Scientific Parallel Computing: a short course

Valpariaso, January 2011

L. Ridgway Scott

The Institute for Biophysical Dynamics, the Computation Institute, and the Departments of Computer Science and Mathematics, The University of Chicago

These lectures are based on the book Scientific Parallel Computing, by L. Ridgway Scott, Terry Clark, and Babak Bagheri [16].

1 Top 500 supercomputers

The *Top 500* web page tracks the evolution of supercomputing worldwide.

It has a database of the 500 most powerful supercomputers around the world over the last two decades.

The five hundred most powerful supercomputers in the world have been parallel computers for about two decades.

Moreover, the number of processors (cores) per system continues to increase.

The following figure shows the distribution of systems by size (number of processors) for the dates June 2005 (left curve, in blue) and June 2010 (right curve, in red).

This clearly shows a trend towards utilization of larger numbers of processors.

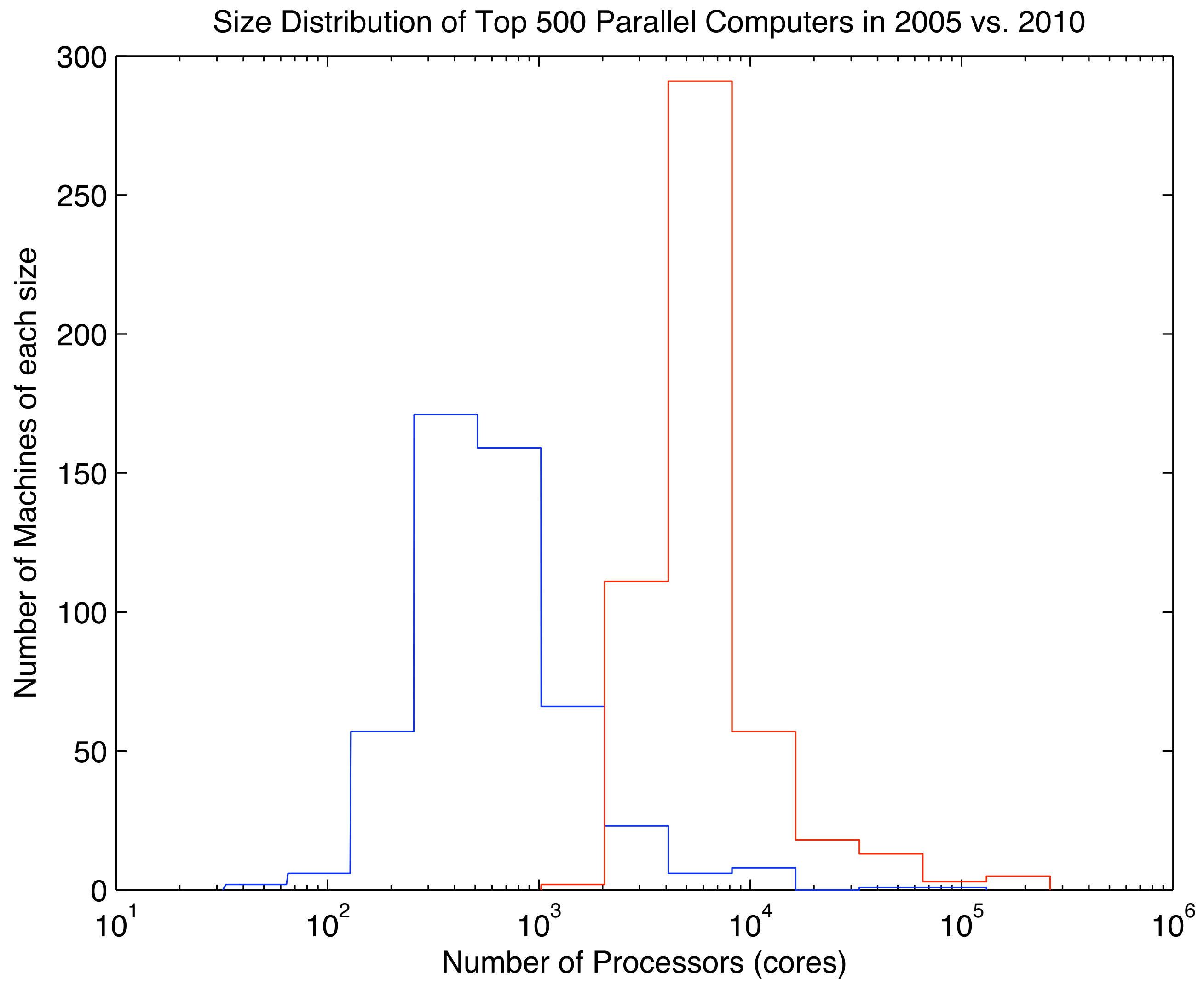


Figure 1: Distribution of systems by size (number of processors) for the dates June 2005 (left curve, in blue) and June 2010 (right curve, in red).

1.1 Top 500 summary

In 2010, all of the top 500 supercomputers had more than 1K processors.

In 2005, only a few supercomputers had more than 1K processors.

In 2005, the majority of the most powerful supercomputers had between 256 and 1K processors.

In 2010, the majority of such supercomputers had between 4K and 8K processors.

In 2010, there were five supercomputers with more than 128K processors.

In 2005, only two supercomputers had more than 32K processors.

Table 1: Parallel performance of U-cycle multigrid for Poisson's equation on IBM BG/L; IT$_j$ denotes average number of iterations of PSOR with relaxation parameter ω for solving the coarsest grid equations. Time in seconds.

P	h_l	$E(n,P)$	Total	Comm.	IT$_U$	On Ω_j			
		$(n=2^{10})$	Time	Time		h_j	IT$_j$	Time	ω
1	$1/2^{10}$	1.0	3.8375	0	11	1/8	6	0.0021	1.52
4	$1/2^{11}$	0.9841	3.8994	0.2217	11	1/16	10	0.0073	1.66
16	$1/2^{12}$	1.0687	3.5908	0.2209	10	1/32	64	0.5282	1.88
64	$1/2^{13}$	0.9008	4.2601	0.3424	10	1/64	105	0.1040	1.91
256	$1/2^{14}$	0.8782	4.3697	0.3648	10	1/128	178	0.1965	1.96
1024	$1/2^{15}$	0.9274	4.1379	0.3962	9	1/256	332	0.3674	1.98

Parallel U-cycle multigrid took only about 4 seconds in solving a linear system with 2^{30} unknowns on 1024 processors of the IBM BG/L.

Scaled speedup for $P = 1024$ is about 950 [4].

1.2 HPC application: employment

Post Doctoral Fellow in Parallel Scalable Algorithms and Software

This position is part of a research program to extend the Uintah software (www.uintah.utah.edu) as applied to challenging problems in combustion and energy to petascale and beyond using novel algorithmic and software approaches. Uintah is a scalable, highly parallel framework designed to simulate complex fluid-structure interaction problems, with a clear partition between the parallel framework and the numerical algorithms.

The postdoc will be expected to perform algorithmic and parallel computing research to help Uintah scale to beyond 98K cores. The work is projected to last for three years and will involve research, analysis, and continuing development of scalable adaptive algorithms in the areas such as adaptive meshing and linear solvers in the context of combustion and energy-related problems combined with implementation and scaling studies using the Uintah code base.

The requirement is for an individual with highly developed algorithmic and C++ programming skills who must have experience in developing parallel algorithms and software. Familiarity with adaptive grid and/or linear solver techniques is desirable. Please contact Martin Berzins (mb@sci.utah.edu) for further information. [26 Dec 2010]

2 What is Parallel Computing?

Application of two or more processing units to solve a single problem.

Practical objective: scale to thousands of processors.

Requires some form of *synchronization* so that the value in memory is accessed when it is valid.

Can be achieved in various ways: messages, semaphores.

Different languages use different approaches.

Different architectures support different techniques.

Ultimately, parallel algorithms are the central issue.

Performance analysis supports development process.

The secrets to parallel computing

Figure 2: Depiction of a critical section, cover of Scientific Parallel Computing, by Scott, Clark and Bagheri, Princeton Univ. Press, 2005 [16].

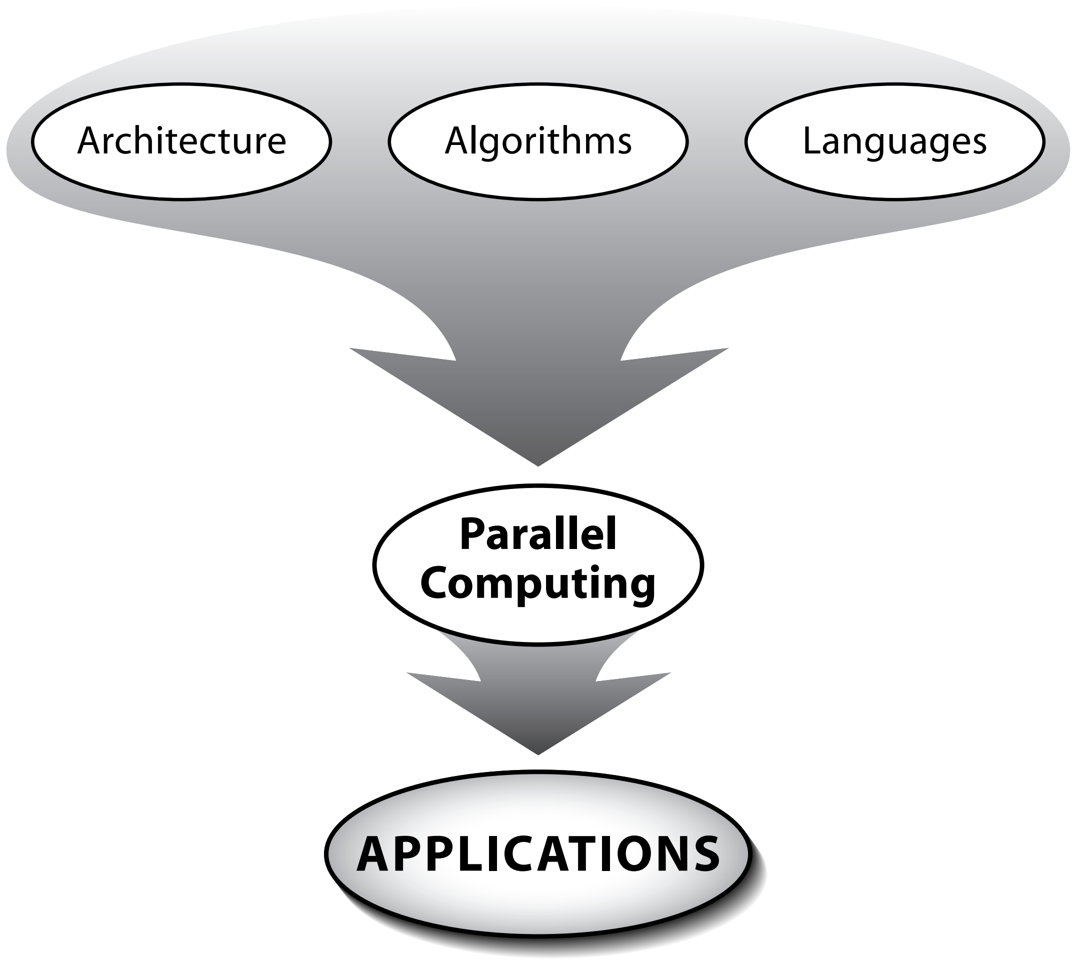


Figure 3: Knowledge of algorithms, architecture, and languages contributes to effective use of parallel computers in practical applications.

3 Plan for the course

Basic notions relating to performance

- Influence of memory and load balancing

Performance measures

- Speedup, Efficiency, Amdhahl's Law, and Scalability

PMS notation and computer architecture

- Influence of memory systems on performance

Dependence analysis (loop-carried dependences)

Basic algorithms: linear systems

- Scalable algorithms for sparse triangular systems

Novel algorithms: parallelization in time

- Scalable algorithms for solving ODE's

4 Performance

In scientific computing, performance is a constraint, not an objective.

Definition 4.1 *A Cray*[a] *is* 10^9 *floating-point operations per second (one floating-point operation per nanosecond), the level of performance being achieved by a single processor at Seymour Cray's untimely death.*

[a]The development of the first "supercomputers" was largely the result of efforts lead by Seymour Cray (1925-1996). Seymour Cray earned a BS in engineering from the University of Minnesota in 1950. He co-founded Control Data Corporation (CDC) in 1957; the CDC 6600 is the primary candidate for the title of "first supercomputer." The series of supercomputers bearing Cray's name were produced by Cray Research. Started in 1972, this company was headquartered in Seymour's boyhood home, Chippewa Falls, Wisconsin, also home of the Jacob Leinenkugel Brewing Co.

4.1 Memory effects

Memory access is the critical issue in high-performance computing.

Definition 4.2 *The* **work/memory ratio** ρ_{WM}*: number of floating-point operations divided by number of memory locations referenced (either reads or writes).*

A look at a book of mathematical tables tells us that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \cdots \tag{4.1}$$

Slowly converging series good example for studying basic operation of computing the sum of a series of numbers:

$$A = \sum_{i=1}^{N} a_i . \tag{4.2}$$

Computation of A in equation (4.2) requires $N - 1$ floating-point additions and involves $N + 1$ memory locations: one for A and n for the a_i's.

Therefore, work/memory ratio for this algorithm is $\rho_{\mathrm{WM}} = (N-1)/(N+1) \approx 1$ for large N.

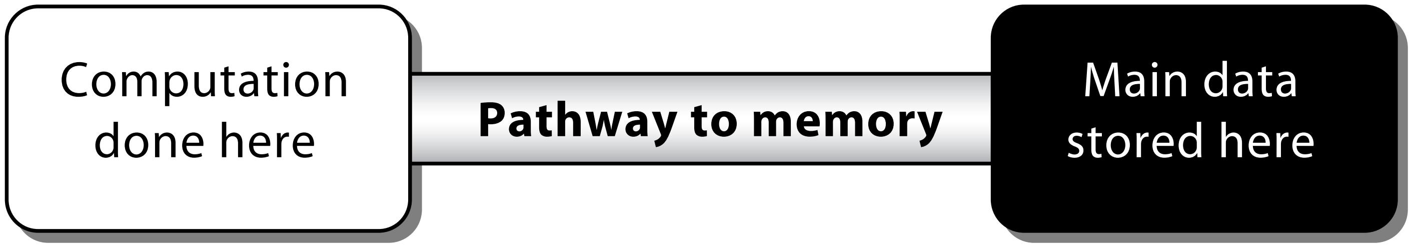


Figure 4: A simple memory model with a computational unit with only a small amount of local memory (not shown) separated from the main memory by a pathway with limited bandwidth μ.

Theorem 4.1 *Suppose that a given algorithm has a work/memory ratio* ρ_{WM}*, and it is implemented on a system as depicted in Figure 4 with a maximum bandwidth to memory of* μ *billion floating-point words per second. Then the maximum performance that can be achieved is* $\mu\rho_{\mathrm{WM}}$ *GFLOPS.*

Theorem 4.1 provides an upper bound on the number of operations per unit time, by assuming the floating-point operation blocks until data are available to the cpu.

Therefore the cpu cannot proceed faster than the rate data are supplied, and it might proceed slower.

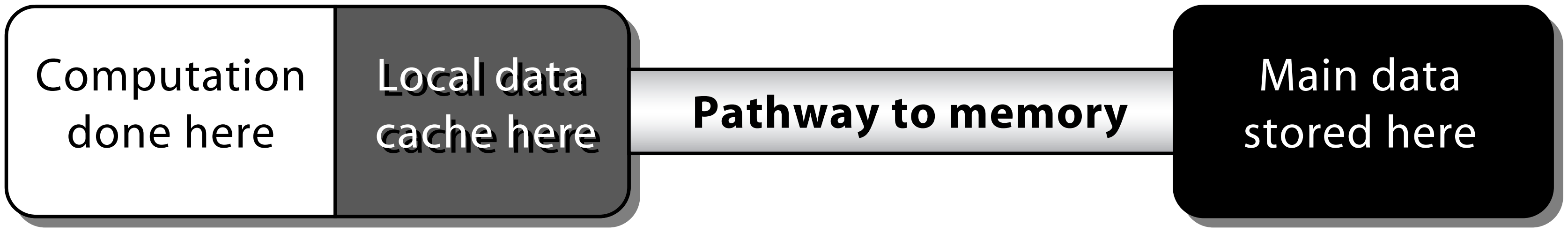


Figure 5: A memory model with a large local data cache separated from the main memory by a pathway with limited bandwidth μ.

The performance of a two-level memory model (as depicted in Figure 5) consisting of a cache and a main memory can be modeled simplistically as

$$\begin{aligned}\frac{\text{average cycles}}{\text{word access}} =& \%\text{hits} \times \frac{\text{cache cycles}}{\text{word access}} \\ &+ (1 - \%\text{hits}) \times \frac{\text{main memory cycles}}{\text{word access}},\end{aligned} \tag{4.3}$$

where %hits is the fraction of cache hits among all memory references.

Figure 6 indicates the performance of a hypothetical application, depicting a decrease in performance as a problem increases in size and migrates into ever slower memory systems. Eventually the problem size reaches a point where it can not ever be completed for lack of memory.

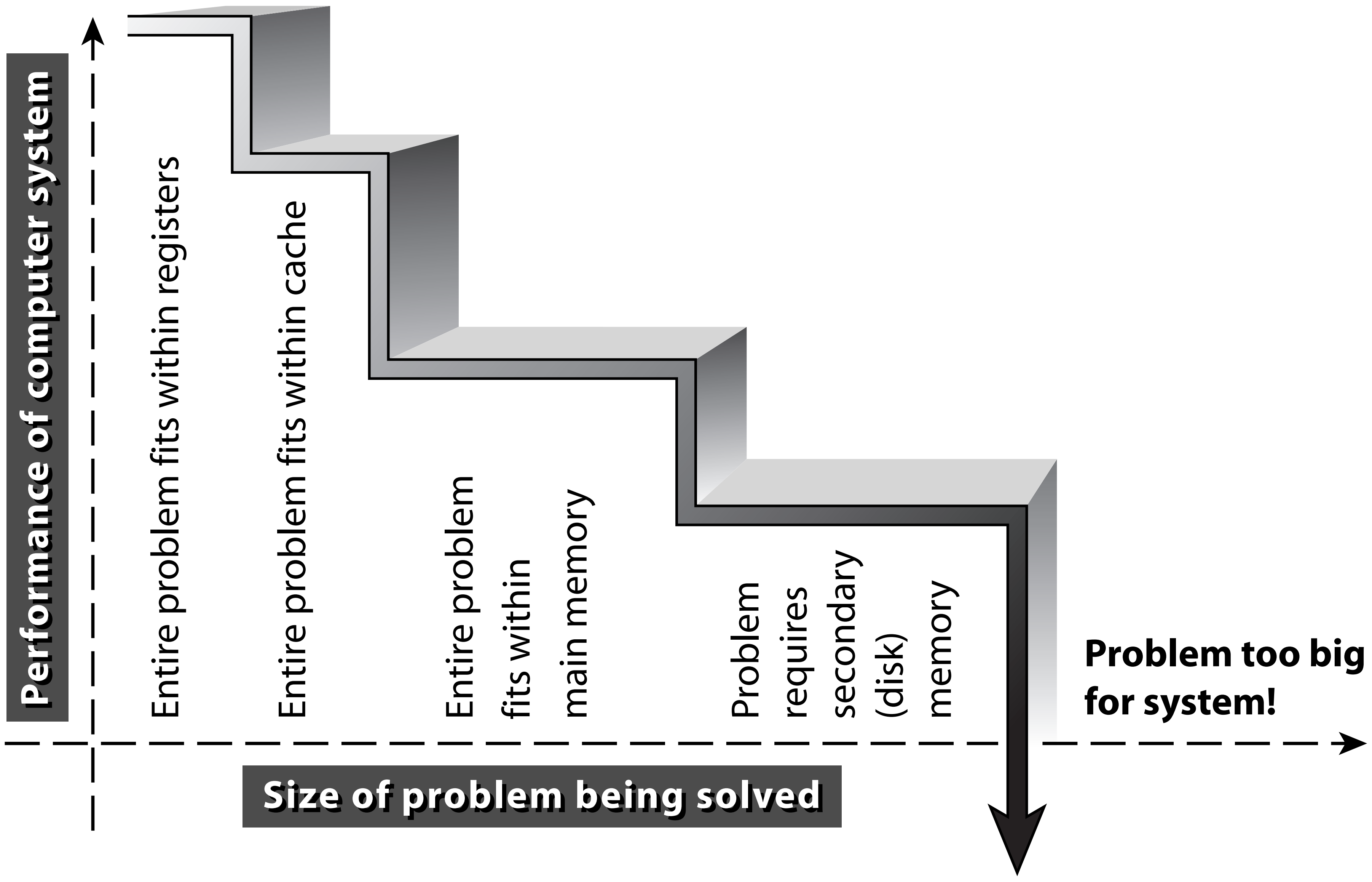

Performance of computer system
Entire problem fits within registers
Entire problem fits within cache
Entire problem fits within main memory
Problem requires secondary (disk) memory
Problem too big for system!
Size of problem being solved

Figure 6: Hypothetical model of performance of a computer having a hierarchy of memory systems (registers, cache, main memory, and disk).

4.2 Simple sums

We begin with the summation problem (4.2):

$$A = \sum_{i=1}^{N} a_i.$$

Assume for simplicity that N is an integer multiple, k, of P: $N = k \cdot P$. Then we can divide the reduction operation into P partial sums:

$$A_j = \sum_{i=(j-1)k+1}^{jk} a_i \tag{4.4}$$

for $j = 1, \ldots, P$. Then

$$A = \sum_{i=1}^{P} A_i. \tag{4.5}$$

Leaving aside the last step (4.5), we have managed to create P parallel *tasks* (4.4) each having $k = N/P$ additions to do on $k = N/P$ data points.

