

### 3.4 Scalability analysis (cont)

e.g. Scaled problem size analysis: isoefficiency of 2D decomposition finite-difference

For 2D decomposition of finite-difference mesh  $N \times N \times N_z$ ,

each processor has  $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}} \times N_z$  points to compute

And must exchange  $2 \frac{N}{\sqrt{P}} N_z$  points with each of 4 neighbours

Therefore,

$$E = \frac{t_c N^2 N_z}{t_c N^2 N_z + 4 P t_s + 8 \sqrt{P} t_w N N_z}$$

For constant efficiency,

$$t_c N^2 N_z \sim E (t_c N^2 N_z + 4 P t_s + 8 \sqrt{P} t_w N N_z)$$

This is satisfied by  $N \sim \sqrt{P}$

Total computation  $O(N^2) \Rightarrow$  for constant efficiency, computation must scale as  $\sim O(P)$

$\Rightarrow$  2D decomposition is **MORE SCALABLE** than 1D

$\Rightarrow$  **Higher dimensional decompositions are more efficient** (again) (surface-to-volume effect: sends more messages [4 instead of 2] but less data  $N \sqrt{P}$  instead of  $NP \Rightarrow$  comm reduced unless  $P$  small [ $P < 4$ ] or  $t_s$  is much larger than  $t_w$ )

### 3.4 Scalability analysis (cont)

#### 3.4.3 Execution profiles

If scalability analysis suggests that the performance is poor on problem sizes and machines of interest, we can use models to identify likely sources of inefficiency and hence areas for possible improvement.

Poor performance could be due to excessive

- ✓ replicated computation
- ✓ idle time
- ✓ message startups
- ✓ data transfer costs

Plot contributions of these things to total execution time

## 3.4 Scalability analysis (cont)

e.g. 1D decomposition finite-difference algorithm:

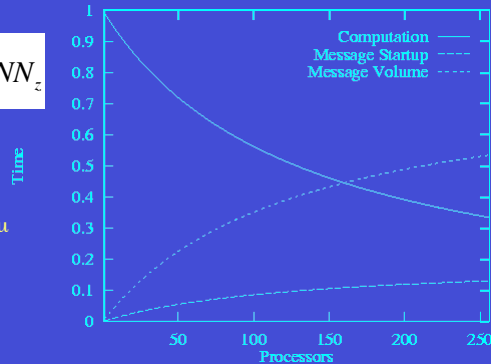
$$T_{1D\_finite\_diff} = \frac{t_c N^2 N_z}{P} + 2t_s + 4t_w N N_z$$

Fractional contributions of computation, message startup and message transfer costs (no replicated comp, no idle time)

N=512, Nz=1, ts=200µsec, tw=0.8µsec, tc=1µsec

Varying P, immediately see:

- ✓ For large P, dominated by data transfer
- ✓ For large P, message startup not insignificant
- ✓ Increase Nz, startup costs decrease and efficiency improves



Can use such techniques to refine design. e.g.

Replicated comp dominating? Maybe swap for increased comm.

High message startup? Agglomerate more.

Data transfer high? Replicate more comp.

## 3.5 Experimental studies

In the previous section, we emphasized analytical modelling of performance.

However, HPC is primarily an experimental discipline! Our modelling really only assists an essentially experimental process, by guiding experiments and explaining results. Many skip the modelling altogether (heathens).

Experimental studies good in early design stages for evaluation of parameters in performance models (message startup cost, data transfer cost, average cost per grid point, average depth of a search tree etc)

In later design stages, compare models with observed performance of coded algorithm.

### 3.5.1. Experimental Design

Identify the data we wish to obtain:

- Execution time of a serial version as a function of problem size to obtain tc?
- Execution time of a simple message-passing program to obtain ts, tw?

Perform experiments for a range of problem sizes and/or processor counts:

- Reduce the impact of errors in individual timing measurements.
- Also may expose where models are weak.

