Chapter 3: A Quantative Basis for Design

Real design tries to reach an optimal compromise between a number of thing

- ✓ Execution time
- ✓ Memory requirements
- ✓ Implementation costs
- ✓ Simplicity
- ✓ Portability
- ✓ Etc

Here try and form an understanding and make some estimates of costs by creating performance models.

Try and

- ✓ Compare efficiency of different algorithms
- ✓ Evaluate scalability
- ✓ Identify bottlenecks and other inefficiencies

BEFORE we put significant effort into implementation (coding).

Chapter 3: A Quantative Basis for Design

Goals:

- ✓ Develop predictive performance models
- ✓ Obtain empirical performance data and use to validate performance models
- ✓ Evaluate/predict scalability
- ✓ Choose between different algorithms
- ✓ Understand how hardware network topology affects communication performance
- ✓ Account for these effects in models
- ✓ Recognize and account for factors other than performance e.g. implementation costs

3.1 Defining Performance

Defining *"performance"* is a complex issue:

e.g. weather forecasting

- ? Must be completed in a max time (e.g. within 4 hours) => execution time metric
- ? Fidelity must be maximized (how much comp can you do to make realistic in that time)
- ? Minimize implementation and hardware costs
- ? Reliability
- ? Scalability

e.g. parallel data base search

- ? Runs faster than existing sequential program
- ? Scalability is less critical (database not getting order of mag larger)
- ? Easily adaptable at later date (modularity)
- ? Needs to be built quickly to meet deadline (implementation costs)

e.g. image processing pipeline

- ? Metric not total time but rather no of images can process per second (throughput)
- ? Or time it takes to process a single image
- ? Things might need to react in ~ real time (sensor)

3.1 Defining Performance

So performance =

Execution time	✓✓	Design costs	✓✓ (prof)
Scalability	✓✓	Implementation costs	✓ (grad student)
Correctness (fidelity)	✓✓✓	Verification costs	✓ (who bothers!)
Memory		Potential for re-use	✓
Throughput		Hardware requirements	
Latency		Hardware costs	
I/O rates		Maintenance costs	
Network throughput		Portability	✓✓

… !!!

Depends on :

- ✓ Computational kernel
- ✓ Communication infrastructure
- ✓ Actual hardware (CPU, network hardware, disks etc)

3.2 Approaches to performance modelling

Three common approaches to the characterization of the performance of parallel algorithms:

- ✓ Amdahl’s Law
- ✓ Observations/measurements
- ✓ Asymptotic analysis

We shall find that these are mostly ***inadequate*** for our purposes!

3.2.1 Amdahl's Law

Define:

Speedup = (execution time on single processor)/(execution time on multiple processors)

Except for embarrassingly parallel applications, every parallel application has a sequential component.

Amdahl's Law says that, eventually, the sequential component will limit the parallel speedup.

If fraction of code that can be parallelized = P (=> sequential fraction=1-P),

then for N processors

Max speedup : $$S(N) = \frac{T(1)}{T(N)} \le \frac{1}{(1-P)+(P/N)} \le \frac{1}{1-P}$$

e.g. Fully sequential : P=0.0 => max speedup = 1

50% parallel : P=0.5 => max speedup = 2

100% parallel : P=1.0 => max speedup = infinity

N	P=0.5	P=0.90	P=0.99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10,000	1.99	9.91	99.02

3.2.1 Amdahl's Law (cont)

In early days of parallel computing, it was believed that this would limit the utility of parallel computing to a small number of specialized applications.

However, practical experience revealed that this was an inherently sequential way of thinking and it is of little relevance to real problems, if they are designed with parallel machines in mind from the start.

In general, dealing with serial bottlenecks is a matter of management of the parallel algorithm.

In the solution, some communication costs may be incurred that may limit scalability (or idle time or replicated computation) but serial bottlenecks can be overcome

Amdahl's Law is really only relevant mainly only where a program is parallelized incrementally

- Profile application to find the demanding components
- Adapt these components for parallel application

This partial or incremental parallelization is only effective on small parallel machines. It looks like a "fork and join" => amenable to threads paradigm (e.g. openMP) and usable within small no. of multiprocessors of a node only.

Amdahl's Law can be circumvented by designing complete parallel algorithms and trading communication to mask the serial components.

3.2.1 Amdahl's Law (cont)

Amdahl's Law also misses a number of other things that are important today:

$$S(N) = \frac{T(1)}{T(N)} \le \frac{1}{(1-P)+(P/N)} \le \frac{1}{1-P}$$

- Assumes problem size is fixed

  Many times really it is not "will this job finish faster" but "if you give me more computer, I'll do a bigger problem in same time" -- strong scaling vs weak scaling.

  Serial sections often remain constant in size whilst parallel sections increase with the problem size => serial fraction decreases with the problem size. This is a matter of scalability.

  Gustafson's Law: How much slower is serial on a problem N times bigger?

$$S(N) = \frac{T(1)}{T(N)} \le \frac{(1-P)+(PN)}{(1-P)+(P)} \le \frac{1+P(N-1)}{1}$$

  Serial fraction gets quickly masked and allows infinite speedup!

- Assumes no discontinuities in processor performance curve.

  What about caches? What happens when more processors = less work per processor = less mem needed per proc = eventually fits in cache! Superlinear speedup!

- Assumes only resource varied is processor count.
- Ignores any overhead

$$S(N) = \frac{T(1)}{T(N)} \le \frac{1}{(1-P)+(P/N)+V(/N?)} \le \frac{1}{(1-P)+V} \le \frac{1}{(1-P)}?$$

- Assumes processors are homogeneous – not at all true in many of today's architectures!

  1 fat core (X faster), multiple thin:

$$S(N) = \frac{T(1)}{T(N+1)} \le \frac{1}{(1-P)/X?+(P/N)+V/X?} \text{ or } \frac{1}{(1-P)+(P/N)X?+V}?$$

  Fat does serial? Or parallel slower?

http://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/

3.2.2 Extrapolation from observations

Very often see code performance defined by a single data point which is then extrapolated to other problem sizes and other numbers of processors.

e.g. "We measure speedup of 10.8 on 12 processors with problem size N=100"

Is 10.8 good or bad? Seems ok, right, since the max might be considered to be 12?

But what about the performance at other parameters?