Definition 4.3 *The* **iteration space** *of a given set of (possibly nested) loops is a subset of the Cartesian product of the integers consisting of the set of all possible values of loop indices. The dimension of the Cartesian product is the number of nested loops (it is one if there is only one loop). When the exact set of loop indices is not known without running the code, the iteration space is taken to be the smallest set known to contain all of the loop indices.*

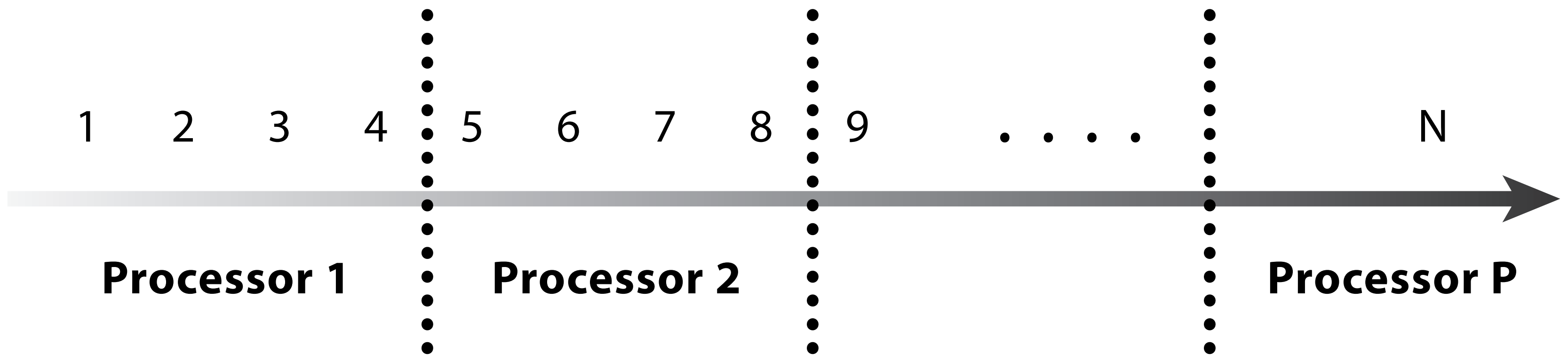


Figure 7: The iteration space for the summation problem with a simple *decomposition* indicated by dotted lines for a *granularity* of $k = 4$.

The algorithm (4.2) has been parallelized using an approach referred to as *data parallelism*.

4.3 Load balancing

Figure 7 depicts graphically the iteration space for the summation problem (4.2) parallelized using the decomposition in (4.4).

If the work is not distributed equally, then one processor may end up taking longer than the others.

Definition 4.4 *Suppose that a set of parallel tasks (indexed by $i = 1, \ldots, P$) execute in an amount of time t_i. Define the average execution time*

$$\text{ave}\{t_i \;:\; 1 \le i \le P\} := \frac{1}{P} \sum_{1 \le i \le P} t_i. \tag{4.6}$$

The **load balance** β *of this set of parallel tasks is*

$$\beta := \frac{\text{ave}\{t_i \;:\; 1 \le i \le P\}}{\max\{t_i \;:\; 1 \le i \le P\}}. \tag{4.7}$$

A set of tasks is said to be **load balanced** *if β is close to one.*

4.4 About the definition

The amount the load balance β differs from the ideal case $\beta = 1$ measures the relative difference between the longest task and the average task:

$$1 - \beta = \frac{\max\{t_i \; : \; 1 \le i \le P\} - \text{ave}\,\{t_i \; : \; 1 \le i \le P\}}{\max\{t_i \; : \; 1 \le i \le P\}}. \tag{4.8}$$

Note that we have compared the *average* time with the maximum time, not the minimum time.

The relevance of this will become clearer in Section 4.5 and Section 7.5.

Have defined load balance in terms of time of execution instead of amount of computational work to be done.

Performance (Definition 4.1) need not be the same for different tasks, and the cost of the computation is proportional to the time it takes, not to how many floating-point operations get done.

Often try to achieve load balance by balancing the amount of work to be done, but dynamic load balancing is often needed.

4.5 Minimum run time is not relevant for load balance

Optimal run time can be achieved when one processor does nothing.

Suppose P processors are used to sum $n = (P-1)k+1$ numbers.

All but one processor does $k-1$ floating point operations involving k summands.

When $k \leq P$, the fastest possible execution time involves at least one processor doing $k-1$ floating point operations with k summands.

Proof: If less than k summands are assigned to all processors, then at most $(k-1)P$ summands would have been assigned. This means that the number of unassigned summands is at least

$$k(P-1)+1-(k-1)P = P-k+1.$$

Since $k \leq P$, at least one summand is left out, and this would not be a valid algorithm.

Therefore, the fastest possible execution time involves at least one processor doing $k-1$ floating point operations with k summands.

5 Limits to Parallel Performance

Parallel performance is more complex than sequential performance in many ways: first and foremost, it is more difficult to achieve good parallel performance, but it is also more difficult to predict and even to measure.

This is due largely to the fact that essentially independent computers are cooperating on a single task.

With simple (low-performance) sequential processors, it has been possible to predict sequential performance based on a textual analysis of code or a mathematical description of an algorithm, by counting the number of basic operations.

With parallel computation, other factors become critical, such as

- synchronization,
- load balance, and
- communication costs.

5.1 Limits to Parallel Performance: outline

There are significant limits to utilizing parallel computers efficiently.

Perhaps the most well known *caveat* is Amdahl's Law.

This is a statement about the potential *speed-up* achievable with a number of parallel processors.

Based on the notion of speed-up, one can define a notion of *parallel efficiency*.

The basic notion of speed-up as addressed in Amdahl's law relates primarily to problems of fixed size.

In many cases, one deals with problems of varying size, characterized by some parameter usually related to the size of the data set.

In this case, it may be appropriate to use large computers only for large problems.

The notion of *scalability* relates to whether a given algorithm or code can be used efficiently on a range of problem sizes when using an appropriately chosen number of processors.

5.2 Summation example

Consider the simple parallel summation algorithm to compute $\sum_{i=1}^{N} a_i$.

Each processor computes a partial sum

$$A_j = \sum_{i=s_j}^{e_j} a_i.$$

Consider a collection of individual workstations or personal computers connected by some broadcast medium like Ethernet.

Time of execution on each processor separately should be proportional to N/P assuming that N is divisible by P.

The constant of proportionality will depend on properties of the individual processors and their own memory systems (which we assume are large enough to hold the required data).

For the sake of simplicity, we will normalize so that our unit of time is such that this constant of proportionality is one. Thus the time of execution is exactly N/P.

If communication is done on a single broadcast network, e.g., Ethernet connecting several workstations, then communication is in conflict.

Every time a message is sent, everyone must listen to it even though it may not be for them.

Assume that the algorithm is just for each processor in turn to broadcast their value A_i and have every processor compute $A_1 + \cdots + A_P$ separately.

This would mean that the amount of time to complete the communication would be proportional to P since there is one piece of data per processor to transfer.

The constant of proportionality γ can be interpreted as the the number of floating point additions that can be done in the time it takes to send one word.

Then our time estimate for the total time is

$$T_P = \frac{N}{P} + (\gamma + 1)P, \tag{5.9}$$

including the time to sum the A_i's locally.

In Figure 8, we plot the execution time as a function of the number of processors used with $\gamma = 9$ and $N = 10,000$. Note that (5.9) implies that T_P will eventually start to *increase* for P sufficiently large.

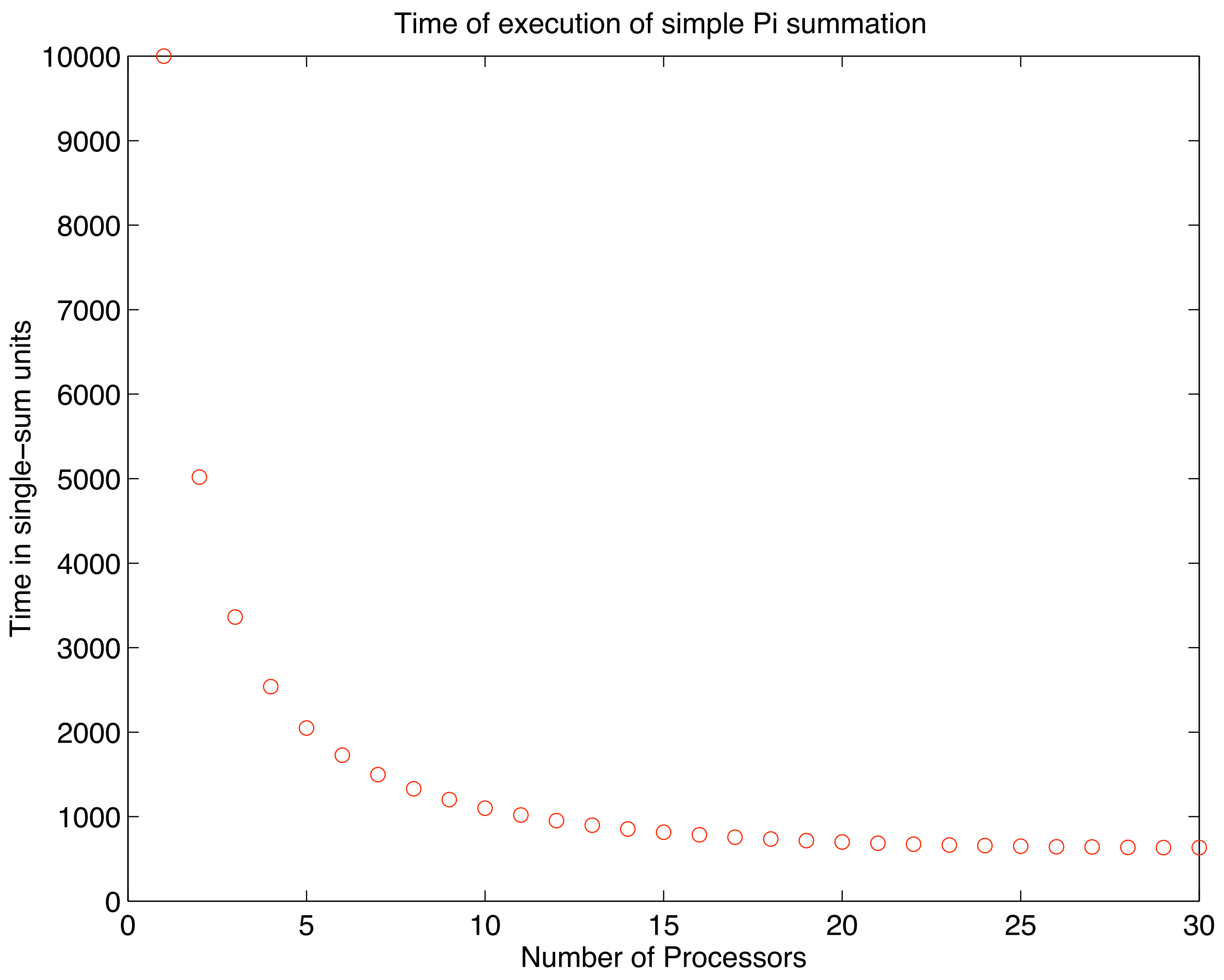


Figure 8: Performance of the summation problem (in arbitrary time units) as a function of the number of processors used with $\gamma = 9$ and $N = 10,000$.

6 Performance measures

This plot in Figure 8 is very forgiving; gives the illusion of positive achievement as long as it decreases.

A more critical way to evaluate the behavior of our parallelization is to consider the *speed-up* that has been achieved.

6.1 Speed-up

Definition 6.1 *The* **speed-up**, S_P, *using* P *processors is defined to be the ratio of the sequential and parallel execution times*

$$S_P = \frac{T_1}{T_P}. \tag{6.10}$$

More precisely, T_1 *should be the time of execution of the best available sequential solution of the problem, and* T_P *should be the time of execution of the parallel solution of the problem on* P *processors.*

The speed-up for the performance in (5.9) is

$$S_P = \frac{N}{\frac{N}{P} + (\gamma+1)P} = \frac{P}{1+(\gamma+1)P^2/N}. \tag{6.11}$$

We refer to the case $S_P \approx P$ as **linear speed-up** or **perfect speed-up**.

In the very fortunate case that $S_P > P$ we call it **super-linear speed-up**, or perhaps more grammatically correctly, **supra-linear speed-up**.

In practice, S_P can sometimes exceed P due to the effective improvement of individual processor performance due to decreasing local data size.

Figure 9 shows speed-up plot corresponding to the execution times in Figure 8.

This is a more demanding view of performance as it emphasizes the deviation from the ideal linear speed-up indicated by the straight line of slope one.

In Figure 8, we saw something bad (execution time) decreasing as a function of the number of processors, so that gave us a positive feeling.

But now we see something good (speed-up) beginning to flag, so we begin to be more skeptical.

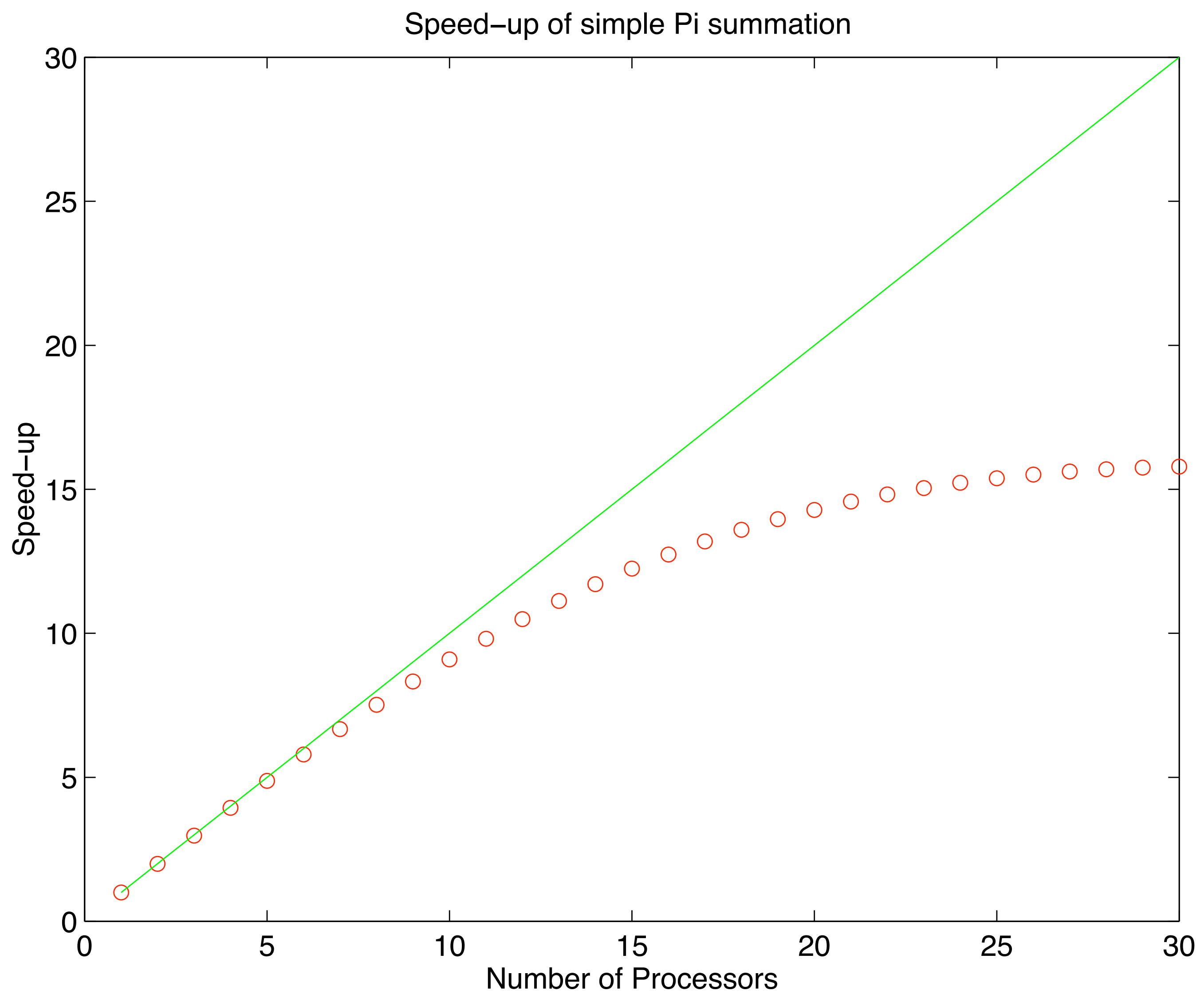


Figure 9: Hypothetical performance of a parallel implementation of summation: speed-up.

Often, S_P will be significantly less than P, and there is no guarantee that it will be greater than one!

If $S_P < 1$, it should be called "slow-down" not speed-up.

S_P will not increase indefinitely in all cases. More likely, as in the example (5.9), it will reach a maximum and then decrease.

Since S_P is inversely proportional to the execution time T_P, the maximum of S_P will occur at the minimum of T_P.

In some cases, it may not be possible to run a problem on one just processor due to memory or time constraints.

In this case, we may want instead to define speed-up with respect to the smallest number P' of processors for which the problem will run. Define

$$S_P^{P'} = \frac{T_{P'}}{T_P} \tag{6.12}$$

to be the corresponding **relative speed-up** in this case for $P \geq P'$. We expect $S_P^{P'} \approx P/P'$ in the ideal case.

6.2 Parallel efficiency

The most demanding view of performance data compares observed speed-up with ideal speed-up, obtaining a measure of *efficiency*.

Definition 6.2 *The* **parallel efficiency**, E_P, *using* P *processors is defined to be the ratio of the speed-up and* P, *that is*

$$E_P := \frac{S_P}{P} = \frac{T_1}{PT_P}, \tag{6.13}$$

where T_1 *is time of execution of best sequential solution of the problem, and* T_P *is time of execution of the parallel solution with* P *processors.*

Figure 10 shows parallel efficiency for performance data in Figure 8.

Now something good (efficiency) is going down; now we are not so pleased.

However, the parallel efficiency of a code is critical to the economic viability of using it on a given number of processors.

The efficiency provides a measure of the relative utilization of the resources of a parallel computer, as follows.

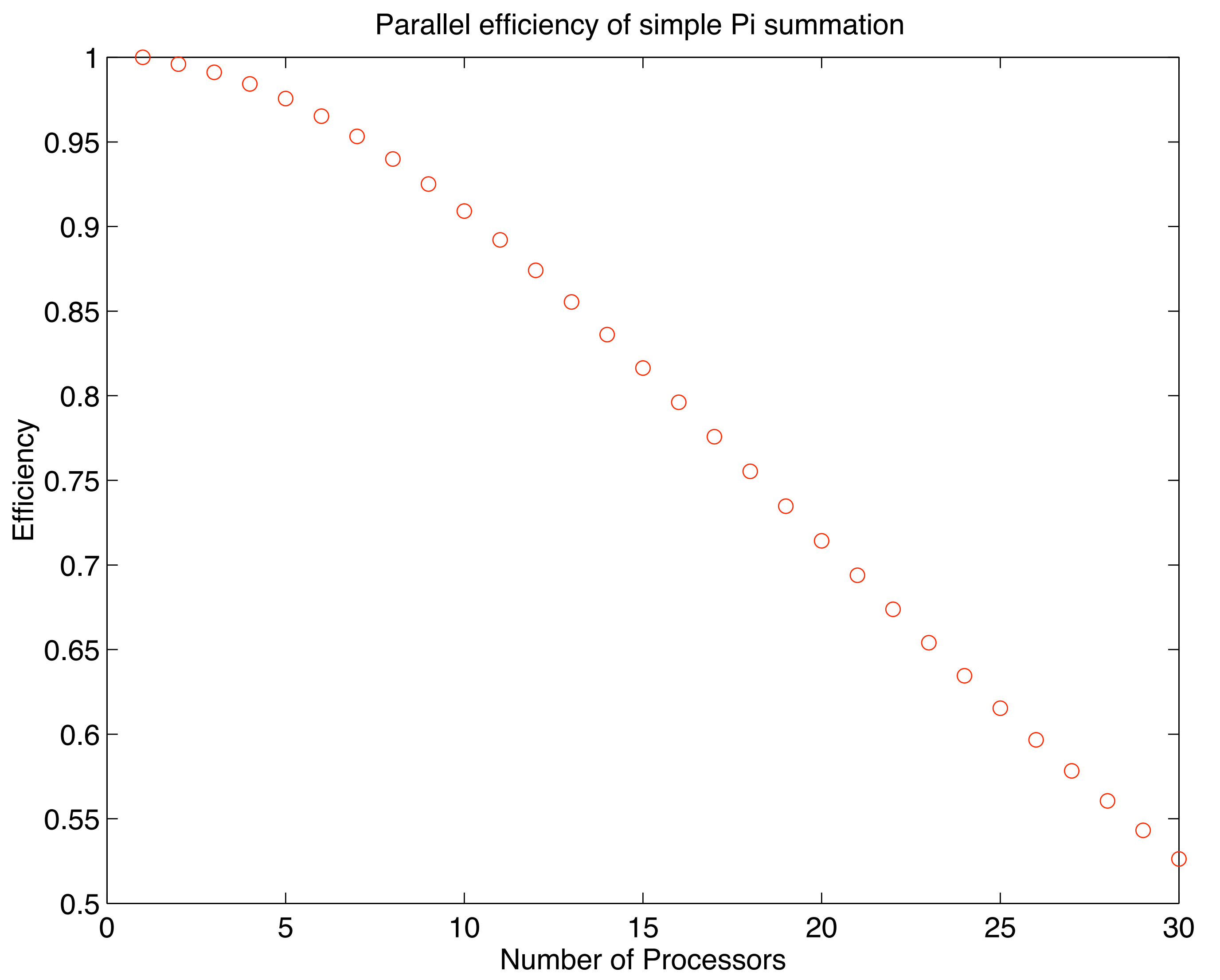


Figure 10: Hypothetical performance of a parallel implementation of summation: efficiency.