## 3.5 Experimental studies (cont)

### 3.5.2 Obtaining and Validating Experimental Data

It is a challenge to obtain accurate and reproducible results.

Execution times can be obtained in a variety of ways:

- Introduce timers into the code itself (but one on each processor? Take max? Avg? Representative processor?)
- Profiling or tracing tools

Experiments should be repeated to verify the results. Should not vary by more than 2-3% if fine-tuning an algorithm.

However, execution times can fluctuate wildly depending on many factors:

Non-deterministic algorithm: e.g. use of random numbers; search tree; dynamic allocation of tasks to processors. Random nos can be controlled using a reproducible random number generator. Dynamic: run a lot of trials and average or normalise by some measure of the amount of work done (e.g. no. of search nodes visited)

Inaccurate timers: Some timers do not have very good resolution or are plain inaccurate. Decrease relative error by increasing execution times (e.g. more work, more iterations, ...)

Startup and shutdown costs: Need to exclude system management components associated with acquisition of processors, loading code, allocation of virtual memory etc. Make sure you time only the relevant bits of the code.

## 3.5 Experimental studies (cont)

Interference from other programs: On a non-dedicated machine, other users compete with you for resources such as processor use, network bandwidth, i/o bandwidth etc. System functions such as accounting, backups, etc can affect too. Competition can occur even if the processors are dedicated. e.g. domain decomposition may have many subdomains that traverse the same sub-network and therefore compete for i/o bandwidth (you can program to avoid this though).

Contention: Multiple messages sent at the same time may contend for network bandwidth. Some networks (e.g. Ethernet-connected LAN) can only send one message at a time, and so multiple messages must be backed-off and resent, increasing the latency time. Bad scheduling can result in repeated sendings = increased execution time. Improve scheduling! (randomise?)

Random resource allocation: The operating system may schedule processors for use in a random manner, which can affect execution times if communication costs depend on the processor location in a network. Force the same processor allocation for all experiments?

Best approach:

- ✓ Measure things for significant amount of execution time
- ✓ Measure things numerous times
- ✓ Measure things several different ways (redundancy: time components of code AND total)

## 3.5 Experimental studies (cont)

### 3.5.3 Fitting Data to Models

e.g. trying to determine  $t_s$  and  $t_w$  in  $T_{\text{msg}} = t_s + t_w L$

Obvious methods:

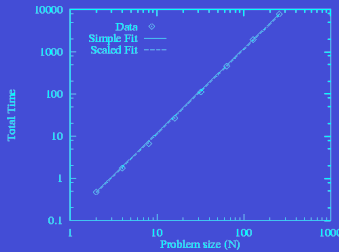
- ✓ Linear fit
- ✓ Least squares fit : minimise  $\sum (obs(i) - model(i))^2$
- ✓ Scaled least squares fit  $\sum \left( \frac{obs(i) - model(i)}{obs(i)} \right)^2$  (gives more weight to smaller but desirable high P results)

e.g. 1D finite-difference

Determine  $t_c$  from  $t_{\text{comp}} = t_c N^2 N_z$

Do 3 expts for each N (Nz fixed, SPARC, non-ded)

N	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	Mean
2	0.477	0.471	0.479	0.476
4	1.75	1.73	1.73	1.74
8	6.62	6.63	6.68	6.64
16	26.9	26.9	26.4	26.7
32	112	112	112	112
64	459	459	460	459
128	1930	1929	1934	1931
256	7949	7873	7897	7906



N	Performance Model	
	Observed	Simple Scaled
2	0.476	0.480 0.448
4	1.74	1.92 1.79
8	6.64	7.68 7.16
16	26.7	30.7 28.7
32	112	123 115
64	459	491 459
128	1931	1966 1835
256	7906	7864 7340

Simple linear  $t_c=0.0120\text{msec}$

Scaled LS  $t_c=0.0112\text{msec}$

Model pretty good!