Say the sequential algorithm scales like $N + N^2$

--- a computationally easy part (N) and a computationally intense part (N^2)

Reasonable models of performance speed may then be:

1. $T = N + (N^2 / P)$

   - a parallelization that partitions the intense part and replicates the easy part

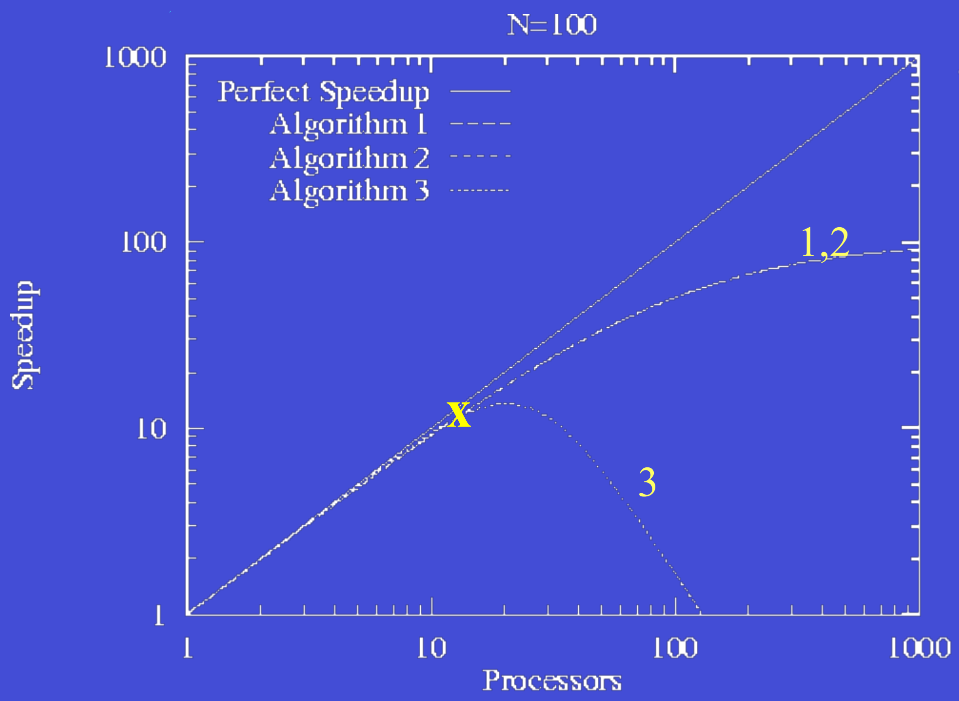
2. $T = (N + N^2)/P + 100$

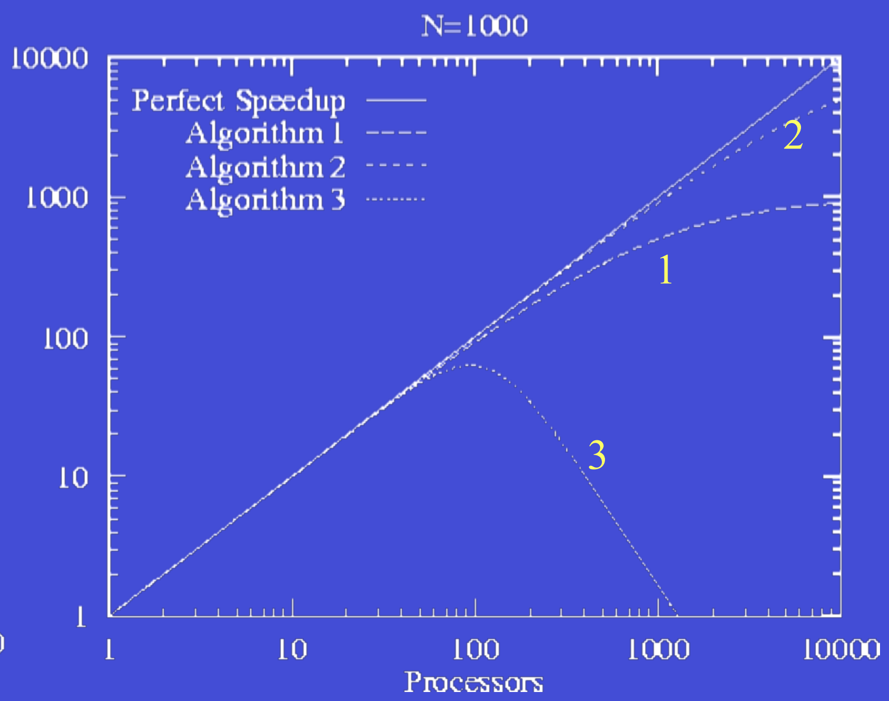
   - a parallelization that partitions all computation but introduces an overhead

3. $T = (N + N^2)/P + 0.6P^2$

   - a parallelization that partitions all computation but introduces an overhead that depends on the partitioning

3.2.2 Extrapolation from observations (cont)





All these algorithms have a speedup of about 10.8 for P=12, N=100

However, very different behaviour at higher N and P:

N=100, high P => all are bad!

N=1000, high P => algorithm 2 the best!

=> ***Need to do scalability analysis for many N and many P!!!!***

3.2.3 Asymptotic analysis

Often see

"asymptotic analysis reveals that the algorithm requires O(NlogN) time on O(N) processors"

Meaning: there exists a constant c and minimum problem size N_0 such that for all N > N_0, the cost(N) < c Nlog(N) on P processors.

This is the cost for large N and large P. It ignores lower order terms that may be important for problem sizes and processor counts of practical interest!

e.g. algorithm may have a cost = 10N + NlogN

Asymptotically (large N) this is O(NlogN)

BUT for N<1024, 10N > NlogN!!

Also, what is the absolute cost?

~ NlogN => = c NlogN

What is C?