6.3 Efficiency and cost

Suppose that you have to pay the same rate ρ for time for each processor used, without regard for problem size or total number of processors used.

This very simple charging algorithm simplifies our argument.

The cost C_1 of using one processor to do a job is

$$C_1 = \rho T_1$$

(ρ is the cost per unit time) whereas the cost C_P of using P processors is

$$C_P = \rho T_P P = \frac{\rho T_1}{E_P} = \frac{C_1}{E_P}. \tag{6.14}$$

Thus the cost of using P processors is related to the sequential cost by a simple factor of $1/E_P$.

A more realistic measure of the cost of using a parallel computer would be to compare with the cost on an independent sequential computer.

7 Limits to performance

There are many things that can limit parallel performance. Here we present some key results that quantify some of these effects.

7.1 Amdahl's Law

Suppose that a particular code takes time T_1 to execute on one processor and that there is a **sequential fraction** f of this code that cannot be (or has not been) parallelized (for whatever reason).

Amdahl's Law assumes that, no matter how many processors P are used, the sequential part of the code will require at least fT_1 time to complete.

The remainder of the code, by assumption, takes $(1-f)T_1$ time to execute on one processor.

Amdahl's Law assumes that this part of the code will require at least $(1-f)T_1/P$ time to execute on P processors.

Combining these two time estimates, the time T_P to execute this code using P processors is bounded below by the sum of the sequential fraction fT_1 of the execution time and the remaining fraction $(1-f)T_1$ divided by P:

$$T_P \geq fT_1 + \frac{(1-f)T_1}{P}. \tag{7.15}$$

Combining (7.15) and (6.10), we obtain **Amdahl's Law**:

$$S_P \leq \frac{1}{f + (1-f)/P}. \tag{7.16}$$

Even with an infinite number of processors, the speed-up could not exceed

$$S_P \leq \frac{1}{f}. \tag{7.17}$$

This weaker statement is also sometimes referred to as Amdahl's Law. It says that the ratio of the parallel and sequential execution times cannot be less that the sequential fraction of the code:

$$\frac{T_P}{T_1} \geq f. \tag{7.18}$$

7.2 But wait there's more

Other factors can limit parallel performance in addition to the sequential fraction of a code.

This is why inequalities are used in (7.15) and (7.16).

In all the examples we have seen, there is some communication that must go on between processors which would not occur (or be simpler) in the sequential case.

Thus we may assume that communication will add some additional overhead, as a general rule.

This will arise in a variety of ways (through message latency, network or bus contention, or network bandwidth limitations) depending on particular architectural details of the computer, but it will almost always appear in some guise when $P > 1$.

7.3 But wait there's more

Lack of synchronization also limits parallel performance

Some parts of a parallel computation cannot be initiated until other parts are completed. Some processors may be forced to sit idle waiting for others to complete required tasks. In this time, no useful computation is done.

This means that the "parallel" computation time will not be the sequential time divided by P even if there is no sequential fraction to the code and the communication volume is minimal.

A frequent cause of idleness is a lack of load balance (Section 7.5) in one part of the computation which is followed by some communication.

The assumptions here do not take into account the fact that the individual processor performance may *increase*. This can and does allow speed-up to exceed the theoretical linear limit in practical calculations [3]. Amdahl's law assumes uniform processor behavior, which is a reasonable approximation. However, individual processor performance is becoming increasingly data dependent, so caution should be exercised in using the model.

7.4 Efficiency and Amdhahl's Law

Amdahl's Law implies that that the parallel efficiency (Definition 6.2) is bounded by

$$E_P \leq \frac{1}{1 + (P-1)f}. \tag{7.19}$$

When the quantity $(P-1)f$ is small, expanding the quotient implies that E_P decreases like

$$E_P \approx 1 - (P-1)f. \tag{7.20}$$

Thus f can be interpreted as the slope of the efficiency curve, E_P, as a function of P, near the point $P = 1$.

7.5 Load balance and efficiency

In the derivation of Amdahl's Law we assumed the best case for estimating parallel performance, namely that the parallel work could be perfectly divided into P parts. This is the case of perfect *load balance* (Definition 1.5.7), where *load* refers to a measure of the amount of work done by each individual processor.

Recall that the definition of load balance (Definition 1.5.7) is stated in terms of the times t_i for each processor to do its part of the calculation. The calculation is load balanced when all the t_i's are approximately the same. The parallel execution time can be defined simply in terms of the t_i's as follows:

$$T_P = \max \{ t_i \; : \; 1 \le i \le P \} . \tag{7.21}$$

It is reasonable to assume that

$$T_1 \le \sum_{1 \le i \le P} t_i . \tag{7.22}$$

If not, then the parallel execution could provide a faster sequential execution by executing each separate part in sequence.

Applying the definition of efficiency (Definition 6.2) and (7.21) and (7.22), we find

$$E_P = \frac{T_1}{PT_P} \leq \frac{\sum_{1\leq i\leq P} t_i}{P \max\{t_i \ : \ 1 \leq i \leq P\}} = \frac{\text{ave}\{t_i \ : \ 1 \leq i \leq P\}}{\max\{t_i \ : \ 1 \leq i \leq P\}} \tag{7.23}$$

where we recall the definition (1.5.3) of ave $\{t_i \ : \ 1 \leq i \leq P\}$. Thus the ratio of the average time on each processor to the maximum time on each processor, which we used as the definition of load balance (Definition 4.4), gives an upper bound on the efficiency of a calculation. We state this important result as the following Theorem.

Theorem 7.1 *Let t_i denote the time for the i-th processor to do its part of the calculation. Suppose that (7.22) holds. Then the efficiency of the calculation can never exceed the load balance:*

$$E_P \leq \beta := \frac{\text{ave}\{t_i \ : \ 1 \leq i \leq P\}}{\max\{t_i \ : \ 1 \leq i \leq P\}} \tag{7.24}$$

where the numerator was defined in (1.5.3) and β denotes the load balance (Definition 4.4).

It is permissible to combine the bounds (7.19) (Amdahl's Law for parallel efficiency) and (7.24). The latter applies to trivially parallel tasks where the sequential fraction f is zero. The parallel speed-up and efficiency of two successive tasks can be estimated as follows from the corresponding speed-up and efficiency of the separate tasks. Suppose that $T_{\cdot}^{a}$ and $T_{\cdot}^{b}$ denote the times of execution of the two separate tasks, labelled "a" and "b" respectively. Thus T_1^a denotes the sequential time of task "a", T_P^b denotes the parallel time of task "b", etc. It is reasonable to assume that the sequential time T_1^{a+b} of the combined tasks "$a + b$" is the sum of the sequential times of the separate parts:

$$T_1^{a+b} = T_1^a + T_1^b. \tag{7.25}$$

On the other hand, it would be prudent only to assume that the parallel time T_1^{a+b} of the combined tasks "$a + b$" is not less than the sum of the parallel times of the separate parts:

$$T_P^{a+b} \geq T_P^a + T_P^b \tag{7.26}$$

as there could be additional contributions to parallel execution time due to communication or synchronization.

Then one can show by simple algebra that the corresponding speed-up S_P^{a+b} of the combined tasks satisfies

$$S_P^{a+b} \leq \left(\frac{\lambda}{S_P^a} + \frac{1-\lambda}{S_P^b} \right)^{-1} \tag{7.27}$$

where

$$\lambda := \frac{T_1^a}{T_1^a + T_1^b}. \tag{7.28}$$

Since efficiency is just speed-up divided by P, we similarly have

$$E_P^{a+b} \leq \left(\frac{\lambda}{E_P^a} + \frac{1-\lambda}{E_P^b} \right)^{-1}. \tag{7.29}$$

8 Scalability

Message: 1023110, 88 lines Posted: 5:34pm EST, Mon Nov 25/85, imported: .... Subject: Challenge from Alan Karp To: Numerical-Analysis, ... From GOLUB@SU-SCORE.ARPA

I have just returned from the Second SIAM Conference on Parallel Processing for Scientific Computing in Norfolk, Virginia. There I heard about 1,000 processor systems, 4,000 processor systems, and even a proposed 1,000,000 processor system. Since I wonder if such systems are the best way to do general purpose, scientific computing, I am making the following offer.

I will pay $100 to the first person to demonstrate a speedup of at least 200 on a general purpose, MIMD computer used for scientific computing. This offer will be withdrawn at 11:59 PM on 31 December 1995.

Some definitions are in order. .....

8.1 Prizes and scalability

Karp's Challenge was met in [7], but they authors also modified the standard notion of speedup to include data size.

Later, the Bell[a] Prizes have been awarded each year since 1988 [11].

Suppose a problem can be characterized by a data size N.

The characteristics of a parallel code may vary significantly as N is varied, even more than for a sequential computation as depicted in Figure 6.

For very large data sizes, some small effects (such as a sequential section) may become insignificant relative to the overall computation time.

The notion of *scalability* reflects this [7].

[a]Gordon Bell (1934–) spent 23 years at Digital Equipment Corporation as Vice President of Research and Development, where he was the architect of various mini- and time-sharing computers and led the development of DEC's VAX and the VAX Computing Environment. Bell has been involved in, or responsible for, the design of many products at Digital, Encore, Ardent, and a score of other companies. He has been involved in the design of about 30 multiprocessors.

8.2 Scalability example

The parallelization of the summation problem in (4.4) and (4.5) has been modeled for a broadcast medium to have an execution time proportional to $N/P + (\gamma+1)P$ (see (5.9)).

Choosing P as a function of N can yield arbitrarily large speedup in apparent contradiction to Amdahl's Law.

For the summation problem, cf. (5.9),

$$E_P = \left(1 + (\gamma+1)\frac{P^2}{N}\right)^{-1}. \tag{8.30}$$

For fixed N, this implies that the efficiency goes to zero as P goes to infinity.

But if we choose P as a function of N as N increases, we can obtain an efficiency that does not go to zero, as it would for the case of a fixed N.

For example, suppose P and N are related by the equation $P = \sqrt{N}$. Then the efficiency is constant: $E_P = (1 + (\gamma+1))^{-1}$.

8.3 Data-dependent performance measures

As a preliminary step in the discussion of scalability, we introduce the notions of data-dependent speedup and data-dependent parallel efficiency as a function of both the number of processors P and the data size N.

Definition 8.1 *The* **data-dependent speedup** $S_{P,N}$ *using* P *processors for a problem with data size* N *is defined to be the ratio of the sequential and parallel execution times*

$$S_{P,N} = \frac{T_{1,N}}{T_{P,N}}, \tag{8.31}$$

where $T_{1,N}$ *is the time of execution of the best sequential solution of the problem with data size* N *for and* $T_{P,N}$ *is the time of execution of the parallel solution of the problem with data size* N *on* P *processors. The* **data-dependent parallel efficiency** $E_{P,N}$ *using* P *processors is defined to be the ratio of the data-dependent speedup and* P*:*

$$E_{P,N} = \frac{S_{P,N}}{P} = \frac{T_{1,N}}{PT_{P,N}}. \tag{8.32}$$

8.4 Scalability definition

Definition 8.2 *An algorithm is said to be* **scalable** *if there is a* **minimal efficiency** $\epsilon > 0$ *such that, given any problem size* N, *there is a number of processors* $P(N)$, *which tends to infinity when* N *tends to infinity, such that the efficiency* $E_{P(N),N}$ *remains bounded below by* ϵ, *that is,*

$$E_{P(N),N} \geq \epsilon > 0 \tag{8.33}$$

as N *is made arbitrarily large*.

The number of processors $P(N)$ chosen for a given data size N is not specified in this definition, except that it should increase to infinity as N increases.

Practically, often one has a precise choice of $P(N)$ for which we ask whether or not (8.33) holds, such as the memory-constrained case (8.41) described subsequently.

Leads to more restrictive notion of scalability than Definition 8.2.

8.5 Scalability of summation

Suppose we take $P = \sqrt{N}$ in the summation problem, then

$$S_{P(N),N} = k\ N^{1/2} = k\ P \tag{8.34}$$

with the constant $k = \frac{1}{2+\gamma}$.

The scaled efficiency is a fixed value, namely, $E_{P(N),N} = k = \frac{1}{2+\gamma}$.

However, this is not the only $P(N)$ that can lead to scalability.

Let us ask what $P(N)$ would correspond to a given ϵ in Definition 8.2.

The efficiency of the summation algorithm for any $P(N)$ is

$$E_{P(N),N} = \frac{S_{P(N),N}}{P(N)} = \left(1 + (\gamma + 1)\frac{P(N)^2}{N}\right)^{-1}. \tag{8.35}$$

We apply the efficiency constraint as an inequality where $E_{P(N),N} \geq \epsilon$ and rearrange, resulting in

$$P(N)\ \leq\ \sqrt{k\ N}, \tag{8.36}$$

where $k = (\frac{1}{\epsilon} - 1)/(\gamma + 1)$.

8.6 Scalability and Amdahl's Law

The notion of *scalability* seems at first glance to contradict Amdahl's Law, which said that if there is any sequential fraction $f > 0$ of a code then its efficiency must necessarily decrease to zero as P goes to infinity according to (7.19).

The new ingredient that resolves the dilemma is that for a scalable code, the sequential fraction f_N itself decreases to zero as N goes to infinity.

In particular, the sequential fraction f_N of a scalable algorithm must satisfy the bound (in view of (7.16))

$$f_N \leq \frac{C}{P(N)} \tag{8.37}$$

for some constant C.

In the summation problem, the sequential part of the problem is the communication required to collect the partial sums.

This is a function of P alone, whereas the total computation is proportional to N.

8.7 Memory limits on performance

The memory $M(N, P)$ required to implement an algorithm with P processors and data size N may exceed the physical limits of a particular machine. $M(N, P)$ denotes the amount of memory required *per processor*; the total system memory required is thus P times this, i.e., $P \cdot M(N, P)$.

For a sequential problem, it would be natural to assume that

$$M(N, 1) \leq c_0 + c_1 N$$

for constants c_0 and c_1, since N measures the data size.

However, for the parallel case, it is not so easy to say how the local memory size $M(N, P)$ should depend on N (and P).

It may be possible to divide the memory requirements evenly, leading to $M(N, P) \leq c_0' + c_1' N/P$ for possibly different constants c_0' and c_1'.

But codes with such an ideal memory behavior may have less than ideal performance.

8.8 Memory-constrained scaling

Definition 8.3 *An algorithm is said to be* **scalable with respect to memory** *if, given any problem size N, there is a number of processors $P(N)$, which tends to infinity when N tends to infinity, such that the efficiency $E_{P(N),N}$ remains bounded below by a positive constant as in (8.33), and furthermore*

$$M(N, P(N)) \leq M_{\text{MAX}} \tag{8.38}$$

as N is made arbitrarily large, where M_{MAX} is some fixed constant.

Again, the number of processors $P(N)$ chosen for a given data size N has not been specified in this definition, except that it should increase indefinitely as N increases.

We now show that the summation algorithm is not scalable with respect to memory.

8.9 Summation not memory-scalable

Consider the memory scalability of the summation problem in (4.4) and (4.5).

To be memory scalable by Definition 8.3 two constraints must be satisfied. The efficiency, $E_{P(N),N}$ must be bounded below by a positive constant ϵ. In addition, the memory usage $M_{N,P(N)} = \frac{N}{P(N)}$ is subject to the constraint

$$\frac{N}{P(N)} \; \leq \; M_0. \tag{8.39}$$

Therefore, $P(N)$ must be bounded above and below according to

$$\frac{1}{M_0}N \; \leq \; P(N) \; \leq \; \sqrt{k\ N}, \tag{8.40}$$

where $k = (\frac{1}{\epsilon} - 1)/(\gamma + 1)$.

These conditions do not have a solution for $N > kM_0^2$.

The phrase **memory-constrained scaling** refers to the performance of an algorithm for a particular amount of memory per processor, independent of N and P. This corresponds to choosing $P(N)$ such that

$$P(N) = \min \{P \ : \ M(N,P) \leq M_{\text{MAX}}\}, \tag{8.41}$$

where M_{MAX} is any constant (not necessarily the same as in Definition 8.3).

Here we are making the simplifying assumption that $M(N,P)$ is always a decreasing function of P for any given N.

Even if a code is scalable with respect to memory, it may not be scalable for a particular computer if the constant M_{MAX} in (8.38) is too large for the computer to support.

If we take M_{MAX} to be the maximum allowable for a given computer, we can then ask what the efficiency is for $P(N)$ defined by (8.41).

If it is bounded below, then of course the code is scalable with respect to memory.

8.10 Weak scaling

Although the definitions of scalability with respect to memory are theoretically clear, they are difficult to verify in practice, at least on parallel processors which fixed memory at each processor.

One might require a single processor with a very large amount of memory to do the sequential computation to determine $T_{1,N}$.

One variant of scalability, *scaled efficiency* or *weak scaling*, considers the scalability and efficiency with machine resources (memory and processors) as a function of the problem size.

Scaled efficiency has been used to study the multigrid iterative method for solving a partial differential equation [17]. The work estimate (and actual execution time) for this algorithm scales linearly with the data size N, i.e.,

$$T_{1,N} = cN. \tag{8.42}$$

For ideal performance, we would then expect

$$T_{P,N} = \frac{cN}{P}. \tag{8.43}$$

8.11 Multigrid scaling

Suppose we are interested in the efficiency of multigrid in the case that we use the same amount of memory on each processor, independent of the number of processors (cf. the memory-constrained case considered previously). Let n denote this amount; the total data size is $N = nP$. Then $T_{P,nP}$ is the time we would measure for P processors. To compute the efficiency, we would need to know the time $T_{1,nP}$ which we could not run unless there were a particular processor with nP words of memory. However, (8.42) implies that

$$T_{1,nP} = PT_{1,n}. \tag{8.44}$$

Comparing (8.32), we see that

$$\widehat{E}_{P,n} := \frac{T_{1,n}}{T_{P,nP}} \tag{8.45}$$

provides an estimate of efficiency $E_{P,n}$ in which the amount of memory per processor is held fixed. This is a type of memory-constrained scaling, but of course it is not a general definition. For algorithms whose time of execution is not linear in the data size, a more complex formula would be appropriate.

8.12 Repeat of Table 1

U-cycle weak scaling

Table 2: Parallel performance of U-cycle multigrid for Poisson's equation on IBM BG/L; IT$_j$ denotes average number of iterations of PSOR with relaxation parameter ω for solving the coarsest grid equations. Time in seconds.

P	h_l	$E(n,P)$	Total	Comm.	IT$_U$	On Ω_j			
		$(n=2^{10})$	Time	Time		h_j	IT$_j$	Time	ω
1	$1/2^{10}$	1.0	3.8375	0	11	1/8	6	0.0021	1.52
4	$1/2^{11}$	0.9841	3.8994	0.2217	11	1/16	10	0.0073	1.66
16	$1/2^{12}$	1.0687	3.5908	0.2209	10	1/32	64	0.5282	1.88
64	$1/2^{13}$	0.9008	4.2601	0.3424	10	1/64	105	0.1040	1.91
256	$1/2^{14}$	0.8782	4.3697	0.3648	10	1/128	178	0.1965	1.96
1024	$1/2^{15}$	0.9274	4.1379	0.3962	9	1/256	332	0.3674	1.98

9 Some Perspective

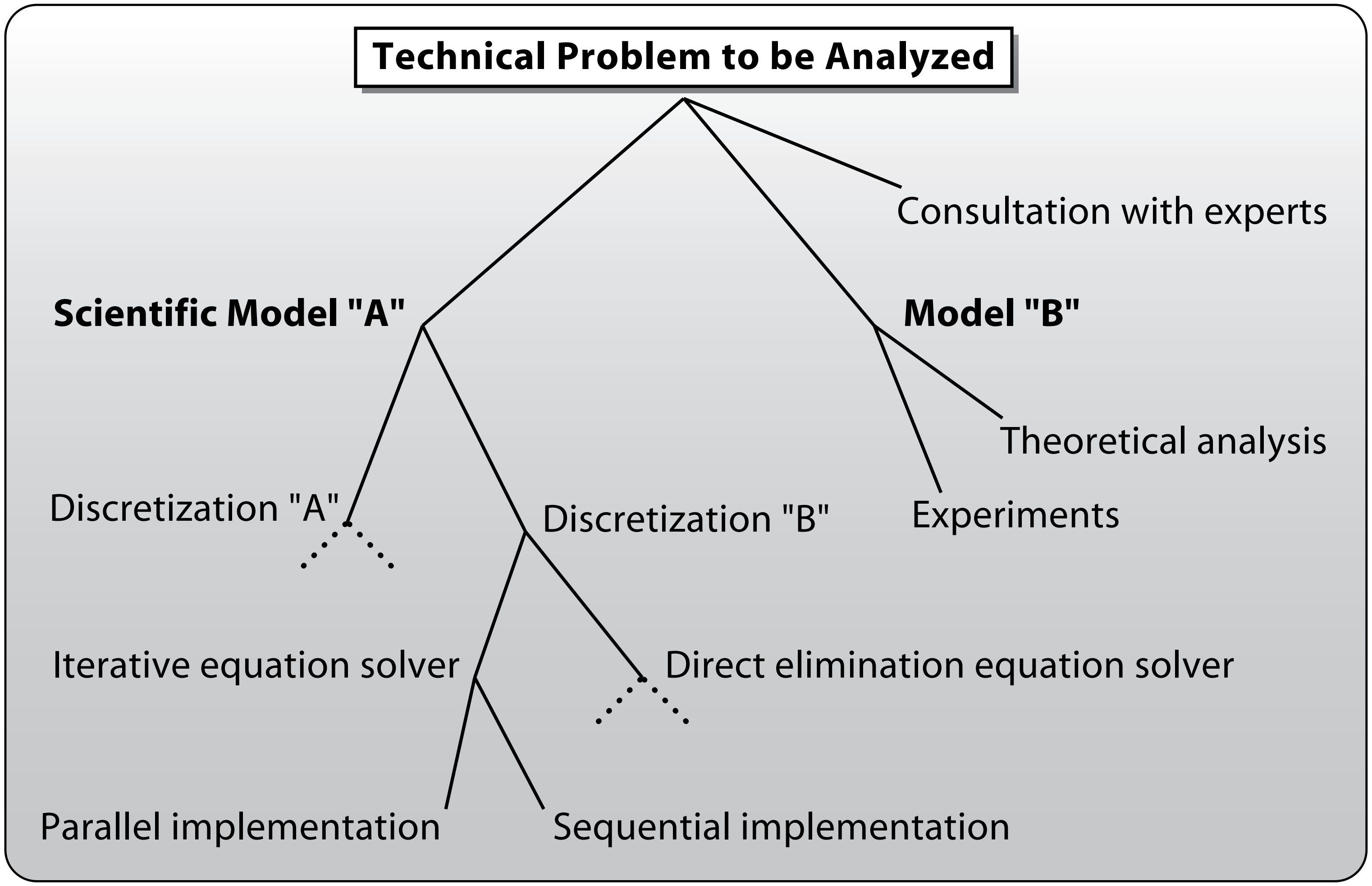

Technical Problem to be Analyzed
Consultation with experts
Scientific Model "A"
Model "B"
Theoretical analysis
Discretization "A"
Discretization "B"
Experiments
Iterative equation solver
Direct elimination equation solver
Parallel implementation
Sequential implementation

Figure 11: The “problem tree” for scientific problem solving. There are many options to try to achieve the same goal.

