise between independent operation (minimise comm) and global knowledge of computational state (improves load balance)

Manager/worker:

Manager maintains problems

Workers request, manager sends

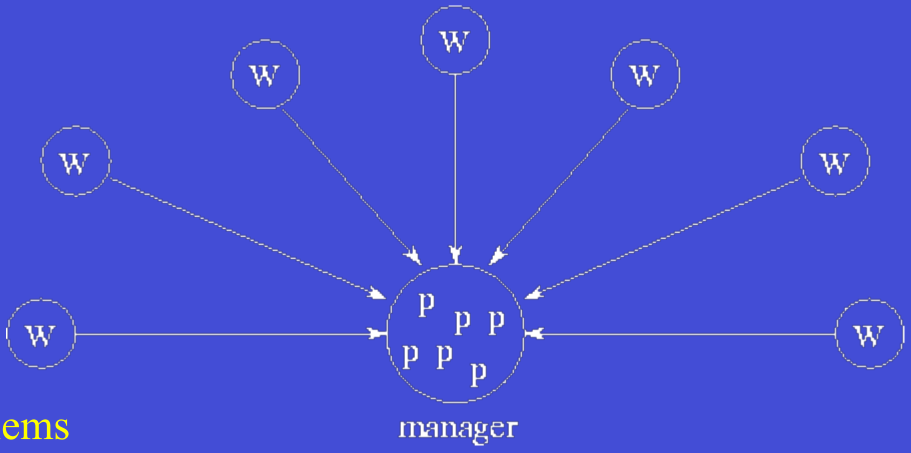
Workers can send new problems to manager

Efficiency depends on number of workers

and relative costs of obtaining vs executing problems

Prefetching/caching problems helps

Can do hierarchical manager/worker (manager for sub-domain of workers)



2.5.2 Task scheduling algorithms (cont)

P C A M

Decentralised schemes:

No central manager

Separate task pool maintained on each processor

Idle workers request tasks from any processor's pool

Task pool is distributed, accessed asynchronously

Variety of access policies can be defined:

- o Access only from small number of pre-defined neighbours
- o Access from other processors at random

Scalable!

Workers must poll for requests (pain)

Or could separate computation and task pool management to different task

Problem: Termination

Workers need to know when to stop!

Easy in centralised of course.

Not so easy in decentralised/distributed (no central record, plus complications e.g. workers all idle but problem still ongoing because are all communicating)

2.5.3 Mapping Design Checklist

P C A M

Mapping decisions seek to balance conflicting requirements of equitable load distribution and low communication costs.

When possible, we use a static mapping scheme that allocates each task to a single processor.

However, if number or size of tasks are not known until runtime (e.g. AMR) may need a dynamic load balancing scheme or reformulate for task scheduling

- ✓ If considering a SPMD design for complex problem, have you also considered dynamic task creation and deletion? Can be simpler, although performance may be problematic (cf dynamic array allocation in Fortran90)
- ✓ If considering a design based on dynamic creation and deletion of tasks, have you considered SPMD? SPMD gives greater control over scheduling but can be complex
- ✓ If using a centralised load balancing scheme, have you confirmed that the manager task will not become a bottleneck? (maybe reduce comm by passing pointers to tasks rather than tasks themselves to the manager etc)
- ✓ If using dynamic load balancing, have you evaluated the relative costs of different strategies? e.g. tried probabilistic or cyclic (simple, cheap)?
- ✓ If using probabilistic or cyclic mapping, do you have a large enough no. of tasks to ensure load balance is reasonable? (order of mag more tasks than procs?)

Chapter 2: Summary

- ✓ Partition
- ✓ Communication
- ✓ Agglomeration
- ✓ Map

Try and accommodate our desires of Concurrency, Scalability, Locality, (Modularity)

Design done? Ready to write the code?

Not quite!