Say C=1000

Competing algorithm with cost ~ $10N^2$ is faster for N < 996

Summary of Section 3.2: *all useful concepts but inadequate!*

3.3 Developing better performance models

Desire:

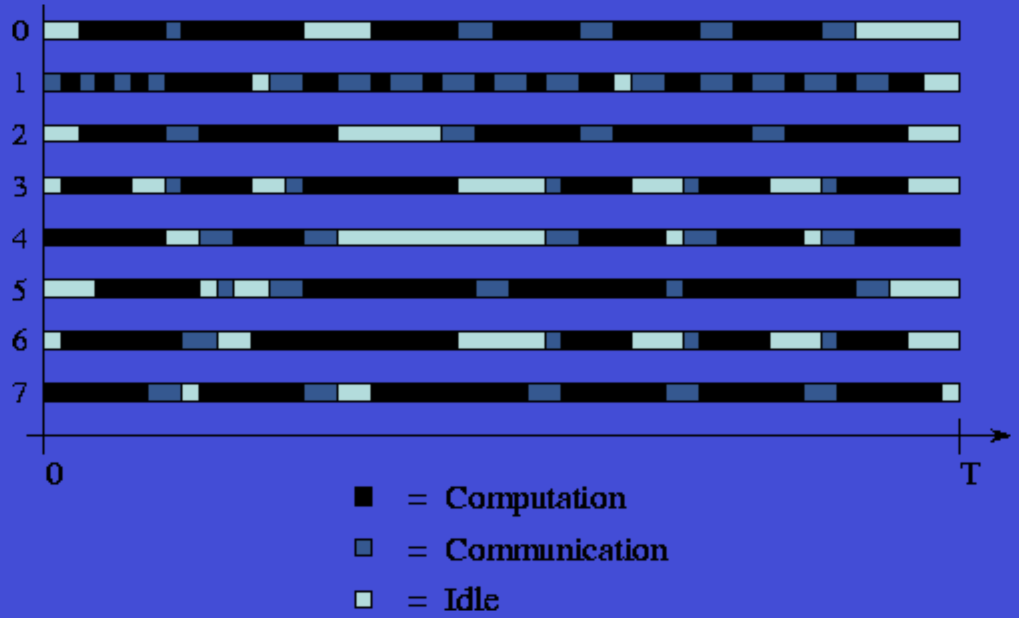
- ✓ Explain available observational timings
- ✓ Predict future performance
- ✓ Abstract out unimportant details

Start with metric = execution time , T (wall clock time) [problems already: e.g. timesharing?]

$T = T(N,P,\ldots)$

Each processor is either

- ✓ Computing
- ✓ Communicating
- ✓ Idling



3.3 Developing better performance models (cont)

Models of total execution time:

1. Pick one processor and hope that is representative (maybe take max?):

$$T = T^j_{comp} + T^j_{comm} + T^j_{idle}$$

2. Average over all processors:

$$\begin{aligned} T &= \frac{1}{P}(T_{comp} + T_{comm} + T_{idle}) \\ &= \frac{1}{P}\left(\sum_{i=0}^{P-1} T^i_{comp} + \sum_{i=0}^{P-1} T^i_{comm} + \sum_{i=0}^{P-1} T^i_{idle}\right) \end{aligned}$$

Take the latter -- easier to determine TOTAL computation/communication than time on individual processors.

We aim for intermediate level of detail in our model:

- specialize to the multicomputer architecture (BUT ignore many hardware details such as memory hierarchy and topology of interconnect).
- Use scale analysis to identify insignificant effects (e.g. ignore initialization if algorithm then does many iterations of a computational step, unless initializing is very costly etc)
- Use empirical studies to calibrate simple models rather than developing more complex models

3.3.1 Execution time

Computation time, T_{comp}:

Depends on the problem size N (or multiple parameters Nx, Ny, Nz etc)

If replicate computations, depends on number of processors, P

Depends on characteristics of the hardware: processor speed, memory system etc => cannot assume total T_{comp} stays constant as P changes (e.g. different P => different use of cache => different total T_{comp}

Communication time, T_{comm}:

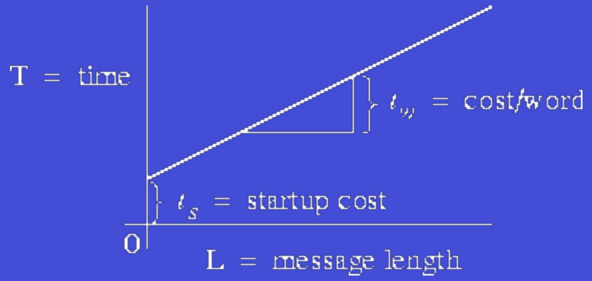
Two distinct types: inter- and intra-processor

Surprisingly, these two are often comparable (unless interconnect slow e.g. Ethernet)

We make this assumption.

Cost of sending a message:

$$T_{msg} = t_s + t_w L$$

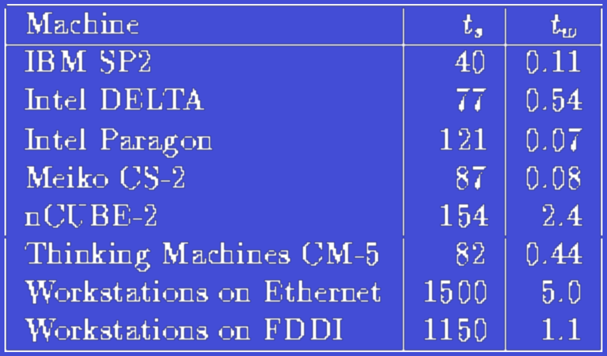


t_s = message startup time = time to initiate communication = LATENCY

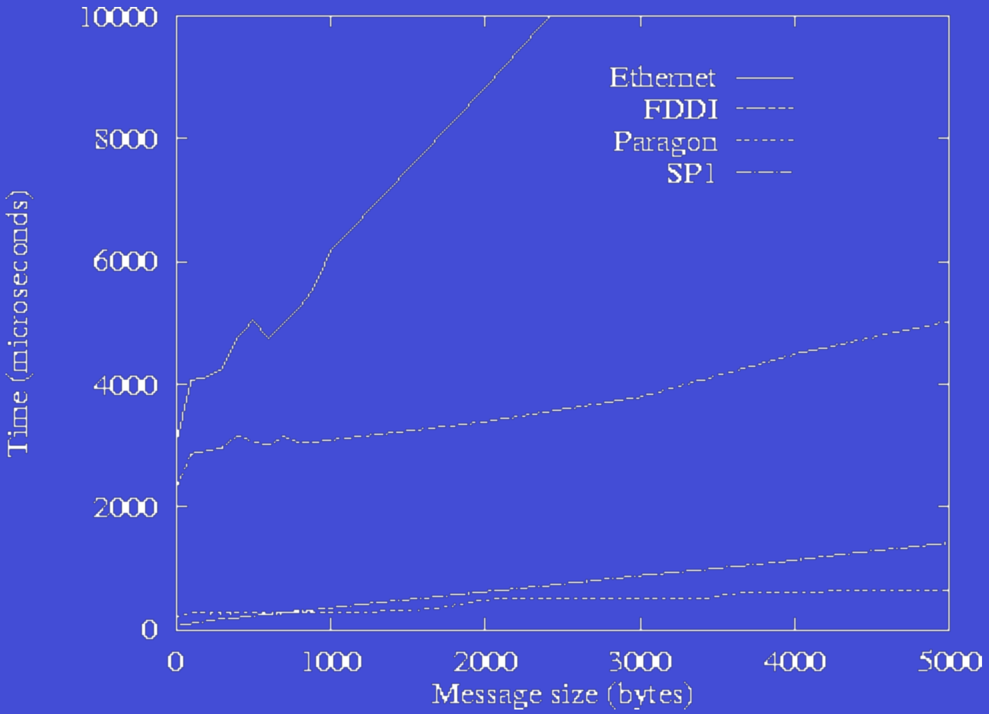
t_w = transfer time per word of data: determined by bandwidth of channel

3.3.1 Execution time (cont)

T_{msg} is highly variable!