10 PMS Notation for Computer Architecture

We utilize the “PMS” notation [18] to describe key components of a computer system. The letters stand for

- Processor
  - a device that performs mathematical operations on data
- Memory
  - a device that stores data and can make it available to other devices
- Switch
  - a device that allows data to be transferred between other devices.

In a PMS diagram, the P’s, M’s, and S’s are connected by “wires” indicated simply by lines. In the simplest case, there would be only one wire coming to a P or an M, with S’s allowing multiple wires to be joined. The full model contains other letters (e.g., “K” for “control”) but we will only use these three elements in our descriptions.

10.1 Diagram of a sequential CPU

PMS notation can be used at multiple scales.

Basic functioning of a **central processing unit** (cpu) can be explained using a PMS diagram.

Registers and memory are connected not directly but through the switch.

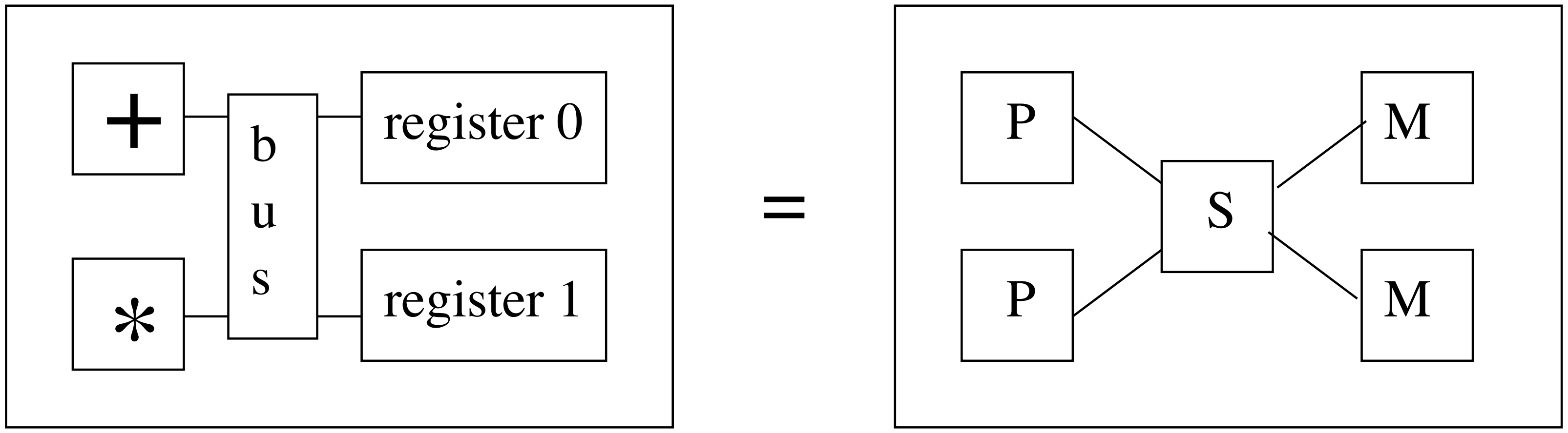


Figure 12: The PMS notation for a hypothetical cpu with only two registers and two functional units connected by a switch.

10.2 Diagram of a basic processor

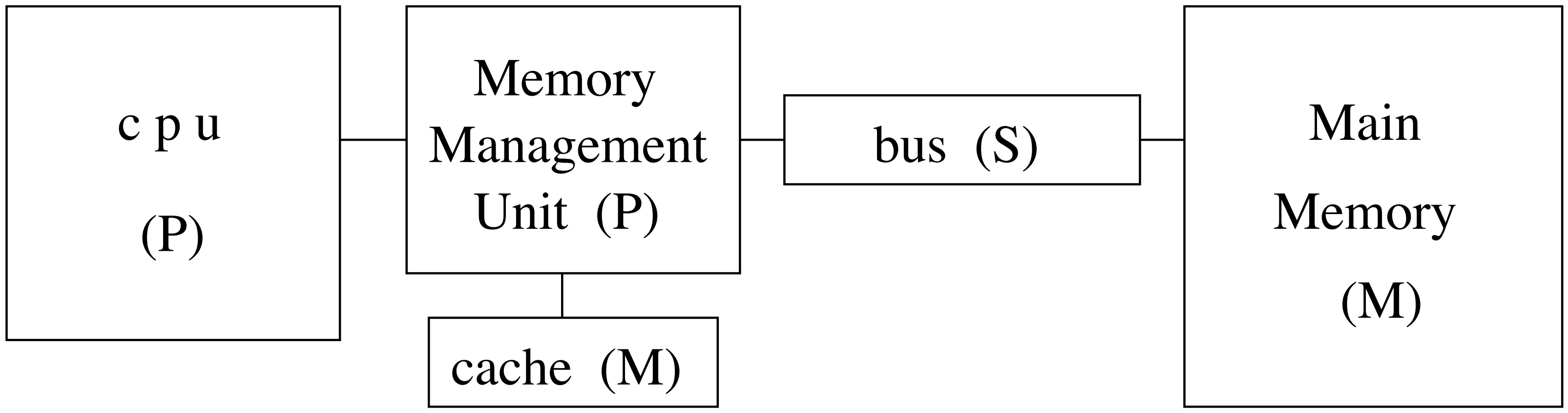


Figure 13: The PMS notation for a basic processor with a cache and memory management logic.

Caches often loaded with chunks of contiguous data (an entire **cache line** or block) whenever there is a cache fault.

Done to increase performance: faster to load contiguous memory locations at one time due to the design of current memory chips.

Loading several nearby memory locations at one time is motivated by the assumption of **locality of reference**, namely, that if one memory location is read now, then other nearby memory locations will likely be read soon.

10.3 Shared Memory Multiprocessor

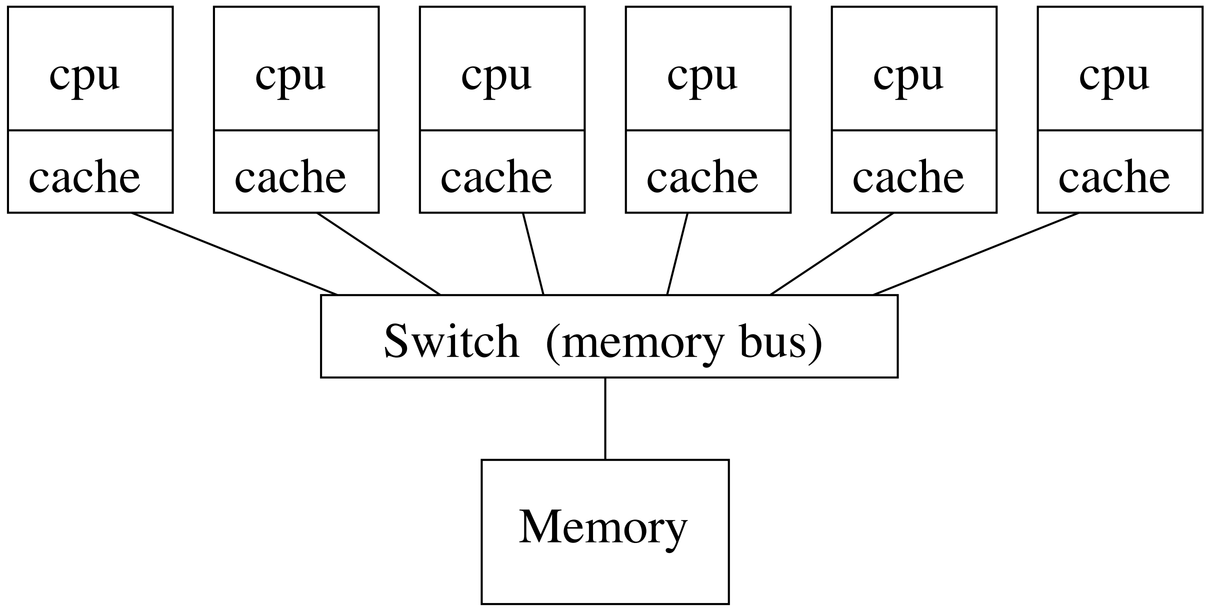


Figure 14: The PMS notation for the a shared-memory multiprocessor with a local cache for each processor.

Introduction of a local cache at each processor increases the potential performance of the overall multiprocessor but also complicates the design substantially.

The major problem is insuring **cache coherence**, that is the agreement of cache values which are mirroring the same values in shared memory.

10.4 Shared Memory Bottleneck

The speed of the memory bus ultimately limits the speedup.

Theorem 10.1 *If a computation has a work/memory ratio ρ_{WM}, and an individual processor does γ operations per memory transfer, then*

$$S_P \leq \gamma\, \rho_{\mathrm{WM}} \tag{10.46}$$

on a basic shared-memory multiprocessor regardless of number of processors P.

Both GPU's and multicore chips use basic shared memory architecture.

New algorithms are required to improve the work/memory ratio ρ_{WM}.

Different architecture required to scale up even to hundreds of processors.

10.5 Distributed-Memory Multicomputer

The set of network “topologies” which have been proposed or even implemented are as diverse as graph theory itself: most basic is a ring.

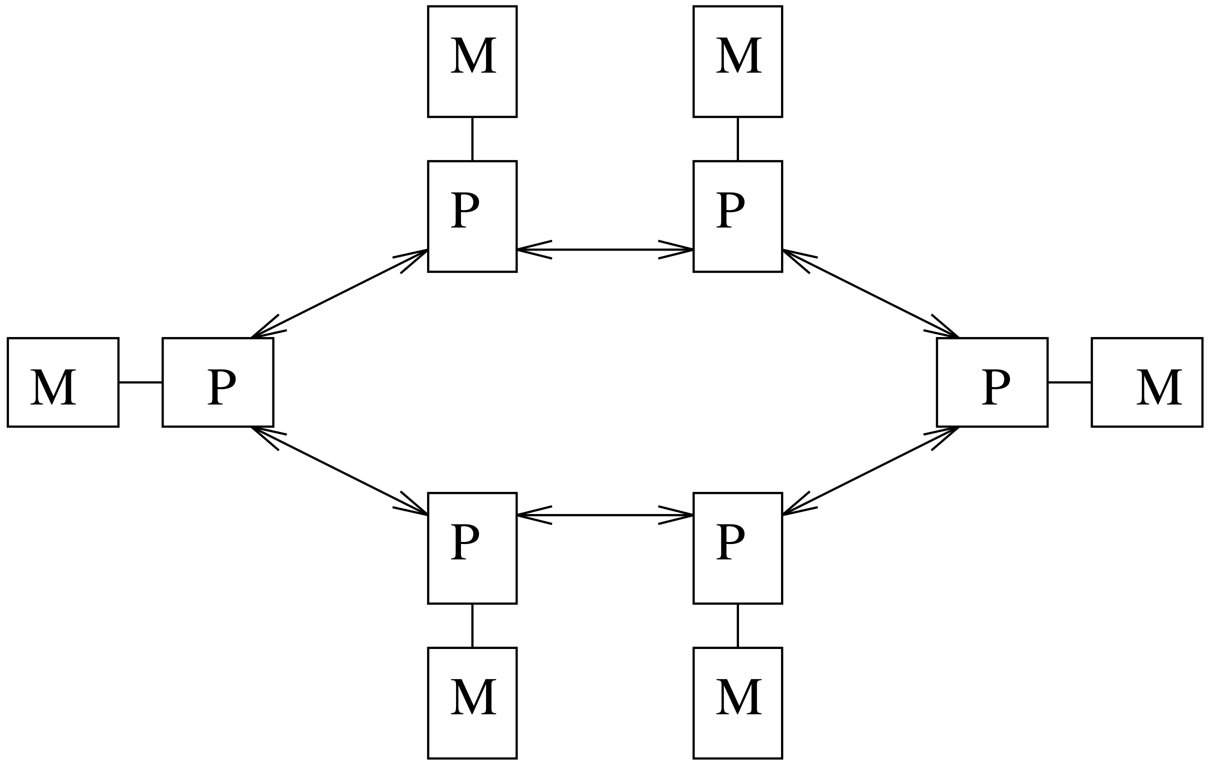


Figure 15: A distributed-memory multicomputer with six processors connected by a network with a ring topology.

10.6 More networks

Two and three dimensional meshes are natural extensions of rings and are used in current commercial parallel systems.

The two-dimensional mesh requires four connections per processor, while the three-dimensional mesh requires six connections per processor.

Many other graphs have been proposed, and some of them have found their way into MPP systems.

These include trees, fat trees, and hypercubes.

Another network of current interest is based on using a cross-bar switch. The resulting graph can be thought of as the complete graph on P nodes, i.e., all of the nodes are directly connected.

A similar type of switch effectively allows arbitrary connections through the use of a **multi-stage interconnect**. The connections in this case can not be represented by a static graph. Any of the nodes can be effectively directly connected to any other node at any given time, but the set of simultaneous connections cannot be very large.

10.7 Hypercubes

Graph: edges of a d-dimensional hypercube, vertices are at points $(i_1, \ldots, i_d)$ where each i_j takes the value zero or one.

This can be viewed as the binary representation of an integer p in the range $0 \leq p \leq 2^d - 1$.

There are d connections per processor: bandwidth scales up as the size of the processor increases, but so does cost.

A related network is **cube-connected cycles**.

Building blocks out of a ring of d processors, each having at least three available connections with two used for the ring connections.

The third connection is used to connect 2^d such rings in a hypercube. Thus one has $d2^d$ processors in this approach.

Can easily be implemented with off-the-shelf technology.

10.8 Data exchange

The cost of exchanging data in a distributed-memory multicomputer can be approximated by the model

$$\lambda + \mu * m \tag{10.47}$$

where m is the number of words being sent, λ is the “latency” corresponding to the cost of sending a **null message**, i.e. a message of no length, and μ is the incremental time required to send an additional word in a message.

Model does not account for contention in the network that occurs when simultaneous communications involve intersecting links in the network.

10.9 NOW what?

Network of workstations (**NOW**) is a basic (low cost) distributed-memory parallel computer.

10.10 Comparison of Parallel Architectures

Figure 16 depicts quantitative and qualitative aspects of four different commercial designs.

The shaded areas depict the pathway from processors (including local memories, e.g. caches) to main memory.

The width of the pathway is intended to indicate relative bandwidths in the various designs.

It is roughly proportional to the logarithm of the bandwidth, but no attempt has been made to make this exact.

The length of the pathway also indicates to some extent the latencies for these various designs.

The vector supercomputer has a relatively low latency, where as the distributed-memory computer or network of workstations has a relatively high latency.

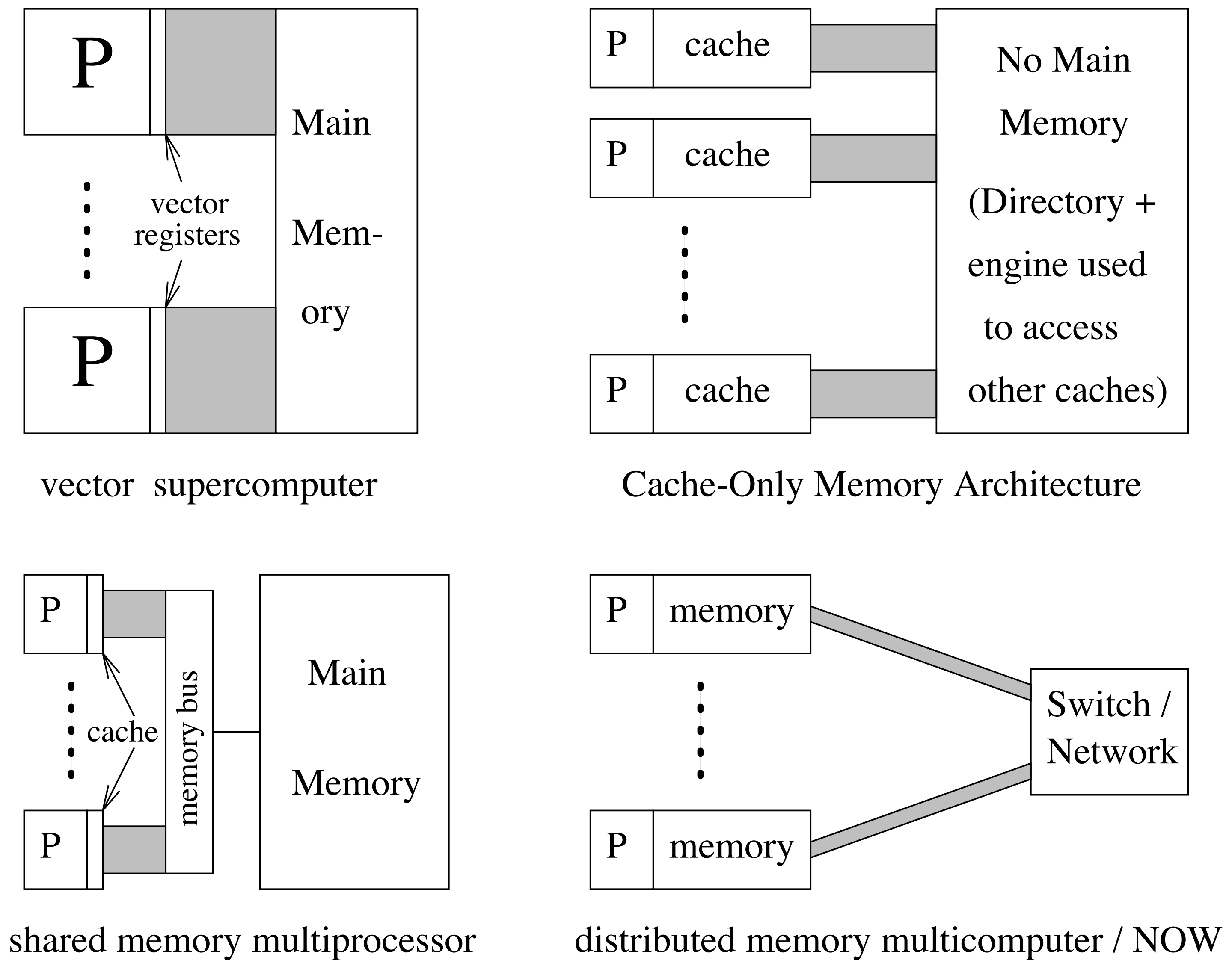

P
P
vector registers
Main Mem- ory
vector supercomputer
P cache
P cache
P cache
No Main Memory (Directory + engine used to access other caches)
Cache-Only Memory Architecture
P
P
cache
memory bus
Main Memory
shared memory multiprocessor
P memory
P memory
Switch / Network
distributed memory multicomputer / NOW

Figure 16: Comparison of various architectures. The shaded areas indicated the bandwidth to memory. Wider paths indicate faster speeds.

10.11 Contrasting Different Parallel Architectures

Figure 16 contrasts different design elements.

The vector supercomputer has a relatively small local memory (its vector registers) but makes up for this in a very high bandwidth and low latency to main memory.

The COMA design provides a large local memory without sacrificing the simplicity of shared memory.

The shared memory multiprocessor also retains this but does not have a very large local memory.

Finally, the distributed-memory computer or a network of workstations has a large local memory similar to the COMA design but cannot match the bandwidth to main memory.

Instead of the COMA directory system, one simply has a network to transmit messages.

Exercise: compare and contrast GPU and multicore architectures.

10.12 Some merits and demerits

No absolute comparisons can be made regarding the relative performance of the different designs.