Some stages of the design process remain:

- We should check to see if our design is likely to meet our performance goals
- Do some simple performance analysis to be able to choose between alternative algorithms
- Think about implementations costs (how easy to write code, code re-use etc)
- Think about how our algorithms may fit with other parts of a larger system

=> Chap 3, Chap 4, …

End of Chapter 2

Case Study 1: Atmospheric Model

Conservation of momentum:

$$\frac{\partial u}{\partial t} - \left(f + u \frac{\tan\phi}{a} \right) v = -\frac{1}{\rho a \cos\phi} \frac{\partial p}{\partial \lambda} + F_\lambda$$

$$\frac{\partial v}{\partial t} + \left(f + u \frac{\tan\phi}{a} \right) v = -\frac{1}{\rho a} \frac{\partial p}{\partial \phi} + F_\phi$$

Solution in two horizontal directions, latitude and longitude

Hydrostatic approximation:

$$g = -\frac{1}{\rho} \frac{\partial p}{\partial z}$$

Coupling in vertical direction

Conservation of mass:

$$\frac{\partial \rho}{\partial t} = -\frac{1}{a\cos\phi} \left(\frac{\partial(\rho u)}{\partial \lambda} + \frac{\partial(\rho v \cos\phi)}{\partial \phi} \right) - \frac{\partial(\rho w)}{\partial z}$$

Horizontal and vertical gradients

Conservation of energy:

$$C_p \frac{\partial T}{\partial t} - \frac{1}{\rho} \frac{\partial p}{\partial t} = Q$$

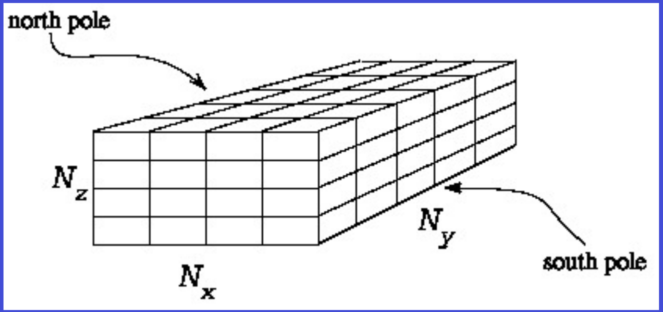
Connections between variables

State equation (atmosphere):

$$p = R\rho T$$

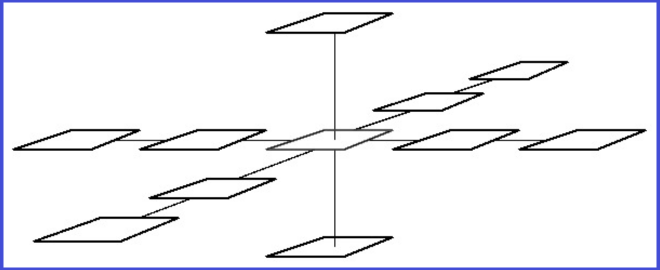
Case Study 1: Atmospheric Model (cont)

"Atmosphere in a box" :



Derivatives grid stencil:

- ✓ 9 points in horizontal
- ✓ 3 points in vertical



Other necessary operations:

$$\text{Total mass} = \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} \sum_{k=0}^{Nz-1} M_{i,j,k}$$

Other processes ("extra physics"=precipitation, clouds, radiation): SAY, LOTS OF THEM AND LOTS OF COMPUTATION Complicated accumulations of all z levels only, no x dependence

$$TCS_k = \prod_{i=1}^{k}(1-cld_i)TCS_1$$
$$= TCS_{k-1}(1-cld_k)_1$$

Student's task:

Go through algorithmic design process (PCAM) for this problem …

Case Study 2: Chip Real Estate

Arranging electronic components on a chip.