Machine	t_s	t_w
IBM SP2	40	0.11
Intel DELTA	77	0.54
Intel Paragon	121	0.07
Meiko CS-2	87	0.08
nCUBE-2	154	2.4
Thinking Machines CM-5	82	0.44
Workstations on Ethernet	1500	5.0
Workstations on FDDI	1150	1.1



Notice:

- Curve matches equation at least for large message sizes
- $t_s > t_w$ => want LARGE messages to mask latency; t_s can dominate for small message sizes
- The values in the table (ping-pong) are generally the BEST achievable!

3.3.1 Execution time (cont)

Idle time, T_{idle}:

Computation and communication time are explicitly specified in the parallel algorithm

Idle time is not => a little more complex. Depends on the ordering of operations.

Processor may be idle due to

1. Lack of computation
2. Lack of data: wait whilst remote data is computed and communicated

To reduce idle time:

For case 1: load-balance

For case 2: overlap computation and communication i.e. perform some computation or communication whilst waiting for remote date:

- Multiple tasks on a processor: when one blocks, compute other. Issue: scheduling costs.
- Interleave communications in amongst computation.

3.3.1 Execution time (cont)

Example: Finite-difference algorithm for atmospheric model (see previous case study)

Grid Nx x Ny x Nz

Assume Nx = Ny = N for simplicity and the 9-point stencil of before

Assume 1-D domain decomposition: partitioned in one horizontal direction (vertical planes) --

P tasks for subgrids N x (N/P) x Nz

No replicated computation => $\boxed{T_{comp} = t_c N^2 N_z}$

where t_c = average computation time per grid point (slightly different at edges from interior etc)

Using a 9 point stencil => each task exchanges 2 planes = 2 N Nz points with each of 2 neighbours (per variable derivative actually)

$\Rightarrow$ $\boxed{T_{comm} = 2P(t_s + 2t_w N N_z)}$

If P divides N exactly, then assume load-balanced and no idle time:

$$\boxed{T_{1D\_finite\_diff} = (T_{comp} + T_{comm})/P}$$

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

3.3.2 Efficiency and speedup

Execution time may not be the best metric. Execution times vary with problem size and therefore should be normalized by the problem size for comparison.

Efficiency -- amount of time processors spend doing useful work -- may provide a more convenient measure.

Characterizes effectiveness with which an algorithm uses resources of a parallel machine in a way that is independent of problem size.

Define ***RELATIVE EFFICIENCY*** as

$$E_{relative} = \frac{T_1}{PT_P}$$

T_1 = execution time on one processor; T_p = execution time on P processors

Related quantity -- ***RELATIVE SPEEDUP***: $S_{relative} = PE = \frac{T_1}{T_P}$ (the factor by which execution time is reduced on P processors)

"Relative" => relative to ***parallel algorithm*** running on one processor. Useful measures for exploring scalability but are not absolute measures:

Algorithm 1: $T_1 = 10{,}000$ $T_{1000} = 20$ (=> $S_{rel} = 500$)

Algorithm 2: $T_1 = 1000$ $T_{1000} = 5$ (=> $S_{rel} = 200$)

Clearly, Algorithm 2 is better on 1000 processors despite S_{rel} information!

Could do with absolute efficiency: use T1 of best uniprocessor algorithm? (we will not distinguish in general between absolute and relative here)

3.3.2 Efficiency and speedup (cont)

e.g. 1D decomposition finite-differences for atmospheric model:

For this algorithm, $$T_1 = t_c N^2 N_z$$

⇒ Efficiency:

$$E = \frac{t_c N^2 N_z}{t_c N^2 N_z + 2Pt_s + 4Pt_w N N_z}$$

3.4 Scalability analysis

We wish to use performance models like those just developed to explore and refine a parallel algorithm design.

We can immediately perform a ***qualitative*** analysis of performance:

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

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

$$E = \frac{t_c N^2 N_z}{t_c N^2 N_z + 2Pt_s + 4Pt_w N N_z}$$

Execution time decreases with increasing P but is bounded below by comm costs

Execution time increases with increasing N, Nz, t_c, t_s, t_w

Efficiency decreases with increasing P, t_s, t_w (due to communication costs)

Efficiency increases with increasing N, Nz, t_c (due to masking of comm costs)

3.4 Scalability analysis (cont)

Simple observations provide interesting insights into algorithm characteristics.

However, not sufficient basis for making design trade-offs. Require ***quantitative*** information for this:

- Need machine specific values for the parameters.
- Get these from empirical studies in general (more later).

Then use models to answer checklist questions:

- ✓ Does algorithm meet design requirements on target machine? For execution time, memory, …)
- ✓ How adaptable is the algorithm?
  - How well does it adapt to different problems sizes, processor counts?
  - How sensitive is the algorithm to t_s, t_w?
- ✓ How does it compare to other algorithms for the same problem? What different execution times can be expected from different algorithms?

Caution: these are of course huge simplifications of complex things (architecture may not be that close to multicomputer etc). Once algorithm implemented, validate models and adjust as necessary. BUT NOT A REASON FOR SKIPPING THIS STEP!

3.4 Scalability analysis (cont)

A quantitative scalability analysis can be performed in two ways:

1. Fixed problem size -- STRONG SCALING
2. Scaled problem size -- WEAK SCALING

3.4.1 Fixed problem size (strong scaling)

In this mode can answer questions like:

How fast/efficiently can I solve a particular problem on a particular computer?

What is the largest number of processors I can use if I want to maintain an efficiency of great than 50%?