Different algorithms will perform differently on different architectures.

- Shared memory multiprocessor will not perform well on problems with limited amounts of computation done per memory reference, that is, ones for which the work/memory ratio, ρ_{WM}, is small (see Definition 4.2).
- A network of workstations cannot perform well on computations that involve a large number of small memory transfers to different processors.
- Massive bandwidth to memory made vector supercomputers relatively easy to use.
- COMA had interesting benefits but at the cost of a complex memory system (directory).

10.13 Future architecture

Energy will become the most important issue in chip design.

- Has already stopped increase in cycle speed and initiated move to multi-core.
- Will put limits on how much of future chips can be "lit up" at any one time.

There will be significant amounts of "dark silicon" on every chip.

Andrew Chien suggests "10 by 10" approach to overcome the "90-10" rule: see suggested reading on my web page.

- Proposes having ten specialized units instead of monolithic cores: e.g., CPU, GPU, DSP, Anton, FPGA, etc. functionality on a single chip.
- This gang of ten would be repeated as space allows.

11 Data Dependences

Dependences force order among computations.

```
z = 3.14159                 z = 3.14159
x = z + 3                   z = z + 3
y = z + 6                   y = z + 6
```

Programs 11

The last two statements in the left column can be interchanged without changing the resulting values of `x` or `y`.

However the last two statements in the right column cannot be interchanged without changing the resulting value of `y`.

In either case, the assignment to the variable `z` (the first line of code in both columns) must come first.

The requirement to maintain execution order results since the variables used in one line *depend* on the assigned values in another line.

Our purpose here is to formalize the definitions of such dependences.

Definition 11.1 *The* **read set** $\mathcal{R}(C)$ *of a computation* C *is the set of memory locations from which data are read during this computation. The* **write set** $\mathcal{W}(C)$ *of a computation* C *is the set of memory locations to which data are written during this computation. The* **access set** $\mathcal{A}(C)$ *is defined to be* $\mathcal{R}(C) \cup \mathcal{W}(C)$.

Sometimes the read set $\mathcal{R}(C)$ is called the `Use` or `In` set, and the write set $\mathcal{W}(C)$ is called the `Def` or `Out` set.

Definition 11.2 *Suppose that* C *and* D *are computations in a code. There is a* **(direct) data dependence** *between* C *and* D *if the following* **Bernstein conditions** *hold:*

$$\mathcal{W}(C) \cap \mathcal{R}(D) \neq \emptyset, \text{ or} \tag{11.48}$$

$$\mathcal{R}(C) \cap \mathcal{W}(D) \neq \emptyset, \text{or} \tag{11.49}$$

$$\mathcal{W}(C) \cap \mathcal{W}(D) \neq \emptyset. \tag{11.50}$$

Note that the order of C and D in Definition 11.2 does not affect whether we declare a data dependence between them.

11.1 Specific dependences and order

Computation C **occurs before** the computation D: $C\uparrow D$, if all of the lines of C precede all of the lines of D (with no overlap) in the standard order of execution.

Specific dependences (11.48) and (11.49) do depend on the order. If (11.48) holds for $C\uparrow D$, then (11.49) holds for $D\uparrow C$, and *vice versa*. The condition (11.50) is independent of order. It is important to distinguish between the different types of dependences in some cases, and so special names are given for the different sub-cases, as follows.

Suppose that the computation C occurs before the computation D. If (11.48) holds, then we say there is a **true dependence**, **flow dependence** or **forward dependence** (all equivalent terminology) from C to D. In the two fragments of code in Program 11, there is no forward dependence in the left column, but there is in the right column. If (11.49) holds, then we say there is an **anti-dependence** or a **backward dependence** between C and D. There is no backward dependence in either column in Program 11. If (11.50) holds, then we say there is an **output dependence** between C and D. There is no output dependence in either column in Program 11.

11.2 Dependence implications

When there is any dependence between computations, we cannot execute them in parallel.

We are most interested in proving that there are *not* dependences between computations in many cases.

However, proving that there *are* dependences between computations can halt a futile effort to parallelize code before it starts.

Dogma: if there are no dependences between two computations, they can be done in parallel.

Dependences can be different depending on different *executions* of the code. For example, the dependences in the code Program 11.2 can only be determined once the data `i` and `j` are read in.

```
read i,j
x(i)=y(j)
y(i)=x(j)
```

Program 11.2: Code with potential dependences based on I/O.

11.3 Loop-Carried Data Dependences

Loop-level parallelism means each loop iteration is executed in parallel.

Dependences inhibit parallelism and lead to inefficiencies even on current scalar processors.

Consider the computation of a norm.

```
        sum = x(1)*x(1)
        DO 1 I=2,N
  1     sum = sum + x(I)*x(I)
```

Program 11.3: Simple code for norm evaluation.

Dependence on `sum` means we need to complete one iteration of the loop before we can start the next.

One way to break the dependence is to reorder the summation, e.g., by adding the odd- and even-subscripted array elements separately. This could be written in Fortran as shown in Program 11.4.

11.4 Loop splitting

Loop splitting [8] can change dramatically the performance, cf. Figure 17.

```
         sum1 = x(1)*x(1)
         sum2 = x(2)*x(2)
         DO 1 I=2,N/2
         sum1 = sum1 + x(2*I-1)*x(2*I-1)
  1      sum2 = sum2 + x(2*I)*x(2*I)
         sum = sum1 + sum2
```

Program 11.4: Code with a split loop for norm evaluation.

The number of splittings in Program 11.4 is defined to be one, whereas Program 11.3 is the case of zero splittings. Any number of splittings k can be done via the code shown in Program 11.5. Figure 17 depicts the performance of Program 11.5 for the evaluation of a norm for vectors of length $n = 16,000$ on popular workstations made by Sun, Digital, IBM and H-P, as well as for the Cray C90 [8]. The key point is the distinct increase in performance of the workstations as the number of splittings is increased, at least to eight or so.

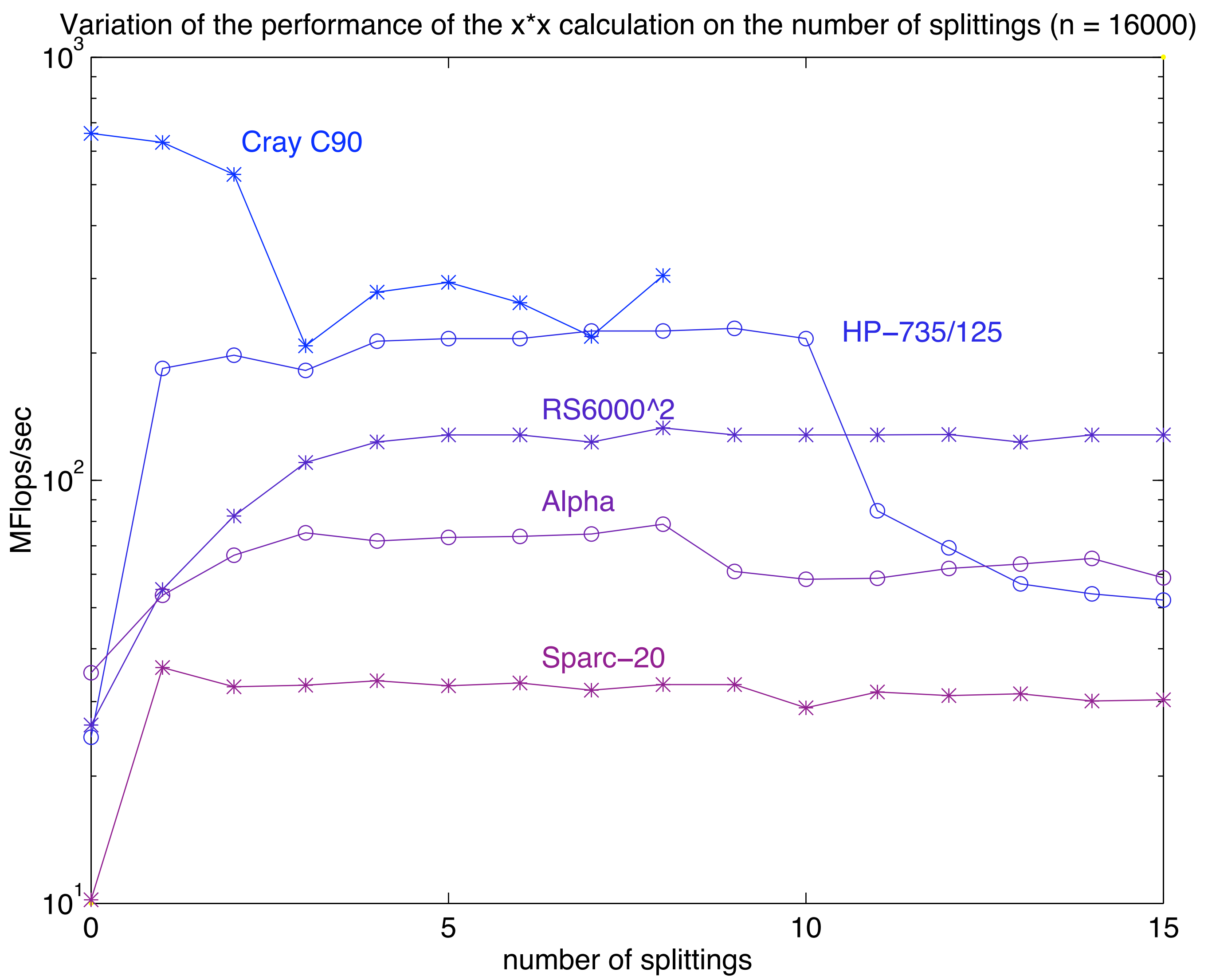


Figure 17: Performance of norm evaluation for vectors of length $n = 16,000$ on various computers as a function of the number of loop splittings.

11.5 Arbitrary loop splitting

```
      DO 1 j=1,k
1     summ(j) = x(j)*x(j)
      DO 3 i=2,N/k
      DO 2 j=1,k
2     summ(j) = summ(j) + x(k*(i-1)+j)*x(k*(i-1)+j)
3     continue
      sum = 0.0
      DO 4 j=1,k
4     sum = sum + summ(j)
```

Program 11.5: Code for norm evaluation with arbitrary number of loop splittings.

Loop splitting is not a universal panacea: severely degrades the performance on the Cray C90.

But this type of re-ordering of loops significant for parallelism at both instruction level as in Figure 17 and coarse-grained level.

Loop splitting is a cyclic (or modulo) decomposition.

11.6 Some definitions

We now turn to a detailed study of dependences in loops so that we can formalize what loop splitting does as well as to prepare for the study of more complex loops.

Loop-carried data dependences between program parts are an important special case of what we have considered so far in Definition 11.2.

Definition 11.3 *A loop is in* **normalized form** *if its index increases from* 0 *to its limit by* 1.

Can convert into normalized form by an affine change of index variables.

```
      do 1 I=11,31,3
    1   A(I)=B(2*I)
```

is normalized by writing $\mathtt{I} = 11 + 3\mathtt{J}$, yielding the normalized loop

```
      do 1 J=0,6
    1   A(11+3*J)=B(2*(11+3*J))
```

That is, all “strides” are eliminated by the obvious change of loop index, and loops with decreasing index are reversed.

The set of loop indices for normalized *nested* loops are **multi-indices**, i.e., n-tuples of non-negative integers, $I := (i_1, i_2, \cdots, i_n)$, where n is the number of loops. By convention, i_1 is the loop index for the outer-most loop, and so on with i_n being the loop index for the inner-most loop. We use the symbol $\vec{0}$ to denote the n-tuple consisting of n zeros. There is a natural total order "$<$" on multi-indices, *lexicographical order*, that is, the order used in dictionaries.

Definition 11.4 *The* **lexicographical order** *for multi-indices is defined by*

$$(i_1, i_2, \cdots, i_n) < (j_1, j_2, \cdots, j_n)$$

whenever $i_k < j_k$ for some $k \leq n$, with $i_\ell = j_\ell$ for all $\ell < k$ if $k > 1$.

We can then write $I \leq J$ if either $I < J$ or $I = J$, and similarly we write $J > J$ if $I < J$.

The standard order of evaluation of nested (normalized) loops provides the same total order on the set of loop indices (i.e., on multi-indices) as the lexicographical order $<$ defined in Definition 11.4. Indeed, the loop execution for index I *comes before* the loop execution for index J if and only if $I < J$.

11.7 Loop-carried dependence

If S is enclosed in at least $n \geq 1$ definite loops with main index variables $l_1, l_2, \cdots, l_n$, then S_I denotes the execution of S for which

$$(l_1, l_2, \cdots, l_n) = (i_1, i_2, \cdots, i_n) =: I.$$

Definition 11.5 *There is a* **loop-carried data dependence** *between parts S and T of a program if there are loop index vectors (i.e., multi-indices) I and J, with $I < J$, such that there is a data dependence between S_I and T_J.*

A loop-carried dependence can be described more precisely as forward (11.48), backward (11.49), or output (11.50) depending on which of the Bernstein conditions hold.

Note that we do not assume that the parts S and T are disjoint or in any particular order.

The ordering required following Definition 11.2 is enforced by the assumption $I < J$.

In view of this, $S_I \uparrow T_J$ in Definition 11.5.

11.8 Privatization of variables

Privatization of variables can remove some loop-carried dependences.

Some loop-carried dependences can be removed by declaring certain variables **private** or **local** to the loop in the language.

Equivalent to adding an extra array index to the variable to make it different for each iteration of the loop.

The simple code in Program 11.8 gives an example of a dependence caused by the use of the temporary variable `TEMP`.

```
      DO 1 I=1,100
      TEMP = I
 1    A(I) = 1.0/TEMP
```

Program 11.8: Code with a dependence caused by the use of a temporary variable.

But this is not an essential dependence.

11.9 Removing an inessential dependence

If we write this as in Program 11.9 there is no longer a dependence.

```
            DO 1 I=1,100
            TEMP(I) = I
  1         A(I) = 1.0/TEMP(I)
```

Program 11.9: Dependence removed by adding an index to the temporary variable.

Privatizing a variable in a loop is therefore a very simple concept.

However, not all variables can be made private without destroying correctness of the loop.

The addition of a line of the form (a pragma in OpenMP)

```
C*private  TEMP
```

to the code in Program 11.8 is intended to produce the equivalent result as if we had explicitly made `TEMP` a different variable for each index as is done in Program 11.9.

12 Dependences in Gaussian elimination

System of equations and standard sequential algorithm for Gaussian elimination:

$$\sum_{j=1}^{n} a_{ij} x_j = f_i \quad \forall i = 1, \ldots, n. \tag{12.51}$$

```
for k=1,n
    for i=k+1,n
      l(i,k) = a(i,k)/a(k,k)
    endfor(i)
    for j=k+1,n
      for i=k+1,n
        a(i,j) = a(i,j) - l(i,k) * a(k,j)
      endfor(i)
    endfor(j)
endfor(k)
```

Program 12: Sequential Gaussian elimination: multiple loops

12.1 Iteration space for Gaussian elimination code

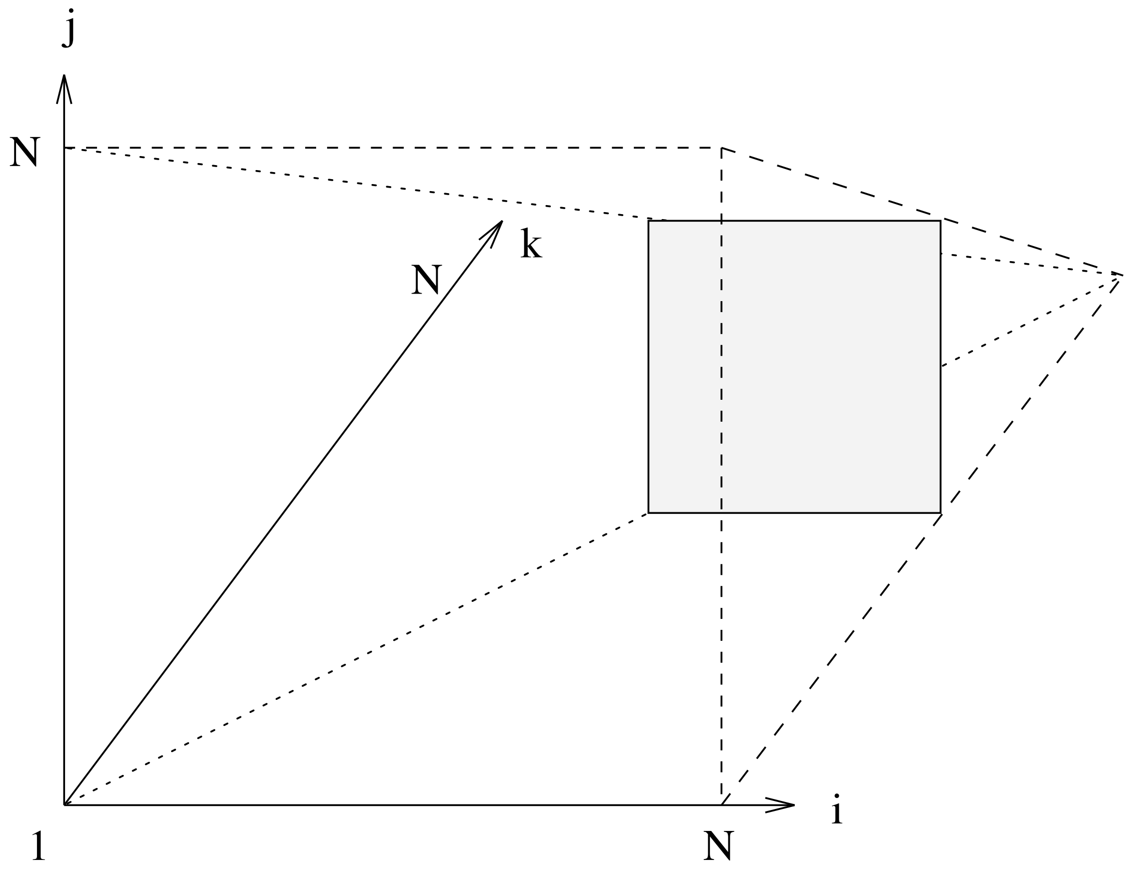


Figure 18: The iteration space for the standard sequential algorithm for Gaussian elimination forms a trapezoidal region with square cross-section in the i, j plane. Within each square (with k fixed) there are no dependencies.

12.2 What GE does

Gaussian elimination reduces the system (12.51) to a triangular system

$$\sum_{j=1}^{i} \ell_{ij} y_j = f_i \quad \forall i = 1, \ldots, n. \tag{12.52}$$

The matrix L is the lower-triangular matrix computed in Program 12 (which determines the values below the diagonal) with ones placed on the diagonal [9].

The system (12.52) can be solved by the algorithm shown in Program 13.1. Gaussian elimination performs a **matrix factorization**

$$A = LU \tag{12.53}$$

where U is the upper-triangular part of the matrix A at the end of Program 12.

The solution X in (12.51) is the solution of

$$UX = Y \tag{12.54}$$

where Y is determined by (12.52).

Solution of (12.54) determined by an algorithm analogous to Program 13.1.

12.3 Properties of GE

The dominant part of the computation in solving (12.51) is the factorization Program 12 in which L and U are determined.

The triangular system solves in (12.52) and (12.54) require less computation.

For this reason, we focus first on the parallelization of Program 12, returning to triangular systems in Section 13.

There are no loop-carried dependences in the inner-most two loops (the `i` and `j` loops) in Program 12 because $i, j > k$.

Therefore these loops can be decomposed in any desired fashion.

We now consider two different ways of parallelizing Program 12.

The algorithm in Program 12 for Gaussian elimination can be parallelized using a message-passing paradigm as depicted in Program 12.4. It is based on decomposing the matrix column-wise, and it corresponds to a decomposition of the middle loop (the `j` loop) in Program 12. A typical decomposition would be cyclic, since it provides a good load balance.

12.4 Standard parallelization of Gaussian elimination

```
for k=1,n
  if( " I own column k " )
    for i=k+1,n
      l(i,k) = a(i,k)/a(k,k)
    endfor(i)
    "broadcast" l(k+1 : n)
  else "receive" l(k+1 : n)
  endif
    for j=k+1,n ("modulo owning column j")
      for i=k+1,n
        a(i,j) = a(i,j) - l(i,k) * a(k,j)
      endfor(i)
    endfor(j)
endfor(k)
```

Program 12.4: Standard column-storage parallelization Gaussian elimination [6].

12.5 Parallel complexity of GE

We can estimate the time of execution of the standard Gaussian elimination algorithm as follows.

For each value of k, $n-k$ divisions are performed in computing the multipliers `l(i,k)`, then these multipliers are broadcast to all other processors.

Once these are received, $(n-k)^2$ multiply-add pairs (as well as some memory references) are executed, all of which can be done in parallel.

Thus the time of execution for a particular value of k is

$$c_1(n-k) + c_2\frac{(n-k)^2}{P} \tag{12.55}$$

where the constants c_i model the time for the respective basic operations.

Here, c_2 can be taken to be essentially the time to compute a “multiply-add pair” $a = a - b * c$ for a single processor.

The constant c_1 can measure the time both to compute a quotient and to transmit a word of data.