## 3.6 Evaluating the implementation

As well as helping with the design of the algorithm, evaluation of performance models can be important after the code is implemented. Wish to use model for validation and verification of implementation.

Even if considerable care is taken in designing and carrying out experiments, should still **not** expect observed and predicted to agree perfectly. However, hope differences can help assess where implementation was inefficient or where the model is deficient.

Process:

1. First check for mistakes!
2. Then check that model and implementation are really measuring the same thing!
3. Then obtain execution profile from the implementation. Get more detailed view of program behaviour:
  - ✓ Time in initialisation
  - ✓ Time in different phases of the computation
  - ✓ Idle time
  - ✓ Total no. and volume of messages communicated
  - ✓ Do for range of problem sizes and processor counts.

## 3.6 Evaluating the implementation (cont)

Potential problems that may be unearthed:

### 3.6.1 Execution time slower than expected

May indicate:

**Load imbalances:** Action - Study/profile costs in individual processors

**Replicated computation:** we failed to parallelise something important! Did we assume falsely that replicating something might be cheap? Action - Study costs in different segments of the code using varying P.

**Competition for bandwidth:** The simple linear model for communication costs might not be good enough, due to competition for bandwidth etc (Action - *more later*)

**Tool/algorithm mismatch:** Our conceived algorithm may just not implement well with the available tools (e.g. overlapped comp/comm by many tasks in one processor: may not work well if library does not implement tasks efficiently)

## 3.6 Evaluating the implementation (cont)

### 3.6.2 Execution time faster than expected

Wahoo! But, best to know what is going on ... (no action necessarily required!)

If affect become more marked as P increases = **speedup anomaly**.

Speedup may be greater than linear -- **superlinear!**

Possible causes:

**Cache effects:** Each processor has a small amount of fast local memory = **cache**. On larger numbers of processors, a fixed size problem gets more of its data into cache  $\Rightarrow$  total computation time  $T_{\text{comp}}$  will decrease. If this more than compensates for the extra overheads of more processors ( $T_{\text{comm}}, T_{\text{idle}}$ ) then speedup can be superlinear, efficiency  $> 1!$  Similarly, the increased physical memory in a multiprocessor may reduce the cost of memory accesses by avoiding the need for virtual memory paging.

**Search anomalies:** e.g. Parallel branch-and-bound algorithm just plain faster if more concurrent searches! If solns exist at varying depths in the tree, then multiple depth-first searches will, on average, explore fewer tree nodes before finding a solution than a sequential depth-first. Parallel algorithm is **NOT** the same as the sequential algorithm in this case.

### 3.6 Evaluating the implementation (cont)

**Example: 1D decomposition finite-difference** (N=512, NZ=10, Intel Delta ts=200, tw=2μsec)

An example of one thing that might happen:

Speedup rather than raw execution time to make results clearer for larger P

Model predicts continuous speedup with P

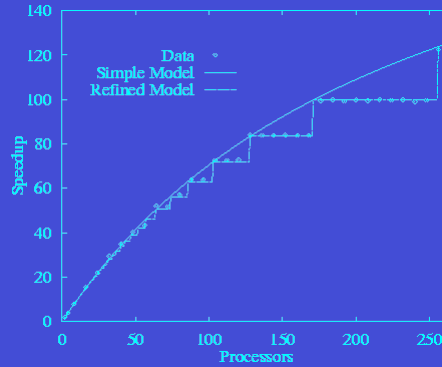
Observations show stepwise speedup

Execution time for algorithm:

$$T_{1D\_finite\_diff} = \frac{t_c N^2 N_z}{P} + 2t_s + 4t_w N N_z$$

Comp times depend on P

Comm times do not depend on P



We assumed that N is divisible by P and so each processor has N/P slices of data

But when varying P, for some P, N is NOT divisible by P!