It is important to consider both T (execution time) and E (efficiency):

E will generally decrease monotonically with P

T will generally decrease monotonically with P

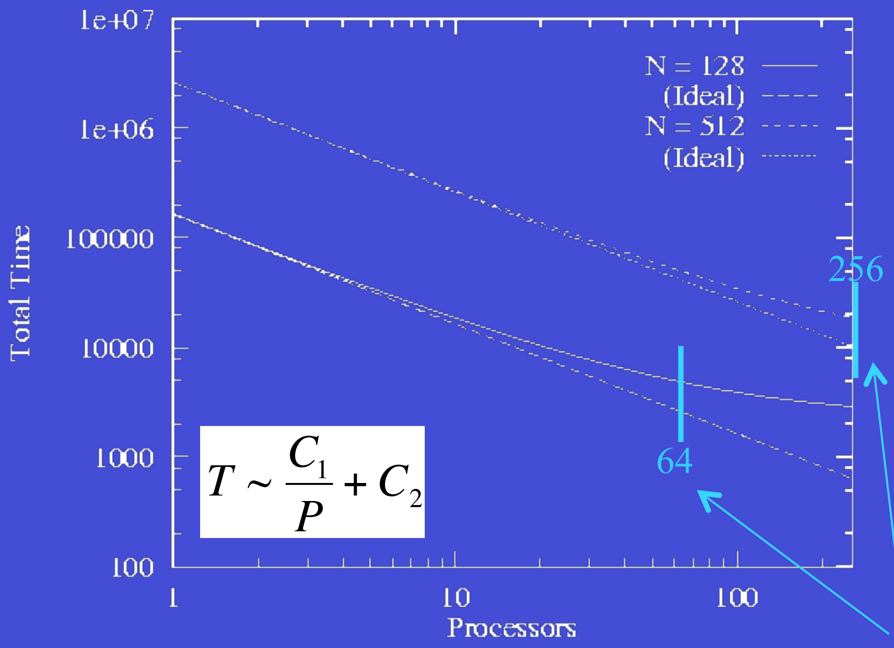
BUT T may actually increase with P if the performance model contains a term proportional to a positive power of P. In such cases, it may not be productive to use more than some maximum number of processors for a particular problem size (and given machine parameters)

3.4 Scalability analysis (cont)

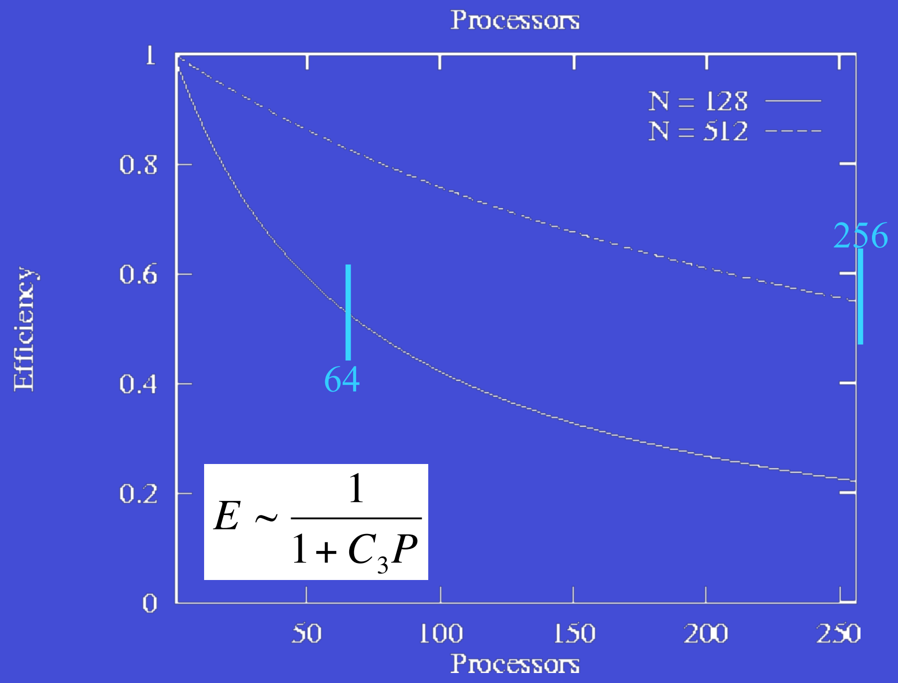
e.g. Fixed problem size analysis for finite-difference

Plots of T and E as a function of P and N (Nz=10)

using machine parameters characteristic of a fine-grained multicomputer: tc = 1 µsec, ts = 100 µsec, tw = 0.4 µsec



$$T \sim \frac{C_1}{P} + C_2$$



$$E \sim \frac{1}{1 + C_3 P}$$

Actual max allowed in decomp

(must be 2 planes per proc min)

For fixed problem size, execution time decrease tails off as get to higher P, as communication costs start to dominate

3.4 Scalability analysis (cont)

3.4.2 Scaled problem size (weak scaling)

In this approach, are not considering solving a fixed problem faster, but rather what happens as go to larger and larger problems.

Consider how must the amount of computation performed scale with P to keep E constant = function of N = ***ISOEFFICIENCY FUNCTION***.

Isoefficiency function ~ O(P) => ***highly scalable***, since amount of computation needs to increase ***only linearly*** with P

Isoefficiency function ~ $O(P^2, \ldots, P^a, a>0)$ or $O(e^P)$ => ***poorly scalable***

Recall
$$E = \frac{T_1}{T_{comp} + T_{comm} + T_{idle}}$$

So E = const, c =>
$$T_1 = c(T_{comp} + T_{comm} + T_{idle})$$

i.e. uniprocessor time must increase at the same rate as the total parallel time, or equivalently, the amount of essential computation must increase at the same rate as the overheads due to replicated computation, communication and idle time.