12.6 GE speedup, efficiency and scalability

Summing over k, the total time of execution is

$$\sum_{k=1}^{n-1}\left(c_1(n-k)+c_2\frac{(n-k)^2}{P}\right)\approx \tfrac{1}{2}c_1 n^2+\tfrac{1}{3}c_2\frac{n^3}{P}. \tag{12.56}$$

Time to execute this algorithm sequentially is $\frac{1}{3}c_2 n^3$. Speed-up for standard column-storage parallelization of Gaussian elimination is

$$S_{P,n}=\left(\tfrac{3}{2}\frac{\gamma}{n}+\frac{1}{P}\right)^{-1}=P\left(\tfrac{3}{2}\frac{\gamma}{n}P+1\right)^{-1} \tag{12.57}$$

where $\gamma = c_1/c_2 =$ ratio of communication to computation time. Efficiency is

$$E_{P,n}=\left(\tfrac{3}{2}\frac{\gamma P}{n}+1\right)^{-1}\approx 1-\tfrac{3}{2}\frac{\gamma P}{n}. \tag{12.58}$$

Thus the algorithm is scalable; we can take $P_n = \epsilon n$ and have a fixed efficiency of

$$\left(\tfrac{3}{2}\gamma\epsilon+1\right)^{-1}. \tag{12.59}$$

12.7 GE efficiency data

(12.58) implies that the efficiency will be the same for values of P and n which have the same ratio P/n.

In Table 3, we see this behavior born out in the columns marked "standard GE" for which the above analysis is applicable. Exercise: overlapped GE.

	16 processors		64 processors	
matrix size	standard GE	overlapped GE	standard GE	overlapped GE
$n = 128$	57%	69%		
$n = 256$	77%	87%	42%	50%
$n = 512$	91%	96%	64%	79%
$n = 1024$	$\boxed{95\%}$	98%	79%	89%
$n = 2024$			$\boxed{93\%}$	98%

Table 3: Efficiencies for two versions of Gaussian elimination for factoring full $n \times n$ matrices on an Intel iPSC.

12.8 Memory scalability for GE (not)

Total storage in Gaussian elimination is n^2 words in sequential case, and

$$M(n, P) = n^2/P + \mathcal{O}(n)$$

for the standard column-storage parallelization, leading to $\mathcal{O}(n)$ storage per processor in the case that we take $P \approx n$.

Thus, GE is not scalable with respect to memory (Definition 8.3).

If we take $P \approx n^2$ efficiency goes to zero, cf. (12.58).

From a practical point of view, it is not the memory that limits the size of n for most current computers since the execution time with $P = \mathcal{O}(n)$ is $T_P = \mathcal{O}(n^2)$.

Take $n = P = 10^8$.

Then $M = n^2/P = 10^8$ (about a gigabyte), but (for a gigaflops processor)

$$T_P = \tfrac{1}{3}10^{16}\nu s = \tfrac{1}{3}10^7 s \approx \text{one month.}$$

That is, the algorithm tends to be more time constrained than memory constrained.

12.9 Weak scaling for Gaussian elimination

GE satisfies the rule $T_{1,mn} \approx m^3 T_{1,n}$ for n large and integer m.

Suppose $N = n^2$ and n is divisible by $k =$ number of columns per processor. Then $P = n/k$ and

$$E_{P,N} := \frac{T_{1,N}}{P\,T_{P,N}} = \frac{T_{1,kPn}}{P\,T_{P,n^2}} \approx \frac{P^2 T_{1,kn}}{T_{P,n^2}}. \tag{12.60}$$

Note that kn is the size of the data on each processor in the P processor calculation.

Suppose that $m = \sqrt{kn}$ is an integer. E.g., pick $n = k^{2\nu+1}$; then $P = k^{2\nu}$. Note that $m^2 = kn = n^2/P$.

Then our weak scaling measure of efficiency is

$$\widetilde{E}_{P,N} = \frac{P^2 T_{1,m^2}}{T_{P,n^2}} = \frac{P^2 T_{1,n^2/P}}{T_{P,n^2}}. \tag{12.61}$$

That is, we compare a computation on P processors for a matrix of size $n \times n$ with a computation on 1 processor for a matrix of size $m \times m$, scaled by P^2.

13 Solving triangular systems in parallel

Key to the scalability of Gaussian elimination (GE) is the fact that the work/memory ratio $\rho_{\text{WM}} = n$.

However, triangular solution has a work/memory ratio $\rho_{\text{WM}} = 1$.

Gaussian elimination reduces a square system of equations to a traingular one (see Section 12).

The latter is (sequentially) trivial to solve, if one starts at the correct end of the triangle [9].

An algorithm for a lower triangular system (with matrix `a` like the matrix `l` generated in Program 12) is in Program 13.1.

There are now loop-carried dependences in both the `i` and `k` loops (exercise), although the inner loop is a reduction.

These loops cannot easily be parallelized, but we will see in Section 13.3 that they can be decomposed in a suggestive way (see Figure 19).

13.1 Backsolve algorithm

Solving a triangular system appears to be essentially sequential at first. In fact, solving an ordinary differential equation by a difference method is quite similar to Program 13.1.

```
x(1)=f(1)/a(1,1)
for i=2,n
    x(i)=f(i)
    for j=max(1,i-w), i-1
      x(i) = x(i) - a(i,j)*x(j)
    endfor(j)
    x(i)=x(i)/a(i,i)
endfor(i)
```

Program 13.1: Sequential solution algorithm for a banded lower-triangluar system.

For generality, we have presented the code for a banded triangular system in Program 13.1 where `w` is the band width.

If $\mathtt{w} = \mathtt{n}$ then we have the full matrix case.

13.2 Triangular scenarios

Solving a triangular system just once: Typically occurs when the triangular system arises via matrix factorization. Since the time for a single triangular system solve is minor compared with matrix factorization, the cost of the triangular system is negligible. Even modest efforts at parallelization may be useful.

Solving a triangular system with many right-hand sides, all available at one time: Due to the large amount of data (the right-hand sides), there are special opportunities for parallelism which make triangular solution appear similar in character to the factorization problem.

Solving a triangular system with many right-hand sides, only one of which is available at a time: This occurs when solving time-dependent problems or non-linear problems in which the triangular system is part of an inner loop. The right-hand side changes as the outer loop progresses. In this case, any initial pre-processing of the triangular system can be amortized. The issue is just producing a solution as quickly as possible.

This is most challenging situation to parallelize efficiently, and therefore we will concentrate on this case.

13.3 The “toy duck” algorithm

Let us begin by considering a straight-forward parallelization of the standard algorithm shown in Program 13.1; similar to overlapped GE.

We will compute the same arithmetic operations in the same order, but we will organize them in a way that presents independent tasks that can be done in parallel.

We describe the ℓ-th stage for $\ell > 1$ assuming all needed information has been initialized; Figure 19 depicts the various computations done at this stage.

Initialization phase left as an exercise.

We assume that k is a parallelization parameter that will be specified later.

It will be a positive integer less than $\frac{1}{2}w$.

At the ℓ-th step, we assume that all x_i's for $i \leq k\ell$ have already been computed and distributed to each processor.

At each step, solution values x_i will be computed for k new indices i and distributed to each processor.

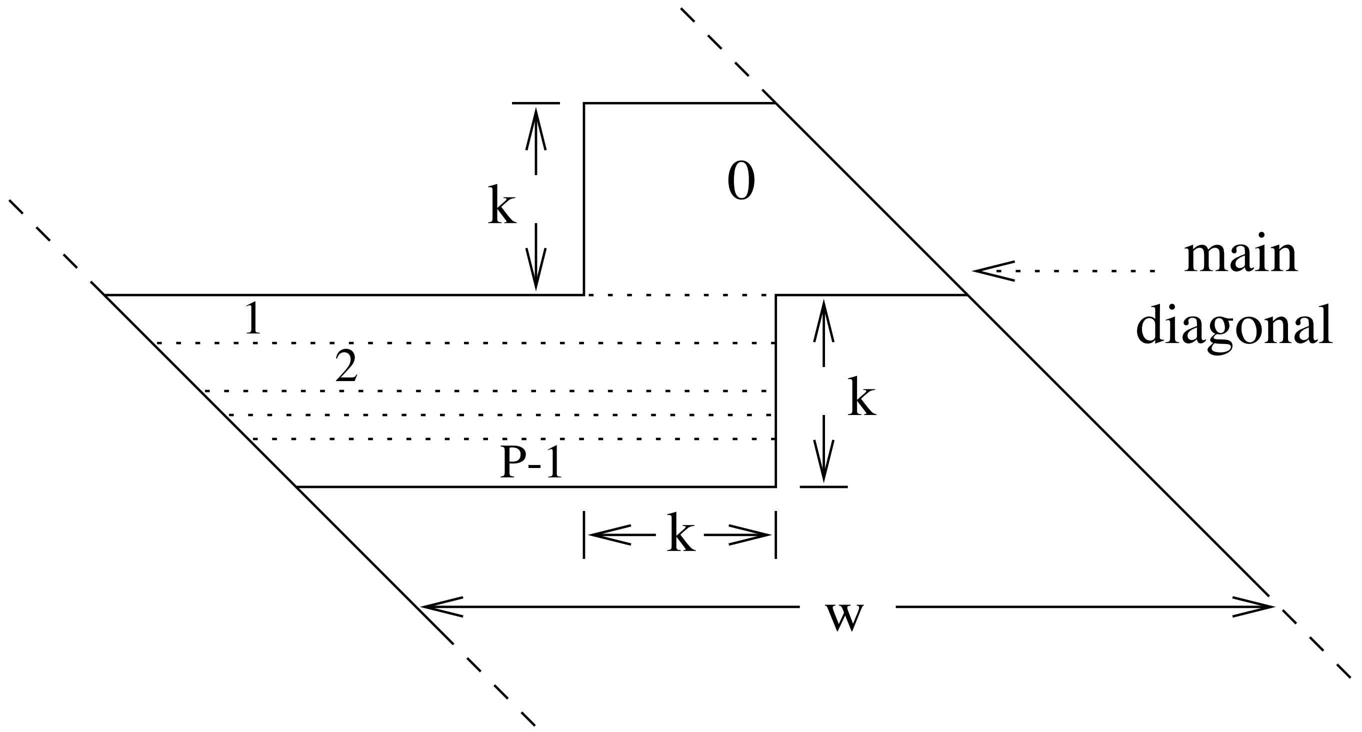


Figure 19: Schematic of "toy duck" parallelization of a banded, triangular matrix equation solution algorithm.

Processors 1 through $P-1$ compute

$$b_i^\ell := \sum_{j=\min(i-w,1)}^{k\ell} a_{i,j}x_j \quad \forall i = 1+(k+1)\ell, \ldots, (k+2)\ell. \tag{13.62}$$

13.4 Typical step in toy duck

In the simplest case (as we now assume) we will have

$$k = \nu(P-1) \tag{13.63}$$

for some integer $\nu \geq 1$, so that each processor ≥ 1 computes ν different b_i's using previously computed x_j's.

Note that this requires access to the (previously computed) values x_j for $j \leq k\ell$.

Simulaneously, processor 0 computes $x_{k\ell+1}, \ldots, x_{(k+1)\ell}$ by the standard algorithm, namely,

$$x_i = a_{i,i}^{-1}\left(f_i - b_i^{\ell-1} - \sum_{j=(k-1)\ell+1}^{i-1} a_{i,j}x_j\right) \quad \forall k\ell < i \leq (k+1)\ell. \tag{13.64}$$

We can assume that $a_{i,i}^{-1}$ has been precomputed and stored, if desired.

At end of this step, processor 0 sends $x_{k\ell+1}, \ldots, x_{(k+1)\ell}$ to other processors, and other processors send $b_{(k+1)\ell+1}, \ldots b_{(k+2)\ell}$ to processor 0.

13.5 Analysis of toy duck

This completes the ℓ-th step for $\ell > 1$.

These steps involve a total of $2k^2$ MAPs, where MAPs stands for **multiply-add pairs**.

Load balance in (13.62) can be achieved in a number of ways.

If $\nu = 2$, then perfect load balance is achieved by having processor 1 doing the first and last row, processor 2 doing the second and penultimate row, and so on.

The total number of operation to compute (13.62) is

$$k(w-k) - \tfrac{1}{2}k^2 = kw - \tfrac{3}{2}k^2 \tag{13.65}$$

MAPs, where MAPs stands for "multiply-add pairs."

Thus the time estimate for (13.62) is proportional to

$$\left(w - \tfrac{3}{2}k\right)\frac{k}{P-1} \tag{13.66}$$

time units, where the unit is taken to be the time required to do one multiply-add pair.

13.6 Continued analysis of toy duck

Processor zero does $\frac{3}{2}k^2$ MAPS, and thus the total time for one stage of the program is proportional to

$$\max\left\{\tfrac{3}{2}k^2, \left(w - \tfrac{3}{2}k\right)\frac{k}{P-1}\right\}. \tag{13.67}$$

These are *balanced* if

$$\tfrac{3}{2}k^2 = \left(w - \tfrac{3}{2}k\right)\frac{k}{P-1} \tag{13.68}$$

which reduces to having

$$P = \tfrac{2}{3}\frac{w}{k}. \tag{13.69}$$

Recalling our assumption (13.63), we find that

$$P(P-1) = \tfrac{2}{3}\frac{w}{\nu} \tag{13.70}$$

Optimal P depends only on the band width w and not on n. Algorithm not scalable in usual sense (Definition 8.2) if w remains fixed independently of n.

13.7 Scaling of toy duck

Total amount of data communicated at each stage is $2k$ words.

(13.68) implies that computational load is proportional to k^2, so this algorithm is scalable if $w \to \infty$ as $n \to \infty$ (exercise).

The case of a full matrix corresponds to $w = n$.

The "toy duck" algorithm has substantial parallelism in this case.

For fixed ν, (13.70) implies that P is proportional to $\sqrt{w}$, and this in turn implies that k is proportional to $\sqrt{w}$.

The amount of memory per processor is directly proportional to the amount of work per processor, so this is proportional to k^2, and hence w, in the balanced case (13.68).

13.8 A block inverse algorithm

The following algorithm can be found in [15]. Let us write the lower triangular matrix L as a block matrix. Suppose that $n = ks$ for some integers k and s.

$$\begin{pmatrix} L_1 & 0 & 0 & 0 & 0 & 0 \\ R_1 & L_2 & 0 & 0 & 0 & 0 \\ 0 & R_2 & L_3 & 0 & 0 & 0 \\ 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 0 & R_{k-2} & L_{k-1} & 0 \\ 0 & 0 & 0 & 0 & R_{k-1} & L_k \end{pmatrix} \tag{13.71}$$

A triangular matrix is invertible if and only if its diagonal entries are not zero (see [9] or just apply the solution algorithm in Program 13.1). Thus any sub-blocks on the diagonal will be invertible as well if L is, as we now assume. That is, each L_i is invertible, no matter what choice of k we make.

Let D denote the block diagonal matrix with blocks $D_i := L_i^{-1}$. If we premultiply D times L, we get a simplified matrix:

$$DL = \begin{pmatrix} I_s & 0 & 0 & 0 & 0 & 0 \\ G_1 & I_s & 0 & 0 & 0 & 0 \\ 0 & G_2 & I_s & 0 & 0 & 0 \\ 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 0 & G_{k-2} & I_s & 0 \\ 0 & 0 & 0 & 0 & G_{k-1} & I_s \end{pmatrix} \tag{13.72}$$

where I_s denotes an $s \times s$ identity matrix, and the matrices G_i arise by solving

$$L_{i+1}G_i = R_i, \quad i = 1, \ldots, k-1. \tag{13.73}$$

The original system $Lx = f$ is changed to $(DL)x = Df$. Note that we can write Df in block form with blocks (or segments) b_i which solve

$$L_i b_i = f_i, \quad i = 1, \ldots, k. \tag{13.74}$$

13.9 Block inverse details

The blocks L_i in (13.74) are $s \times s$ lower-triangular matrices with bandwidth w, so the band algorithm Program 13.1 is appropriate to solve (13.74).

Depending on the relationship between the block size s and the band width w, there may be a certain number of the first columns of the matrices R_i which are identically zero.

In particular, one can see (exercize) that the first $s - w$ columns are zero.

Due to the definition of G_i, the same must be true for them as well (exercize).

Let $\widehat{G}_i$ denote the right-most w columns of $G_i = (0 \quad \widehat{G}_i)$

let M_i denote the top $s - w$ rows of $\widehat{G}_i$ and

let H_i denote the bottom w rows of $\widehat{G}_i$.

Further, split b_i similarly, with

u_i denoting the top $s - w$ entries of b_i and

v_i denoting the bottom w entries of b_i.

13.10 Block inverse notation

We may then write the blocks (strips) x_i of the solution vector in two corresponding parts: y_i denoting the top $s-w$ entries of x_i and z_i denoting the bottom w entries of x_i.

The notation is summarized in

$$\widehat{G}_i = \begin{pmatrix} M_i \\ H_i \end{pmatrix} \qquad x_i = \begin{pmatrix} y_i \\ z_i \end{pmatrix} \qquad b_i = \begin{pmatrix} u_i \\ v_i \end{pmatrix} \qquad \begin{matrix} \}s-w \\ \}w \end{matrix} \tag{13.75}$$

and the dimensions of M_i and H_i are

$$\underbrace{M_i}_{w} \quad \} \, s-w \qquad\qquad \underbrace{H_i}_{w} \quad \} \, w \tag{13.76}$$

13.11 Block inverse reduction

All of these quantities have now simple relationships. First of all we have

$$y_1 = u_1, \quad z_1 = v_1 \tag{13.77}$$

we can inductively determine the z_i's by

$$z_{i+1} = v_{i+1} - H_i z_i \quad \forall i = 1, \ldots, k-1 \tag{13.78}$$

Then we can separately determine the y_i's by

$$y_{i+1} = u_{i+1} - M_i z_i \quad \forall i = 1, \ldots, k-1 \tag{13.79}$$

There are no dependences in (13.79), but (13.78) appears at first to be sequential.

However, if w is sufficiently large, there is an opportunity for parallelism in each iteration of (13.78).

Moreover, (13.78) can be written as a lower-triangular system itself, and we describe an appropriate parallel solution algorithm.

There are two cases to distinguish in applying the above algorithm. One is the case where a system is solved only once, and the systems (13.73) become a major part of the computation. The other is the case where a system is solved many times, and the cost of solving the systems (13.73) can be amortized since they need be solved only once.

The primary amount of work in (13.74) is

$$k\left(ws - \tfrac{1}{2}w^2\right) = wn - \tfrac{1}{2}\frac{w^2 n}{s} \tag{13.80}$$

MAPs, whereas the primary amount of work in (13.78) is

$$w^2(k-1) = \frac{w^2(n-s)}{s} \tag{13.81}$$

MAPs, and the primary amount of work in (13.79) is

$$w(s-w)(k-1) = \frac{w(s-w)(n-s)}{s} = wn - \frac{w^2 n}{s} - w(s-w) \tag{13.82}$$

MAPs.

13.12 Block inverse analysis

The sum of (13.80), (13.81) and (13.82) is nearly $2nw$, twice the amount of work in the sequential case (Program 13.1).

The amount of parallelism in (13.78) is complex to assess [16], but once all the z_i's are computed (and appropriately distributed), (13.79) can be done (trivially) in parallel.

Moreover, (13.80) can also be computed trivially in parallel.

If (13.78) is computed sequentially, then

$$T_P \geq wn\left(\frac{w}{s} + \frac{1}{P}\left(1 - \frac{w}{s}\right)\right) = wn\left(\frac{wP}{n} + \frac{1}{P}\left(1 - \frac{wP}{n}\right)\right) \qquad (13.83)$$

if $P = k$. Therefore

$$E_P^{-1} \geq \frac{wP^2}{n} + \left(1 - \frac{wP}{n}\right) \qquad (13.84)$$

Taking $P \leq \sqrt{n/w}$ would be required for a scalable algorithm.

14 Parallel prefix

Assume the lower triangular matrix L is already of the form

$$x_i + H_{i-1}x_{i-1} = v_i \quad \forall i = 1, \ldots, 2^\kappa \tag{14.85}$$

where all undefined quantities (H_0, H_1, x_0, etc.) are set implicitly to zero.

Then the relation (14.85) between subsequent blocks of the solution x can be written in an extended (specifically, $(w+1) \times (w+1)$) block matrix form as

$$\begin{pmatrix} x_i \\ 1 \end{pmatrix} = \begin{pmatrix} -H_{i-1} & v_i \\ 0_w & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix} =: h_i \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix} \tag{14.86}$$

where 0_w denotes a row vector of zeros of length w and h_i denotes the indicated $(w+1) \times (w+1)$ matrix.

Parallel prefix applies to any associative operation $\oplus$ and produces

$$g_i := h_i \oplus h_{i-1} \oplus \cdots \oplus h_1, \; i = 1, \ldots, 2^\kappa$$

using a binary tree [12].

14.1 Parallel prefix application

Applying induction, we thus see that

$$\begin{pmatrix} x_i \\ 1 \end{pmatrix} = g_i \begin{pmatrix} 0_w \\ 1 \end{pmatrix} \tag{14.87}$$

where $g_i := h_i \oplus h_{i-1} \oplus \cdots \oplus h_1$ and $\oplus$ is matrix multiplication for $(w+1) \times (w+1)$ matrices.

Therefore, we may use parallel prefix to compute the $(w+1) \times (w+1)$ matrices g_i and then recover the solution from (14.87).

Parallel prefix requires 2κ steps involving matrix multiplication of two $(w+1) \times (w+1)$ matrices, and communication (along edges of a tree) which send a $(w+1) \times (w+1)$ matrix at each step.

The computation is $\mathcal{O}(\kappa w^3)$ and the communication is $\mathcal{O}(\kappa w^2)$.

14.2 Parallel prefix not good alone

In the special case that $s = w$ in the algorithm in Section 13.8, the matrices M_i disappear and $G_i = H_i$.

We can thus think of (14.85) as a special case of (13.71).

Parallel prefix applied to this problem requires $\mathcal{O}(\kappa w^3)$ time compared with nw time in the sequential case.

