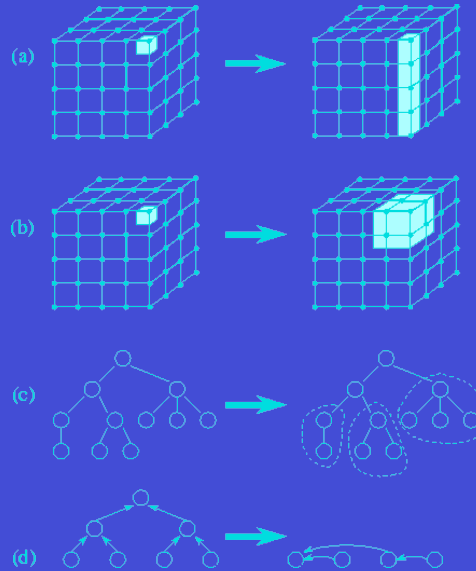


## 2.4.1 Increasing granularity (cont)

PCAM

Examples of agglomerating tasks:



## 2.4.1 Increasing granularity (cont)

PCAM

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

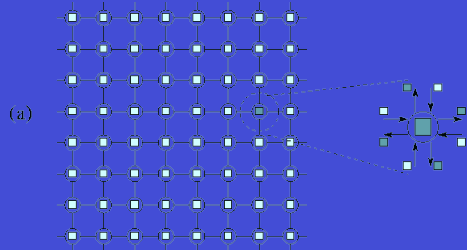
This is a “surface-to-volume” effect: (clear in domain decomposition but true for all)

- Communication costs are proportional to the “surface area” of the sub-domain.
- Computation costs are proportional to the “volume” of the sub-domain.
- $\Rightarrow$  ratio  $\sim N^2/N^3 = 1/N \Rightarrow$  decreases with increasing task size
- $\Rightarrow$  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

$8 \times 8 = 64$  tasks

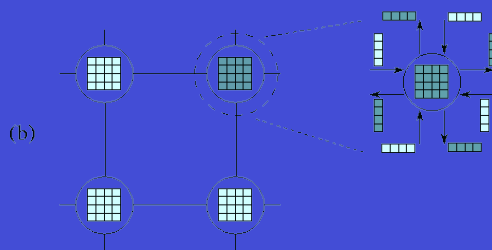
4 communications per task

$\Rightarrow$  Total 256 communications

1 datum per comm  $\Rightarrow$  Total 256 data transferred

1 computation per task

$\Rightarrow$  comm/comp =  $4/1 = 4$



Costs: Agglomerated

$2 \times 2 = 4$  tasks

4 communications per task

$\Rightarrow$  Total 16 communications

4 data per comm  $\Rightarrow$  Total 64 data transferred

16 computation per task

$\Rightarrow$  comm/comp =  $4/16 = 1/4$  (or =1 for total data)

## 2.4.1 Increasing granularity (cont)

P C A M

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

$\Rightarrow$  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

PCAM

Can sometimes trade-off replicated computation for reduced communication

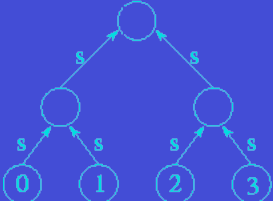
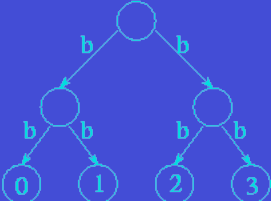
Example: Summation problem (see earlier)

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)

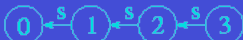
PCAM

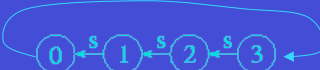
These algorithms are optimal in the sense that no computation is repeated : minimal computations

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 eac 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:  $N*(N-1) \Rightarrow (N-1)^2$

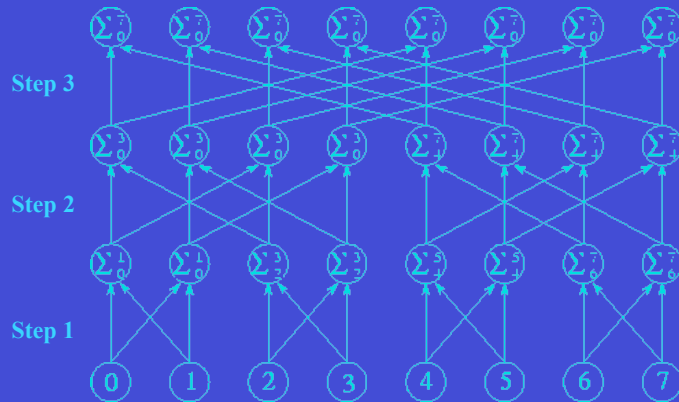
Redundant communications :  $(n-2)*(N-1) \sim (N-1)^2$

## 2.4.1 Replicating computation (cont)

PCAM

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



$\log_2(N)$  steps

N comms per step

=>

$O(N \log_2 N)$  comms

$O(N \log_2 N)$  comms

$\ll O(N^2)$  !!