Note: Scaled problems do not always make sense e.g. weather forecasting, image processing (sizes of computations may actually be fixed)

3.4 Scalability analysis (cont)

e.g. Scaled problem size analysis: isoefficiency of ***1D*** decomposition finite-difference

Recall for 1D decomposition of finite-difference mesh N x N x Nz that

$$E = \frac{t_c N^2 N_z}{t_c N^2 N_z + 2Pt_s + 4Pt_w N N_z}$$

Therefore, for constant E, require

$$t_c N^2 N_z = E(t_c N^2 N_z + 2t_s P + 4t_w N N_z P)$$

Dominant terms
(N, P large)

$$\sim N^2 N_z \qquad \sim N^2 N_z \qquad \sim N N_z P$$

Satisfied by N=P

$$t_c N_z = E(t_c N_z + \frac{2t_s}{P} + 4t_w N_z)$$

(except when P is small)

Scaling N ~ P => no. of grid points to keep E constant ~ P^2

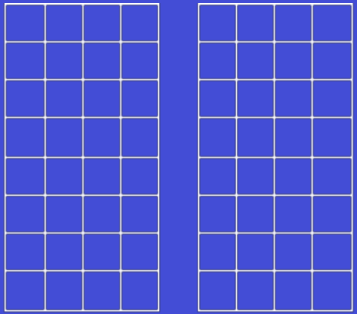
$\Rightarrow$ amount of computation to keep E constant ~ P^2

$\Rightarrow$ isoefficiency of this algorithm ~ $O(P^2)$

3.4 Scalability analysis (cont)

⇒ Amount of computation must increase as the square of the number of processors to keep the efficiency constant. Pretty obvious really:

A

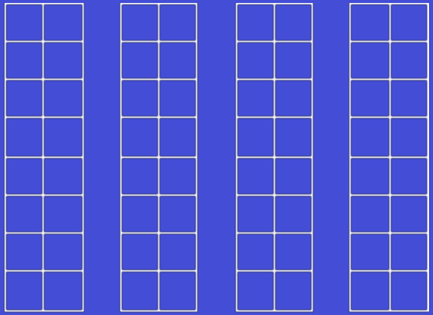


N=8 (8x8) P=2:

Each task has 32 grid points to compute

Must communicate with two neighbours

B



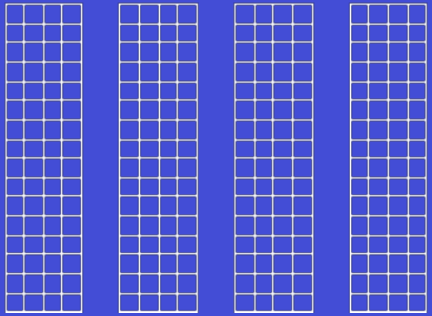
Double P, same N: N=8 (8x8) P=4:

Computation costs same as original

Communication costs double original

=> Efficiency (= ratio) reduced

C



Double P, double N: N=16 (16x16) P=4:

Computation costs 4x original

Communication costs 4x original

=> Efficiency same as original

RECAP from last Thurs

Quantitative basis for design

Desire: improve “performance”; Definition of “performance” ; Create predictive models

Three somewhat helpful but also dangerous ideas:

- Amdahl’s law
- Extrapolation from a small number of observations
- Asymptotic analysis

Developing better models:

- $T_{exec}=T_{comp} + T_{idle} + T_{comm}$
- $T_{msg} = T_s + T_w L$
- Efficiency $E = T_1/PT_p$
- Speedup $S = T_1/T_p$
- Scalability (with P and N) analysis of T_{exec}, E, S:
  - Strong and weak scaling

3.4 Scalability analysis (cont)

CLASS CASE STUDY:

Do scaled problem size analysis for isoefficiency of ***2D decomposition*** finite-difference

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 fractional 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)

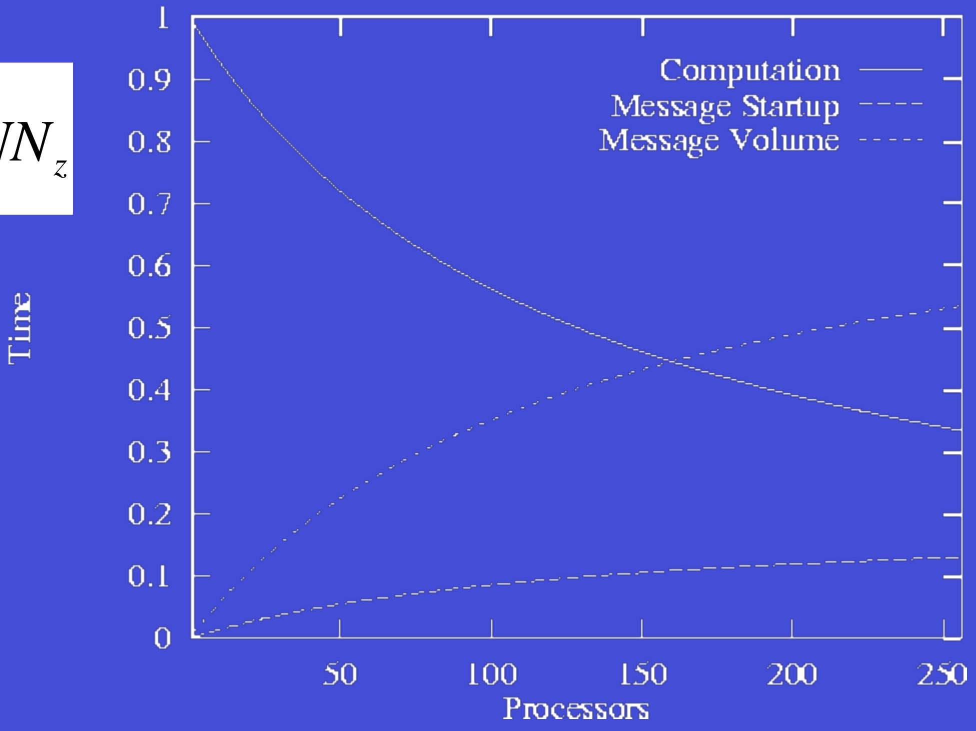
N=512, Nz=1, ts=200μsec, tw=0.8μsec, tc=1μsec

Varying P, immediately see:

- ✓ For large P, dominated by data transfer
- ✓ For large P, message startup not insignificant

If we increased Nz, startup costs decrease and efficiency improves

Fractional contributions to overall execution time



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 no.s 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! (randomize?)