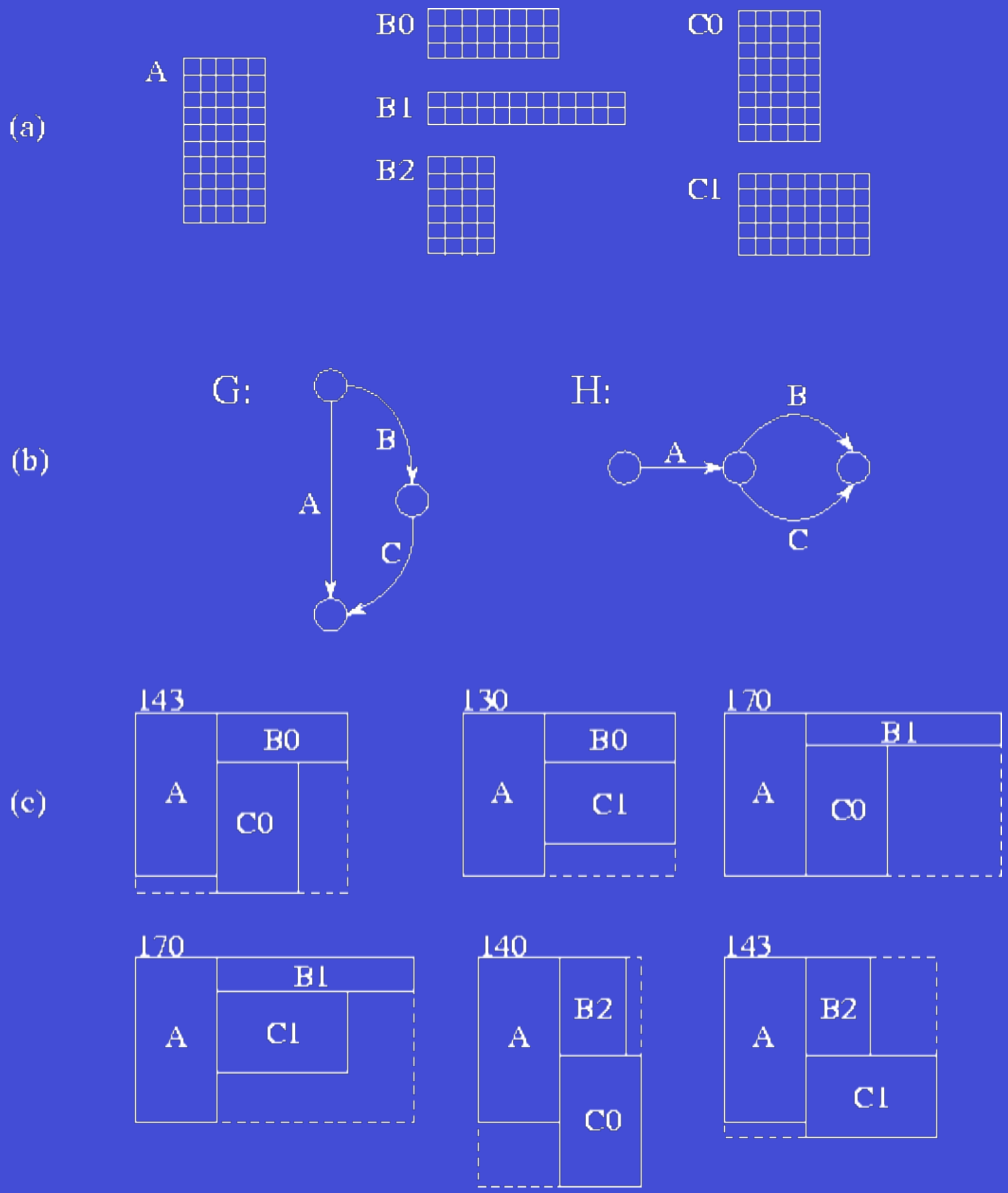
Real estate is a big issue=> Minimise total area

Components have to be in certain order (to connect, heat etc; e.g. above and to left of …)