e.g. N=8, P=3 => 2 processors have 3\*N\*Nz work and 1 has 2\*N\*Nz

Wall time set by the processor with the MAX amount of work = 3\*N\*Nz

Obviously this would be THE SAME for N=9, P=3 (now ALL processors have 3\*N\*Nz work)

### 3.6 Evaluating the implementation (cont)

The uneven distribution of computation when N is not divisible by P leads to idle time:

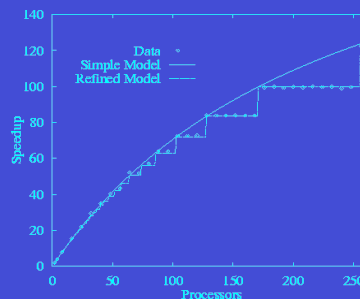
$$T_{comp\_MAX} = \text{ceiling}(N/P) N N_z$$

$$T_{idle} = \sum_{i=0}^{P-1} (T_{comp\_MAX} - T_{comp\_i}) = P T_{comp\_MAX} - T_{comp}$$

Incorporating these thoughts into the original model, we get the refined model:

$$T_{1D\_finite\_diff} = t_c N N_z \text{ceil}[N/P] + 2t_s + 4t_w N N_z$$

This is what is shown on the graph:



## 3.7 A refined communication cost model

As you can see, there is plenty of scope for **refining** our models!

As mentioned earlier, it is entirely possible that our simple communication model based on  $T_{\text{msg}} = t_s + t_w L$  is **not sufficient**, in particular because it **does not account for competition for bandwidth**.

The most serious impact of competition for bandwidth is in algorithms that operate synchronously, in lockstep, that are well-load-balanced => many **SPMD** applications!

Furthermore, the **interconnect network** generally use fewer than  $N^2$  wires to connect  $N$  processors. Therefore, they must include switches (routing nodes). Switching nodes may **block or re-route** messages when several messages require access simultaneously.

Obvious improvement to a communication model is to think of the processors as **SHARING** the available bandwidth.

Scale the data volume by the no. of processors  $S$  that may need to send concurrently over the same wire

i.e. 
$$T_{\text{msg}} = t_s + t_w L S$$

The effective bandwidth available to each processor is  $1/S$  of the true bandwidth.

Does not account for re-transmitted messages but is often adequate.

## 3.7 A refined communication cost model (cont)

The value of  $S$  depends on the properties of the parallel algorithm and the interconnection network. So now need to know something about interconnects!

**Crossbar switching:** avoids contention/competition by using  $O(N^2)$  switches =>  $S=1$   
Highly non-scalable however! Used only for a small no of processors

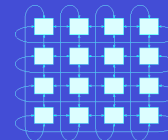
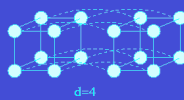
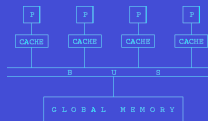
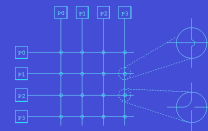
**Bus-based networks:** Processors share the bus =>  $S = \text{no. of procs trying to communicate}$ . Highly non-scalable! Common in SMP machines -- simplify programming by removing locality. Requires caches to avoid bus clashes => reintroduce locality!

**Ethernet:** = slow bus-based network! =>  $S = \text{no. of procs sending simultaneously}$ .

**Mesh networks:** Same as a cross-bar switch excepts the switching nodes are the processors themselves (in the mesh rather than at the edges). For mesh of dimension  $D$ , each processors is connect to  $2D$  neighbours. Extend with toroidal connections.  $S$  depends on algorithm (more later).

**Hypercube:** Dimension  $D$  connects to  $2^D$  "neighbours". Like a mesh plus long-wire connections. Complex. Cannot be laid out in 3D so that all are real neighbours.