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 $T_{msg} = t_s + t_w L$

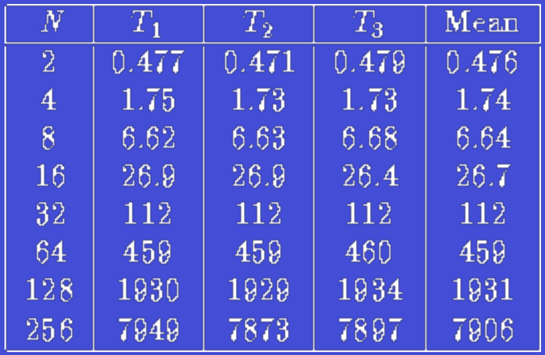
Obvious methods:

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

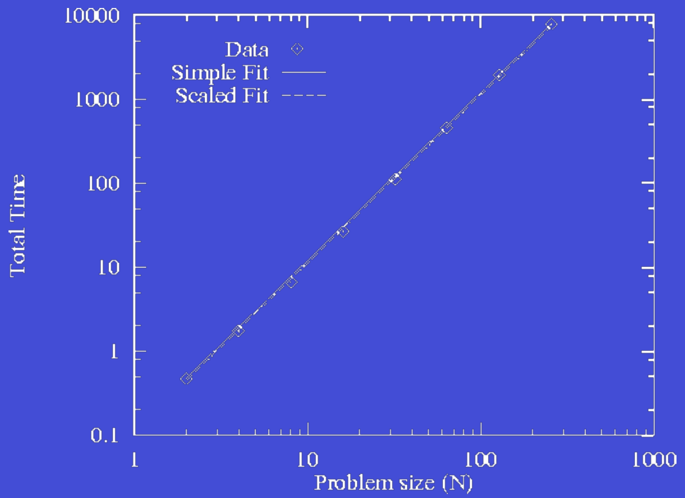
e.g. 1D finite-difference

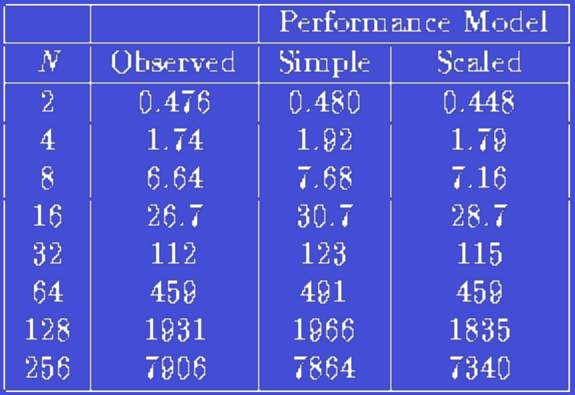
Determine t_c from $t_{comp} = t_c N^2 Nz$

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



N	T_1	T_2	T_3	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





		Performance Model	
N	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 tc=0.0120msec

Scaled LS t_c=0.0112msec

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 behavior:
   - ✓ Time in initialization
   - ✓ 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 parallelize something important! Did we assume falsely that replicating something might be cheaper? 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. we tried to overlap comp/comm by introducing many tasks per processor but this may not work well if library does establish 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 => total computation time T_{comp} will decrease. If this more than compensates for the extra overheads of more processors (T_{comm}, T_{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.

Parallel algorithm anomalies: e.g. Parallel branch-and-bound searchalgorithm is 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:

(Show speedup rather than raw execution time to make results clearer for larger P)

Model predicts continuous speedup with P

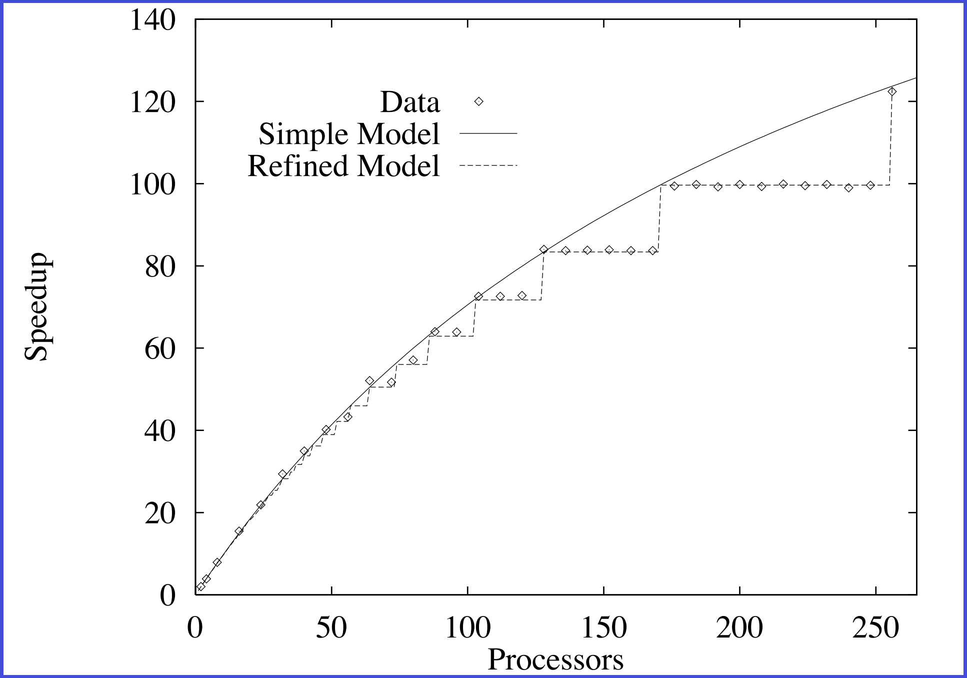
Observations show stepwise speedup

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:

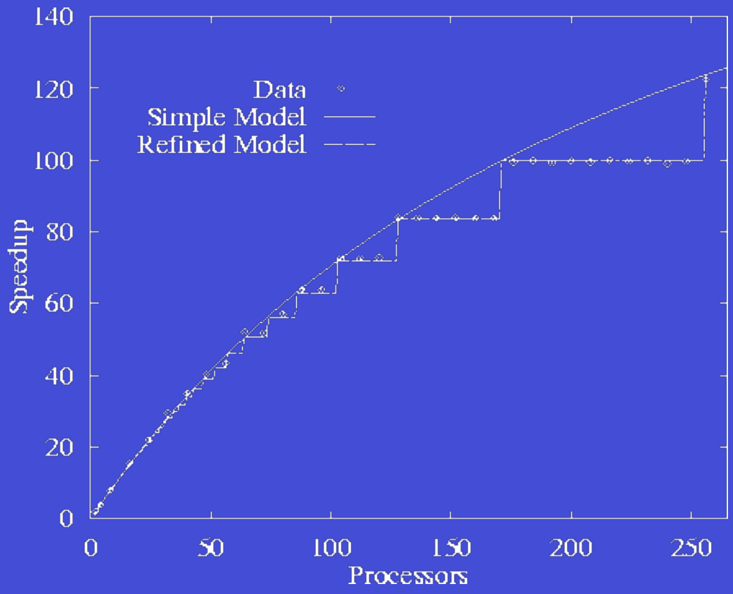
$$T_{comp\_MAX} = ceiling(N/P)NN_z$$

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

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

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

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!