A speedup of $n/\kappa w^2$ is the largest possible with parallel prefix alone. The efficiency would satisfy (recall that $n = Pw$)

$$E_P^{-1} = \frac{P\kappa w^3}{nw} = \frac{P\kappa w^2}{n} = \kappa w = w \log_2 P \qquad (14.88)$$

which does not imply scalability.

Thus parallel prefix alone is not a scalable triangular solver.

However, we now look at combining it with other algorithms.

14.3 Combining with parallel prefix

If (13.78) is computed by parallel prefix then time estimate in (13.83) becomes

$$T_P \approx wn\left(\frac{w^2}{n}\log_2 P + \frac{1}{P}\left(2 - \frac{3wP}{2n} + \frac{w}{n} - \frac{1}{P}\right) + \gamma\frac{w}{n}\log P\right), \quad (14.89)$$

where $\gamma =$ ratio of communication to computation time in parallel prefix. Thus

$$\begin{aligned} E_P^{-1} &\approx \left(\frac{w^2}{n} + \frac{\gamma w}{n}\right) P\log_2 P + \left(2 - \frac{3wP}{2n} + \frac{w}{n} - \frac{1}{P}\right) \\ &= \frac{w^2 + \gamma w}{s}\log_2 P + \left(2 - \frac{3w}{2s} + \frac{w}{n} - \frac{1}{P}\right). \end{aligned} \quad (14.90)$$

We assume $k = P = 2^\kappa$.

Theorem 14.1 *The algorithm in Section 13.8, where (13.78) is solved by parallel prefix, is scalable for* $P\log_2 P \le cn/(w^2 + \gamma w)$ *for any fixed* $c < \infty$. *In particular, we must take* $s \ge \max\{w, \epsilon(w^2 + \gamma w)\log_2 P\}$ *for some* $\epsilon > 0$.

The expression $w^2 + \gamma w$ in the theorem can be improved to $w^2 + \beta w + \lambda$ where λ is the latency and β measures bandwidth.

15 Scalable solution of non-linear time-dependent systems

Mark Maienschein-Cline

Department of Chemistry

University of Chicago

and

L. Ridgway Scott

15.1 Initial value problems

Finite difference methods for solving initial value problem for an ordinary differential equation exhibit limited natural parallelism.

However, for linear systems there are scalable parallel algorithms in which the domain decomposition is in the time domain.

Such techniques are based on scalable methods for solving banded triangular linear systems of equations and have been known for some time (cf. [15, 12]).

What can provide the increasing data size needed for such scalability is a long time interval of integration.

Indeed, there are many simulations in which the primary interest is a very long time of integration.

For example, there is a celebrated simulation of the villin headpiece by molecular dynamics [5].

15.2 An example

We begin with a very simple example here motivated by a swinging pendulum.

To a first approximation, the position $u(t)$ of the pendulum satisfies a differential equation

$$\frac{d^2u}{dt^2} = f(u) \tag{15.91}$$

with initial conditions provided at $t = 0$:

$$u(0) = a_0,\ u'(0) = a_1 \tag{15.92}$$

for some given data values a and a_1.

The variable u can be taken to denote the angle that the pendulum makes compared with the direction of the force of gravity.

Then $f(u) = mg \sin u$ where g is the gravitational constant and m is the mass of the weight at the end of the pendulum.

(We are ignoring the mass of the rod that holds this weight.)

15.3 Dicretization

We can approximate via a central difference method to get a recursion relation

$$u_{n+1} = 2u_n - u_{n-1} - \tau f(u_n) \tag{15.93}$$

where $\tau := \Delta t^2$.

If the displacements of the pendulum position from vertical are small, then we can use the approximation $\sin u \approx u$ to get a linear equation

$$\frac{d^2u}{dt^2} = mgu. \tag{15.94}$$

In this case, the difference equations become a linear recursion relation of the form

$$u_{n+1} = (2 - \tau mg)\, u_n - u_{n-1}. \tag{15.95}$$

The initial conditions (15.92) provide starting values for the the recursion. For example, we can take

$$u_0 = a \quad \text{and} \quad u_{-1} = a_0 - a_1 \Delta t. \tag{15.96}$$

This allows us to solve (15.95) for $n \geq 0$.

15.4 Dicretization as a linear system

The recursion (15.95) corresponds to a banded, lower triangular system of equations of the form

$$Lu = b \tag{15.97}$$

where the bandwidth of L is $w = 2$, the diagonal and subsubdiagonal terms of L are all one, and the subdiagonal terms of L equal $\tau mg - 2$.

The right-hand side g is of the form

$$b_1 = (1 - \tau mg)\,a_0 + a_1 \Delta t \quad \text{and} \quad b_2 = -a_0 \tag{15.98}$$

and $b_i = 0$ for $i \geq 3$.

Thus we can solve this by the scalable parallel algorithms which are discussed in Section 13.

15.5 Nonlinear systems

When f is not linear, such algorithms are not directly applicable.

We can formulate a set of equations to define the entire vector of values u_i as an ensemble, but it is no longer a linear equation.

We can write it formally as

$$F(U) = 0 \tag{15.99}$$

where F is defined by

$$F(U) = L^0 U + \tau m g \phi(U) - b \tag{15.100}$$

with L^0 the same as L above with $\tau = 0$, that is L^0 is a lower triangular matrix with bandwidth $w = 2$, the diagonal and subsubdiagonal terms of L^0 are all one, and the subdiagonal terms of L^0 equal -2. The function ϕ thus contains all of the nonlinearity and has the simple form

$$\phi(U)_{ij} = \delta_{j,i-1} \sin u_{i-1} \tag{15.101}$$

where $\delta_{i,j}$ is the Kronecker delta.

15.6 Newton's method

The Newton-Raphson method can be written in the form

$$F'(U^n)\left(U^{n+1} - U^n\right) = -F(U^n). \tag{15.102}$$

where F' is the Jacobian matrix of all partial derivatives of F with respect to the vector U.

To see how this works, let us return to the pendulum problem. In the case of the pendulum, we have F defined by (15.100). It takes a careful look at the definition, but it is not hard to see that the Jacobian matrix for a linear function of the form $U \to L^0 U$ is the matrix L^0. Thus $J_F(U) = L^0 + \tau m g J_\phi(U)$. With ϕ of the form (15.101), it can be shown that

$$J_\phi(U)_{ij} = \delta_{j,i-1}\phi'(u_{i-1}) = \delta_{j,i-1}\cos u_{i-1}. \tag{15.103}$$

The Jacobian has the same form as the original matrix L. In fact, if F is linear, Newton's method is equivalent to just solving the system (15.97) and converges in one step.

15.7 Linearized initial value problem

There is another way to interpret the Newton algorithm for solving the initial value problem.

Suppose that we have an approximate solution u to (15.91) which satisfies the initial conditions (15.92), and define the residual $R(u)$ by

$$R(u) := \frac{d^2u}{dt^2} - f(u) \tag{15.104}$$

which is a function of t defined on whatever interval u is defined on.

We can apply Newton's method (in the appropriate infinite dimensional setting) to solve

$$R(u) = 0. \tag{15.105}$$

Let us derive the resulting equations by an elementary approach.

15.8 Newton step derivation

Suppose that we try to add a perturbation v to u to get it to satisfy (15.91) (more) exactly. We use a Taylor expansion to write

$$f(u+v) = f(u) + vf'(u) + \mathcal{O}(v^2) \tag{15.106}$$

Thus the sum $u+v$ satisfies an initial value problem of the form

$$\frac{d^2(u+v)}{dt^2} - f(u+v) = \frac{d^2v}{dt^2} - vf'(u) + R(u) + \mathcal{O}(v^2). \tag{15.107}$$

With (15.107) as motivation, we now *define* v by solving the initial value problem

$$\frac{d^2v}{dt^2} = vf'(u) - R(u) \tag{15.108}$$

with initial conditions provided at $t=0$:

$$v(0) = 0,\ v'(0) = 0. \tag{15.109}$$

Then the Newton step for solving (15.105) is the solution v of (15.108-15.109).

15.9 Newton versus discretization

Now (15.102) can now be seen as just a discretization of the initial value problem (15.108).

This can be depicted in a diagram:

$$\begin{tikzcd}
\text{ODE } (15.91) \arrow[r, "\text{Newton}"] \arrow[d, "\delta"] & \text{linearized ODE } (15.108 - 15.109) \arrow[d, "\delta"] \\
\text{diff. meth. } (15.95 - 15.96) \arrow[r, "\text{Newton}"] & \text{discrete Newton method } (15.102),
\end{tikzcd} \tag{15.110}$$

where the symbol δ denotes time discretization.

Thus we expect that, in the limit of small time step, the number of Newton iterates as a function of time would approach a limit.

15.10 Getting Newton started

Newton's method (15.102) converges rapidly once you get close to a solution, but how do you get close in the first place? This does not have a good answer in general, but we can indicate one type of approach here. Suppose that we had a simplified approximation $\tilde{f}$ to f and we solved

$$\frac{d^2u}{dt^2} = \tilde{f}(u) \tag{15.111}$$

exactly, together with the initial conditions (15.92). For example, with $f(u) = \sin u$ we might have $\tilde{f}(u) = u$ as an approximation. This makes (15.111) linear, and in this simple case we can even solve the equation in closed form (in terms of sines and cosines). It is much faster to evaluate $\tilde{f}(u)$ in this case than it is $f(u)$, so the computation of the initial guess would be much less costly.

If we use the solution u to (15.111) as the starting guess for Newton's method, then the initial residual $R(u)$ has a simple interpretation. We can express it simply as

$$R(u) = \frac{d^2u}{dt^2} - f(u) = \tilde{f}(u) - f(u). \tag{15.112}$$

Thus the size of the residual is simply related to the error in approximation of f by $\tilde{f}$. Since the first Newton step v satisfies (15.108), we can bound the size of v in terms of $R(u)$, and hence $f - \tilde{f}$. More precisely, (15.108) becomes

$$\frac{d^2v}{dt^2} = vf'(u) - \left(\tilde{f}(u) - f(u)\right). \tag{15.113}$$

Since v is zero at the start (see (15.92)), it will be small for at least some reasonable interval of time. The size of v can be predicted as u is computed, and the process can be stopped if the prediction gets too large. Thus there is a natural way to control the size of the Newton step in this case.

It would be possible to approximate on a coarser mesh as well. That is, we could use a larger Δt in (15.93) (recall that $\tau = \Delta t^2$). It would then be necessary to interpolate onto a finer mesh to define the residual appropriate for (15.108) (or equivalently in (15.102) in the discretized case). In this way, a multi-grid approach could be developed in the time variable.

15.11 Application to orbit simulation

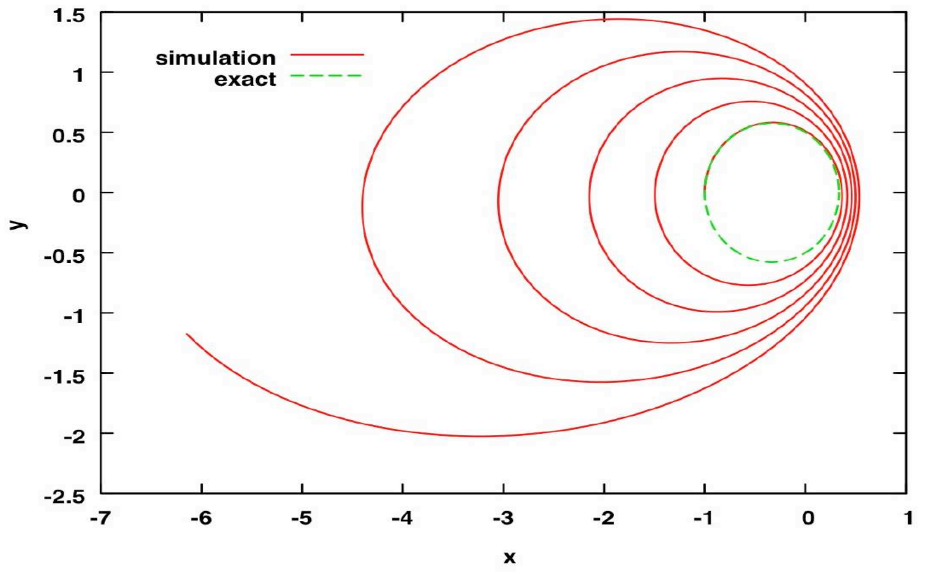
We consider the simple two body problem of planetary motion, with one body (the "sun") fixed at the origin $(0, 0)$. Thus the number of unknown particle positions is $N = 1$, and the number of dimensions $D = 2$. Let the state be $z = (x, y)$ (or (r, θ) in polar coordinates). The particle moves on a potential surface $U = G/r = \frac{G}{(x^2+y^2)^{1/2}}$, so the force field is

$$F(x, y) = (F_1(x, y), F_2(x, y)) = \left(\frac{-Gx}{(x^2 + y^2)^{3/2}}, \frac{-Gy}{(x^2 + y^2)^{3/2}}\right).$$

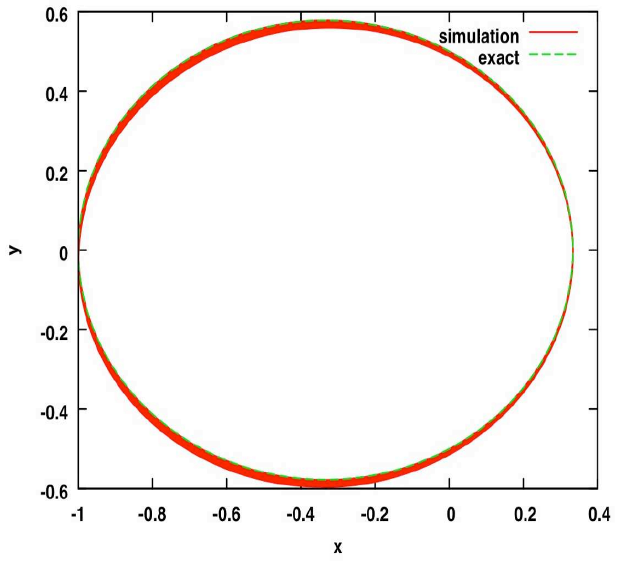
Note as well that

$$J_F(x, y) = \begin{pmatrix} \frac{G(2x^2-y^2)}{(x^2+y^2)^{5/2}} & \frac{3Gxy}{(x^2+y^2)^{5/2}} \\ \\ \frac{3Gxy}{(x^2+y^2)^{5/2}} & \frac{G(2y^2-x^2)}{(x^2+y^2)^{5/2}} \end{pmatrix} \qquad (15.114)$$

Additionally, any given orbit can be mapped onto one with the gravitational source at $(0, 0)$ and initial conditions $(x_0, y_0) = (-1, 0)$, $(\dot{x}_0, \dot{y}_0) = (1, 0)$, and the parameter G variable. The previous values are used as initial conditions.



(a)



(b)

Figure 20: Integration of orbit system, with $G = 4$, gravity from origin, $\triangle t = 0.01$, run for 50 time units, initial condition $(x_0, y_0) = (-1, 0)$, $(\dot{x}_0, \dot{y}_0) = (0, 1)$. Exact solution in polar coordinates is $r = \frac{p}{1+e\cos\theta}$, where $p = (\vec{r} \times \vec{v})/G$, a conserved quantity ($p = 1/G$ here), and $e = 1 - \frac{2}{r_a/r_p+1}$, where r_a is the farthest the the orbit gets from the origin and r_p is the closest; $e = p - 1$ here. The period is $2\pi\sqrt{r_a^3/G}$. (a) Euler scheme, (b) Verlet.

15.12 Computational results for the orbit problem

Figure 21 depicts eight iterations of Newton's method for the orbit problem, each one offset artificially along the time axis.

The dashed (green) line indicates the exact orbit, and the solid (red) line indicates the computed Newton step.

The initial step has a constant state, as indicated by the straight line for the left most pair of curves.

Subsequent iterates follow the orbit more and more, but the first few eventually move away from the orbit.

The fifth iterate agrees with the exact orbit to graphical accuracy, and the remaining iterates home in to the orbit to a tolerance of 10^{-10}.

Whether we require only 5 iterations or insist on 8 iterations, the Newton strategy provides a substantial amount of parallelism for the orbit problem.

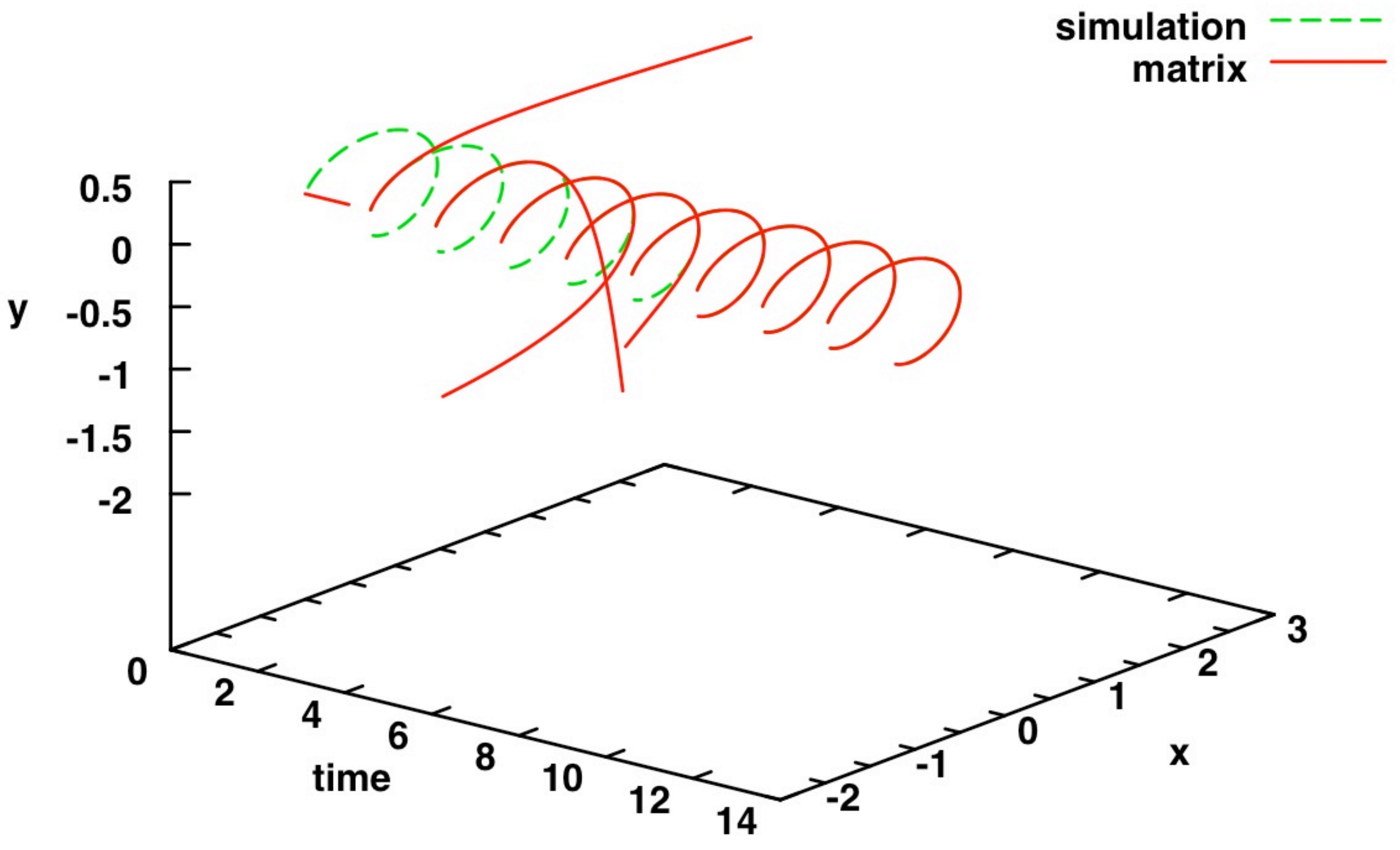


Figure 21: Orbit problem: period = 1.36. 1000 times steps with $\Delta t = 0.001$. Converged in 8 iterations.

The computations in Figure 21 are carried out for a longer time integration in Figure 22.

It appears that for longer times, the number of iterations I of Newton's method required for convergence grows linearly with the number N of time steps.

Figure 22 suggests that

$$I \approx 0.004N = 4\Delta t\,\tau, \tag{15.115}$$

where τ is the time of integration.

Figure 23 confirms this for other time steps and supports the theoretical prediction in (15.110).