Less steps, less redundancy than ring

## 2.4.1 Replicating computation (cont)

PCAM

Agglomeration of tree/butterfly:

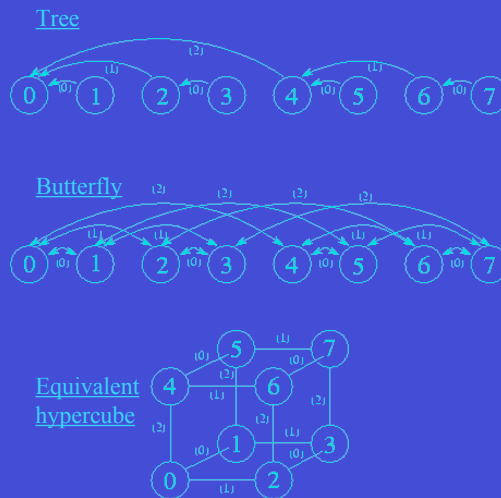
Agglomeration almost always useful if analysis of communication reveals that tasks cannot execute concurrently

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

PCAM

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  $N_{procs\_max} \sim$  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 ( $\Rightarrow$  scalability)

## 2.4.3 Reducing software engineering costs

PCAM

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

Should take into account relative development costs too.

Particularly important when parallelising 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

## 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 tractable way of evaluating the general case. Therefore present rough classification of problems and some representative techniques ...

## 2.5 Mapping (cont)

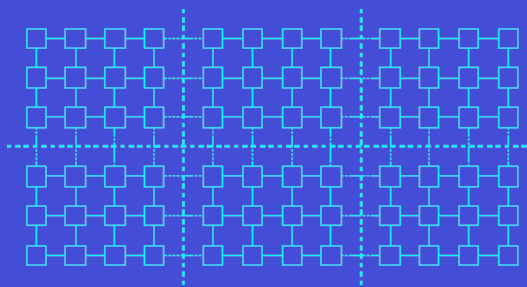
PCAM

### 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)

PCAM

### 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](#) have less overhead

### 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)

PCAM

### 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 techniques briefly ...*

## 2.5.1 Load balancing algorithms

PCAM

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 communication 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)

PCAM

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

Advantages and disadvantages:

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

## 2.5.1 Load balancing algorithms (cont)

PCAM

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

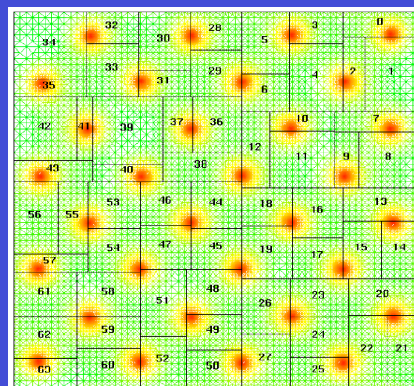
$2/P$  and  $(P-2)/P$  of the load

... 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)

PCAM

### 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

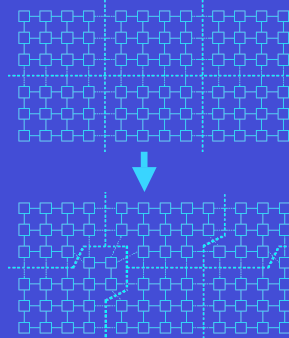
e.g. If processors organised into a logical mesh:

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)

P C A M

### Probabilistic Methods

Allocate 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)

P C A M

### Cyclic Mapping

If

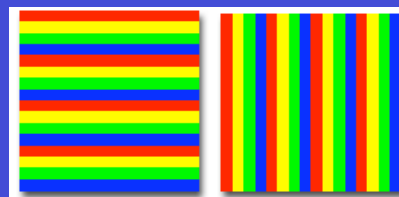
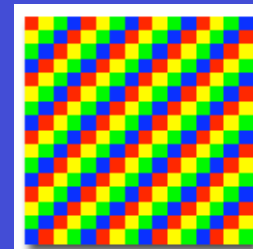
- ✓ variable computational load per grid point
- ✓ significant spatial locality to levels of load

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

Assign each of the P processors with every P<sup>th</sup> 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

PCAM

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

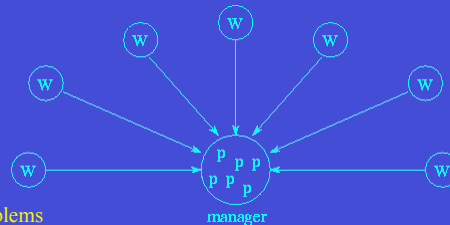
Workers can send new problems to manager

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)

PCAM

### 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

PCAM

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, ...**