But might be able to have various configurations

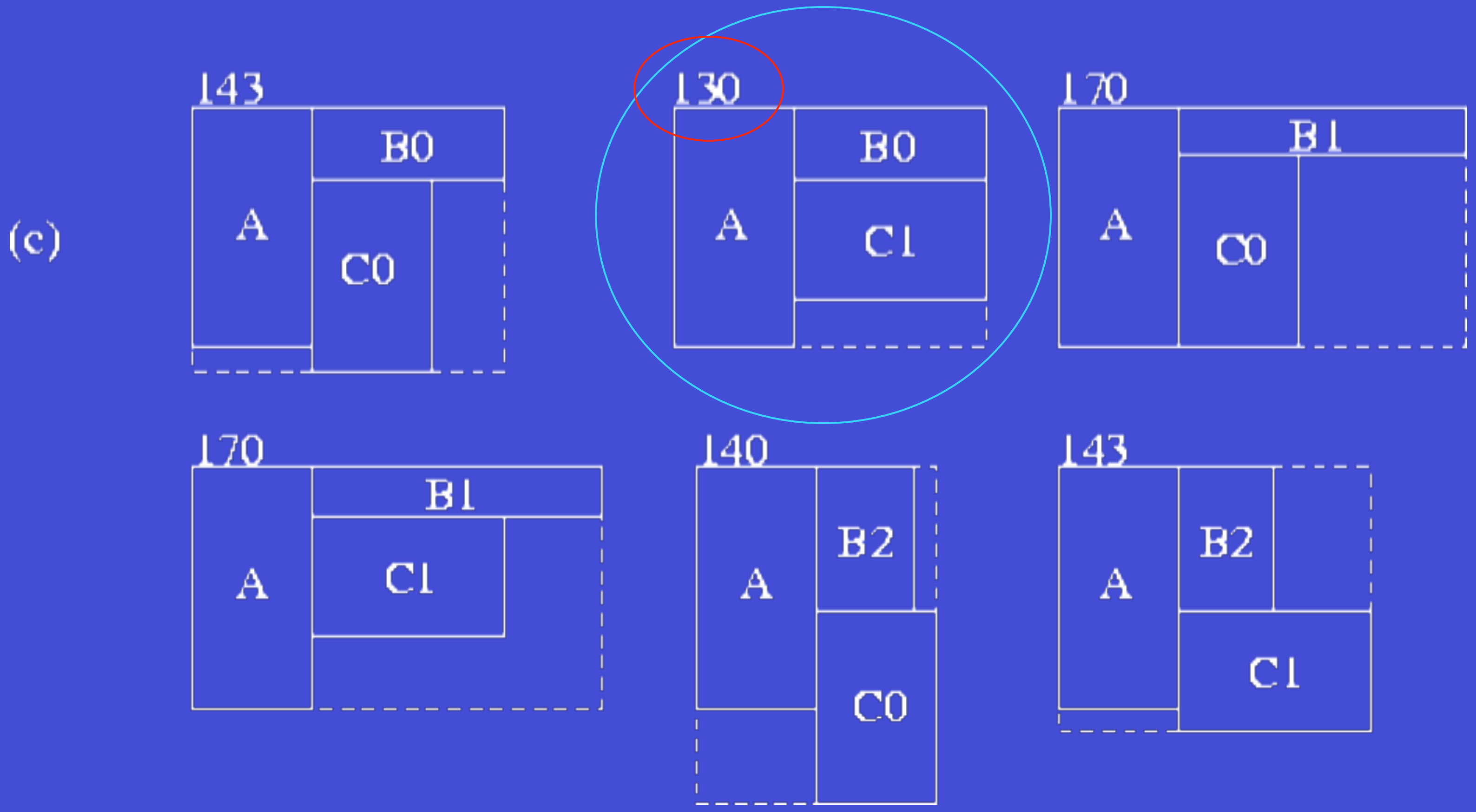
Treat as "cells" of specific volume, various allowable configurations, order specified

=> Search through set of all possible solutions



Case Study 2: Chip Real Estate (cont)

Solution:

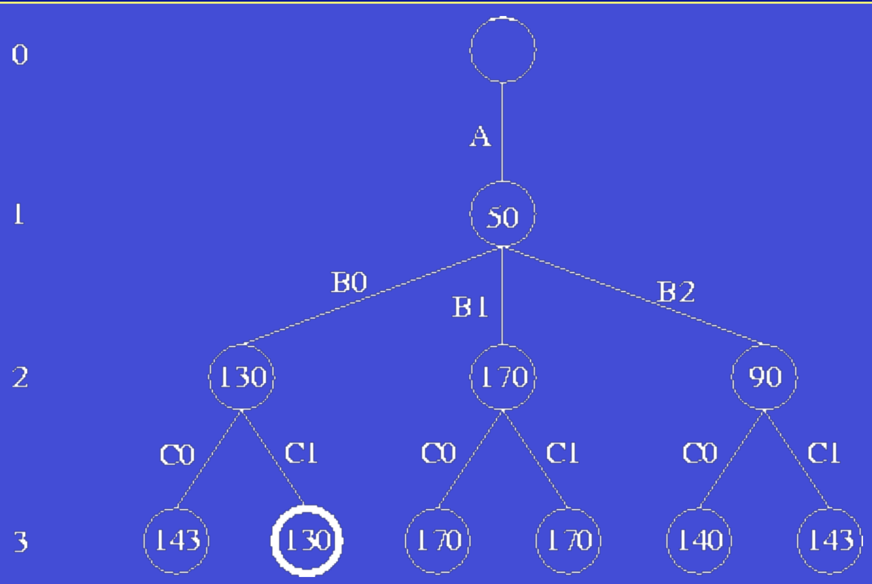


Case Study 2: Chip Real Estate (cont)

How did you get there?:

Depth-first exhaustive search

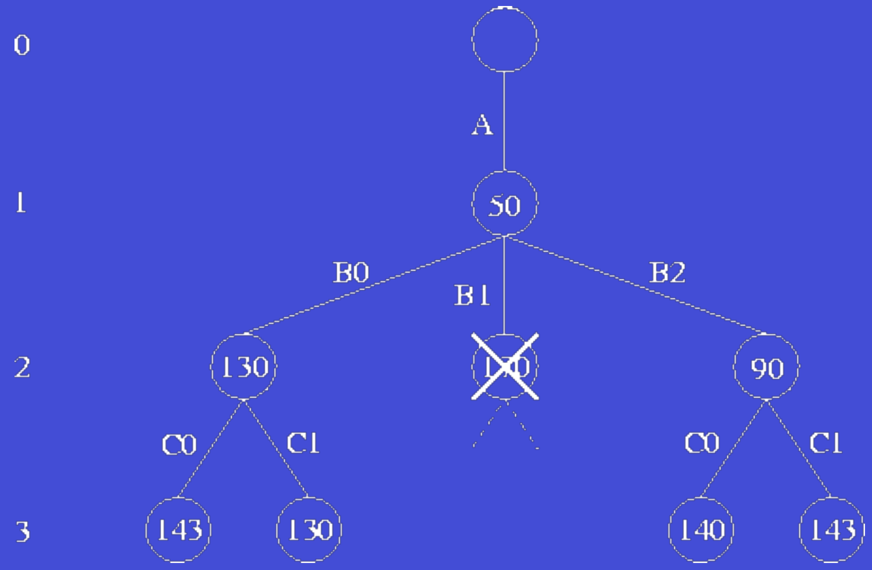
(we already did logic for this)



Faster: Branch-and-bound

Incorporates "pruning":

If area already greater than best known solution, quit this branch



Case Study 2: Chip Real Estate (cont)

Logic:

```
Procedure bnb_search(A)
Begin
  A_min = infinity
  bnb_search_1(A)
End
Procedure bnb_search_1(A)
Begin
  score=eval(A)
  if (score < A_min) then
    if (leaf(A)) then
      A_min=score
      report soln,score
    else
      foreach (child A_i of A)
        bnb_search_1(A_i)
      endfor
     endif
   endif
end
```

Students: Discuss PCAM for this problem …