**Multistage interconnect networks (MIN):** Like crossbar, switching elements are distinct form processors, but in MIN, there are fewer than  $O(N^2)$



### 3.7 A refined communication cost model (cont)

#### e.g. Competition for bandwidth in canonical finite-difference problem

Algorithm with high degree of locality

Per processor communication costs are  $T_{comm\_ideal} = 2t_s + t_w 4 N N_z$

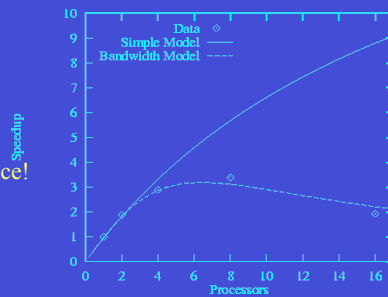
**Mesh or in a hypercube:** No competition in these networks since the one-dimensional ring communication structure embeds in these networks using only nearest-neighbour connections => S=1

**Bus-based network:** only one of the P processors can communicate at any one time. In the algorithm, half the processors need to send at once (max; other half are receiving) => S=P/2 and so

$$T_{comm\_bus} = 2t_s + t_w 2 P N N_z$$

Large impact even on a simple algorithm

Much improved model of the bad performance!



N=512 Nz=5, Ethernet-connected workstations, ts=1500 tw=5

### 3.7 A refined communication cost model (cont)

#### e.g. Competition for bandwidth in butterfly

Algorithm where P tasks use the butterfly (or hypercube) communication structure

The P tasks do log(P) exchanges of N/P data and therefore, in absence of competition, the per-processor communication  $T_{comm\_ideal} = \log(P)(t_s + t_w \frac{N}{P})$

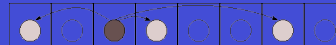
**Crossbar:** no competition

**P-processor hypercube:** no competition (can organise s.t. nearest neighbour comm only)

**Bus:** S=P/2 as before =>  $T_{comm\_ideal} = \log(P)(t_s + t_w \frac{N}{2})$

**Mesh:** Limited number of wires is an issue.

e.g. 1D mesh of P processors.



Each processor generates messages that must travel  $1, 2, \dots, 2^{\log P - 1}$  hops along the mesh

$$\Rightarrow \text{Total no. of hops done in process} = P \sum_{i=0}^{\log P - 1} 2^i = P(P-1)$$

BUT the bi-directional mesh only has  $2(P-1)$  wires

=> min no of steps to complete algorithm = P/2 NOT log(P) due to conflicts

$$\Rightarrow S=P/2 \quad T_{comm\_ideal} = \log(P)(t_s + t_w \frac{N}{2})$$

## 3.8 Input/output

Performance is affected by the movement of data to disk.

Generally an inhibitor to parallelism and efficiency.

Can be thought of as a communication problem **BUT WORSE**:

- Startup costs a lot more
- Highly dependent on system configuration
  - Single disk on master node?
  - Disks attached to each node?
  - Multiple paths to same disk?

### Strategies

**Parallel i/o**: Allows concurrent read/write operations but difficult bookkeeping. Parallel i/o (MPI, NetCDF) and filesystems (GPFS, PVFS) exist but are *immature*!

**Centralised i/o**: Easy bookkeeping, possible bottlenecks (inefficient) (make asynchronous?)

### Note:

A lot of small read/writes is bad. Try to agglomerate

Distribution required on disk may be different from that in memory => work to redistribute

Data may be “striped” on disks (safety measure) = more complex!

## 3.9 Chapter 3 Summary

We have developed mathematical performance models that characterise

*execution time, efficiency, scalability*

in terms of the parameters

*the problem size, the no. of processors, the communication parameters*

We use these performance models in design and implementation cycle:

- **Early** in design process to
  - Choose parallel algorithms
  - Identify problem areas
  - Verify we can meet design specifications
- **Later** in design process
  - Refine models and increase confidence in design
  - Determine unknown parameters (costs)
- **During** implementation
  - Identify implementation errors
  - Refine models