One big issue, as mentioned earlier, it is entirely possible that our simple communication model based on $T_{msg} = t_s + t_w L$ 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. $T_{msg} = t_s + t_w L S$

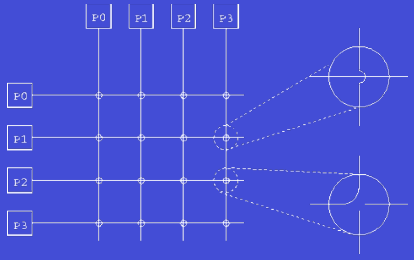
The effective bandwidth available to each processor is 1/S of the true bandwidth.

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

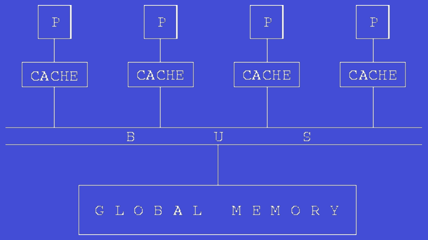
3.7 A refined communication cost model (cont)

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

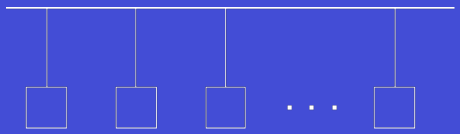


Bus-based networks: Processors share the bus => S = no. of procs trying to communicate. Highly non-scalable! Common in SMP machines -- simplify programming by removing locality. Requires caches to avoid bus clashes => reintroduces locality!



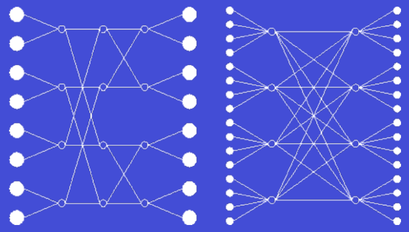
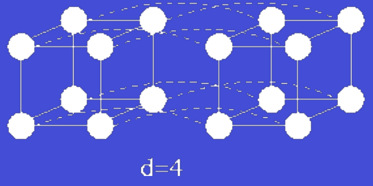
Ethernet: = slow bus-based network! => S= no. of procs sending simultaneously.

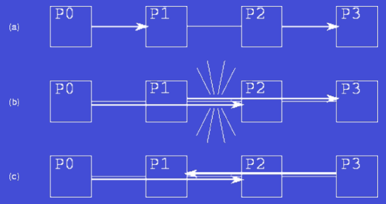
Mesh networks: Same as a cross-bar switch except the switching nodes are the processors themselves (in the mesh rather than at the edges). For mesh of dimension D, each processors is connect to 2D neighbours. Extend with toroidal connections. S depends on algorithm (more later).

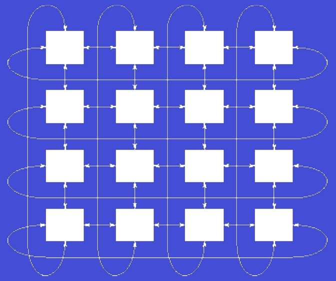


Hypercube: Dimension D connects to 2^D "neighbours". Like a mesh plus long-wire connections. Complex. Cannot be laid out in 3D so that all are real neighbours.

Multistage interconnect networks (MIN): Like crossbar, switching elements are distinct from 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 $T_{\text{comm ideal}} = 2t_s + t_w\ 4\ N\ N_z S$

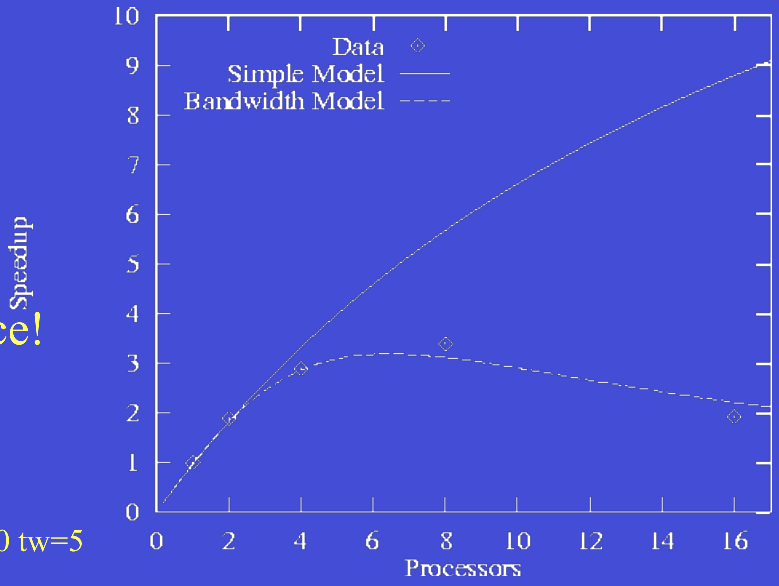
Mesh or in a hypercube: No competition in these networks since the one-dimensional ring communication structure embeds in these networks using only nearest-neighbour connections => S=1

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

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

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 each do log(P) exchanges of N/P data and therefore, in absence of competition, the per-processor communication

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

Crossbar: no competition

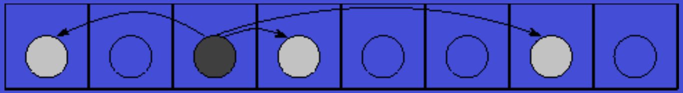
P-processor hypercube: no competition (can organise s.t. nearest neighbour comm only)

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,…,$2^{\log P-1}$ hops along the mesh at the various steps of the butterfly

$\Rightarrow$ Total no. of hops done in process =

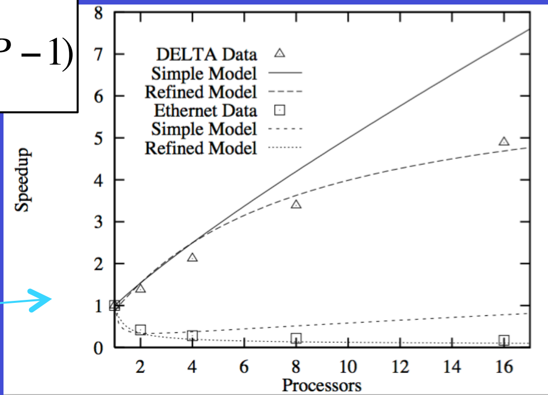
$$P\sum_{i=0}^{\log P-1} 2^i = P(P-1)$$

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

$\Rightarrow$ Min # steps to complete = P/2 NOT log(P) due to conflicts

$\Rightarrow$ S=P/2

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



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 characterize

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