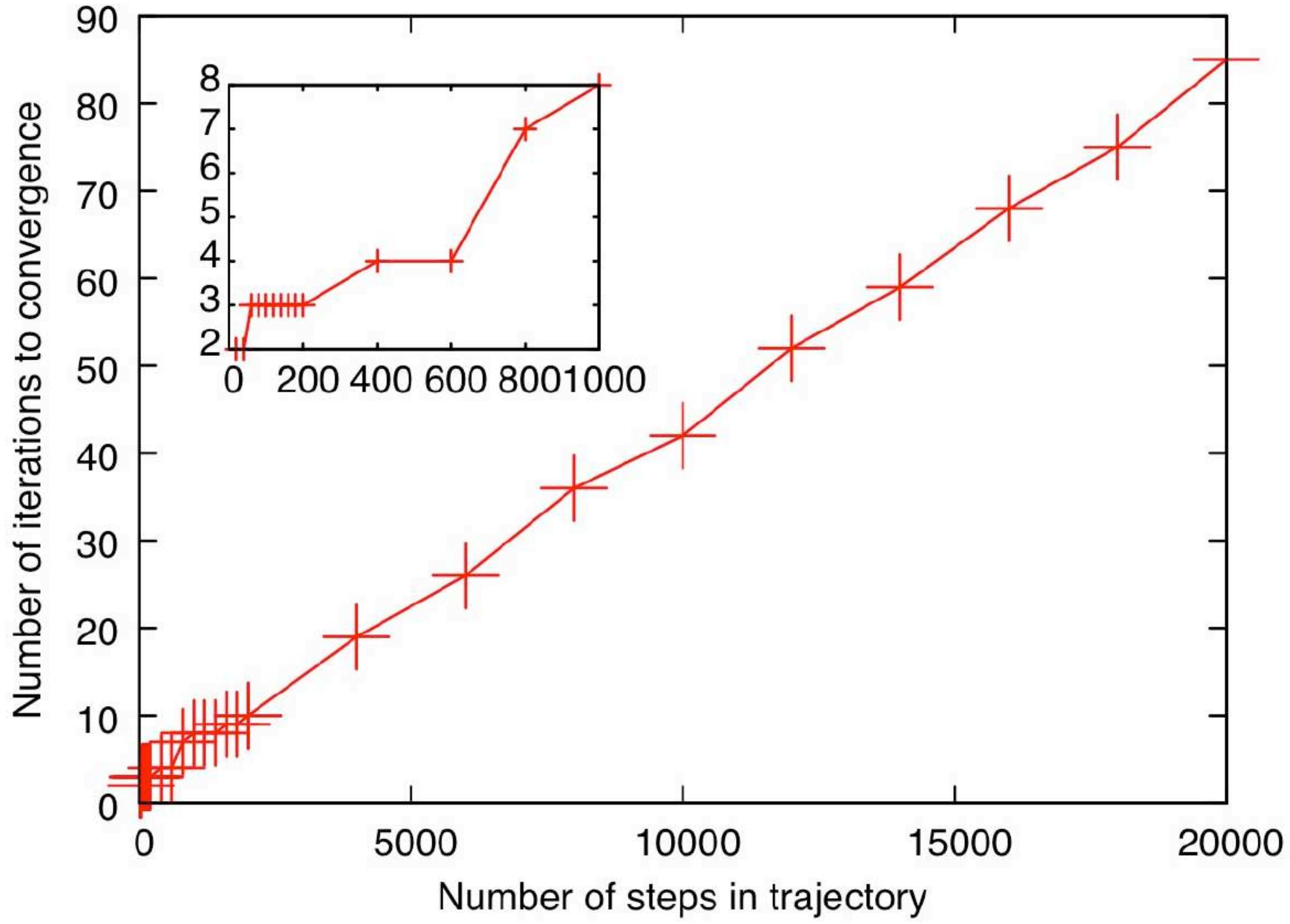


Figure 22: Summary of computations depicted in Figure 21 for different lengths of computation. Vertical axis is number of Newton iterations required to converge.

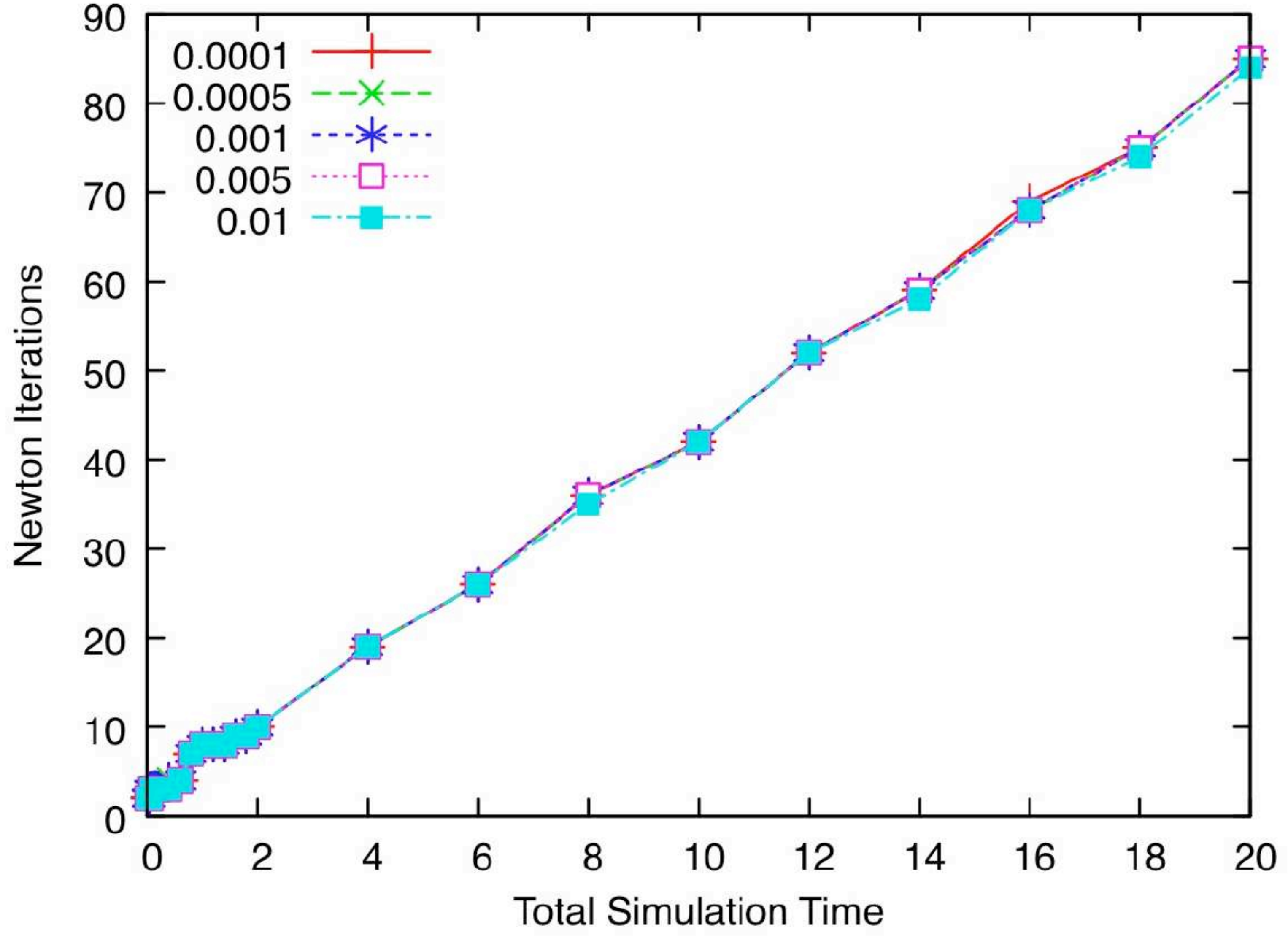


Figure 23: Orbit simulations for different time steps. Horizontal axis: simulation time. Vertical axis: number of Newton iterations. Constant initial guess.

15.13 Parallel computation of the orbit problem

The parallel tridiagonal linear solves can be performed efficiently with

$$P \log P = aN \tag{15.116}$$

for some constant a, with the parallel time $\widetilde{T}_{P,N} \approx cN/P$ for each Newton step for a constant c.

The constant a is ours to adjust; the larger we make it, the larger the granularity of the parallel solution algorithm, although the smaller is P.

Thus we can assume that c does not increase when a is increased.

Assume that the sequential problem takes about $T_{1,N} \approx cN$ time for simplicity.

The number of Newton iterations I required is $bN = \beta \Delta t\, \tau$, as suggested by (15.115), so the asymptotic total parallel execution time (for N large) is

$$T_{P,N} \approx \frac{IcN}{P} = \frac{bcN^2}{P} = \frac{b}{a} T_{1,N} \log P. \tag{15.117}$$

15.14 Speedup of the orbit problem

This says that the speedup would be estimated by the relation

$$S_{P,N} \approx \frac{a}{b \log P} = \frac{a}{\beta \Delta t \, \tau}. \tag{15.118}$$

In particular, this says that the speedup can be arbitrarily large for fixed τ as $\Delta t \to 0$.

On the other hand, (15.118) also says that the efficiency would be restricted by the relation

$$E_{P,N} \approx \frac{a}{bP \log P} = \frac{1}{bN} = \frac{1}{I}, \tag{15.119}$$

consistent with the observation that our algorithm simply duplicates the sequential algorithm I times.

Note that the adjustable parameters (a and P) drop out of the relation (15.119) for efficiency.

15.15 Smarter start

The data presented so far relate to an initial guess for the Newton iteration in which the initial solution is just constant in time.

It is remarkable that this works at all, but it would not be surprising that there are smarter ways to start.

The number of possible ways to do so is unlimited, so we experimented with just a simple approach: solving sequentially with a larger time step.

We chose a time step ten times larger to produce the initial guess.

Figure 24 shows that benefit of coarser time step improves when ultimate goal is to approximate with a smaller time step.

In view of (15.110), we can interpret these data as being equivalent to solving the continuous Newton problem with an initial guess corresponding to a discretization using increasingly finer time steps.

Thus it is not surprising that the curves in Figure 24 appear to tend to a constant as the time step is decreased.

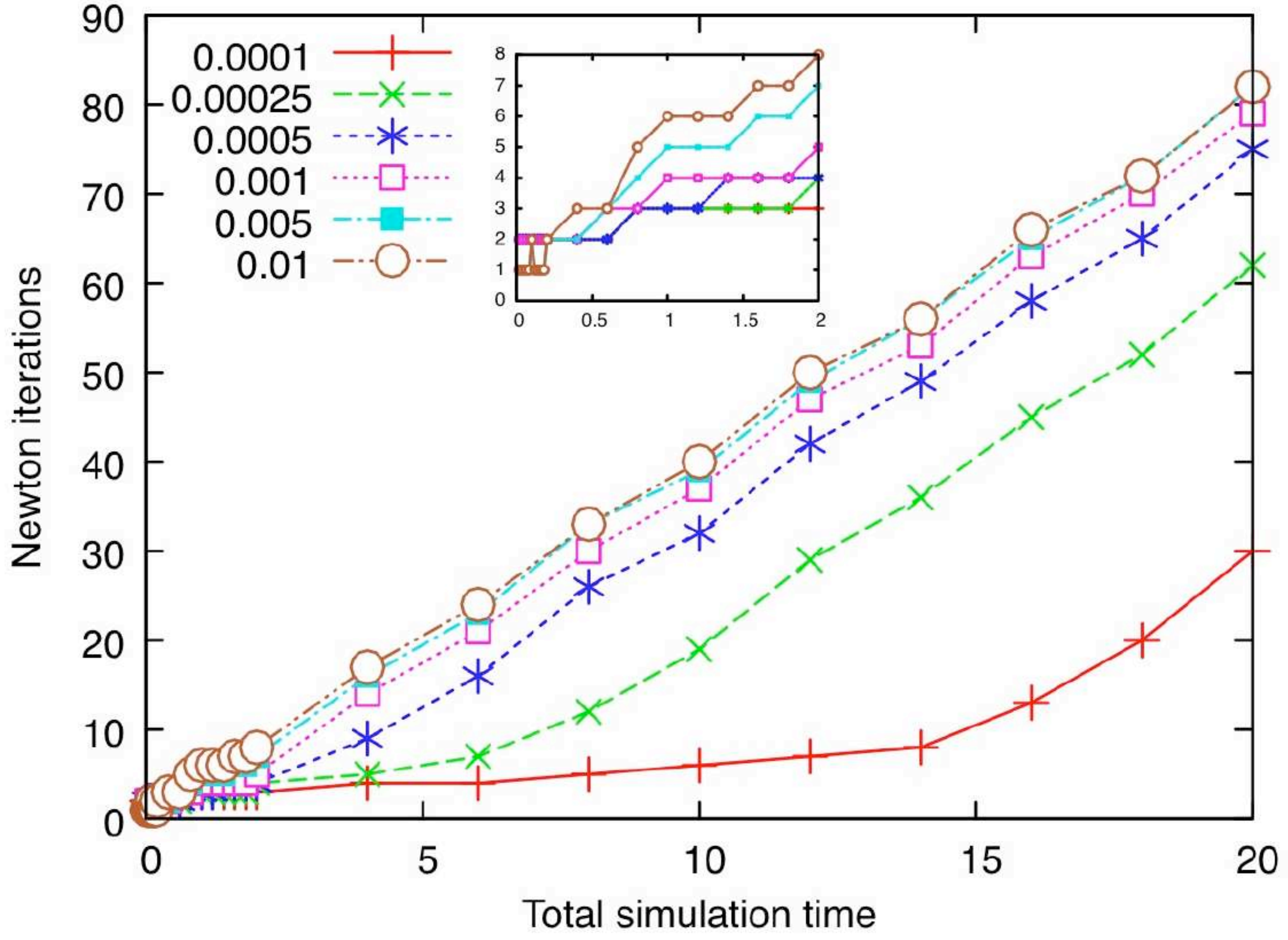


Figure 24: Orbit simulations for different time steps. Horizontal axis: simulation time. Vertical axis: number of Newton iterations. Initial guess is solution using a (ten-times) coarser time step.

15.16 Lorenz attractor

Another example was considered, the Lorenz system

$$\begin{aligned}\dot{x} &= \sigma(y - x)\\ \dot{y} &= rx - y - xz\\ \dot{z} &= xy - bz,\end{aligned} \tag{15.120}$$

where

$$\sigma = 10,\ b = 8/3,\ \text{ and } r = 28. \tag{15.121}$$

Typical behavior is shown in Figure 25 which depicts two solutions that start near each other but quickly diverge.

However, the solutions exhibit a type of near-periodic behavior, cycling around two attractors indicated by plus signs.

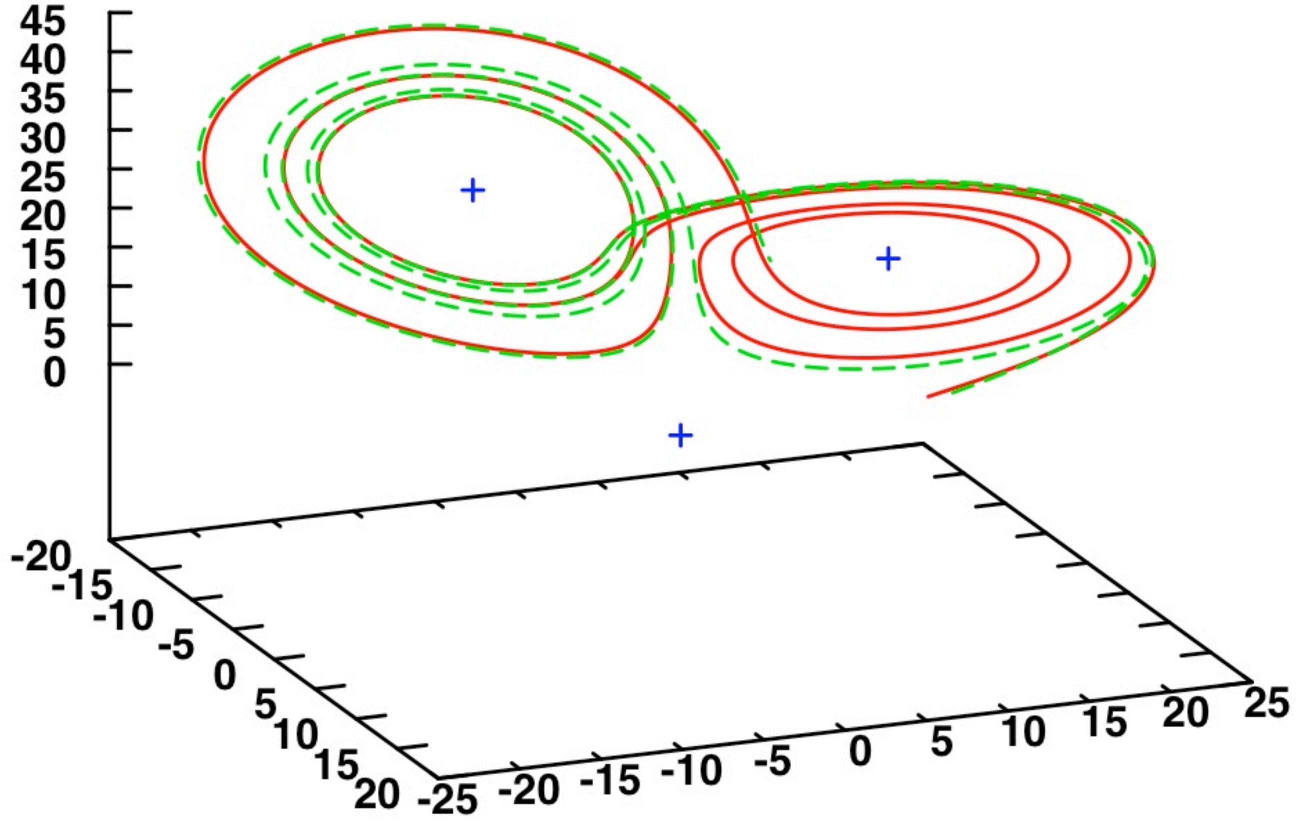


Figure 25: Solutions of the Lorenz system (15.119) with the coefficients shown in (15.121).

15.17 Lorenz attractor Newton parallelization

Figure 26 indicates the number of Newton iterations required to solve the difference approximation to the Lorenz system (15.119) with the coefficients shown in (15.121) using a constant initial guess.

Figure 27 indicates the number of Newton iterations required to solve the difference approximation to the Lorenz system (15.119) with the coefficients shown in (15.121) using a constant initial guess for various time steps.

We see confirmation of the prediction in (15.110).

Figure 28 indicates the decrease in the required number of Newton iterations to solve the difference approximation to the Lorenz system (15.119) with the coefficients shown in (15.121) using an nitial guess based on a (ten times) coarser time step.

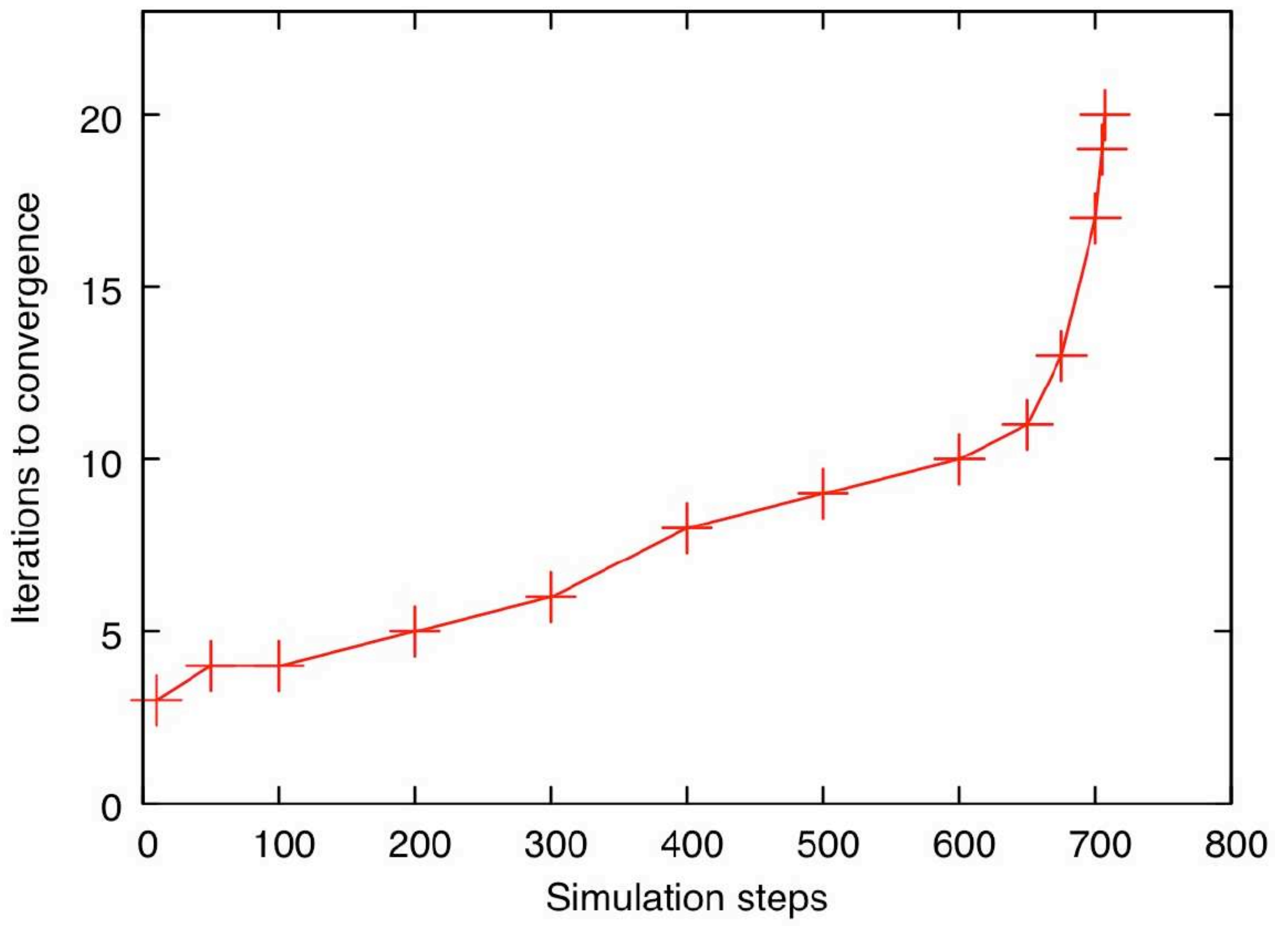


Figure 26: Number of Newton iterations required to solve the difference approximation to the Lorenz system (15.119) with the coefficients shown in (15.121) using a time step of $\Delta t = 0.001$. Constant initial guess.

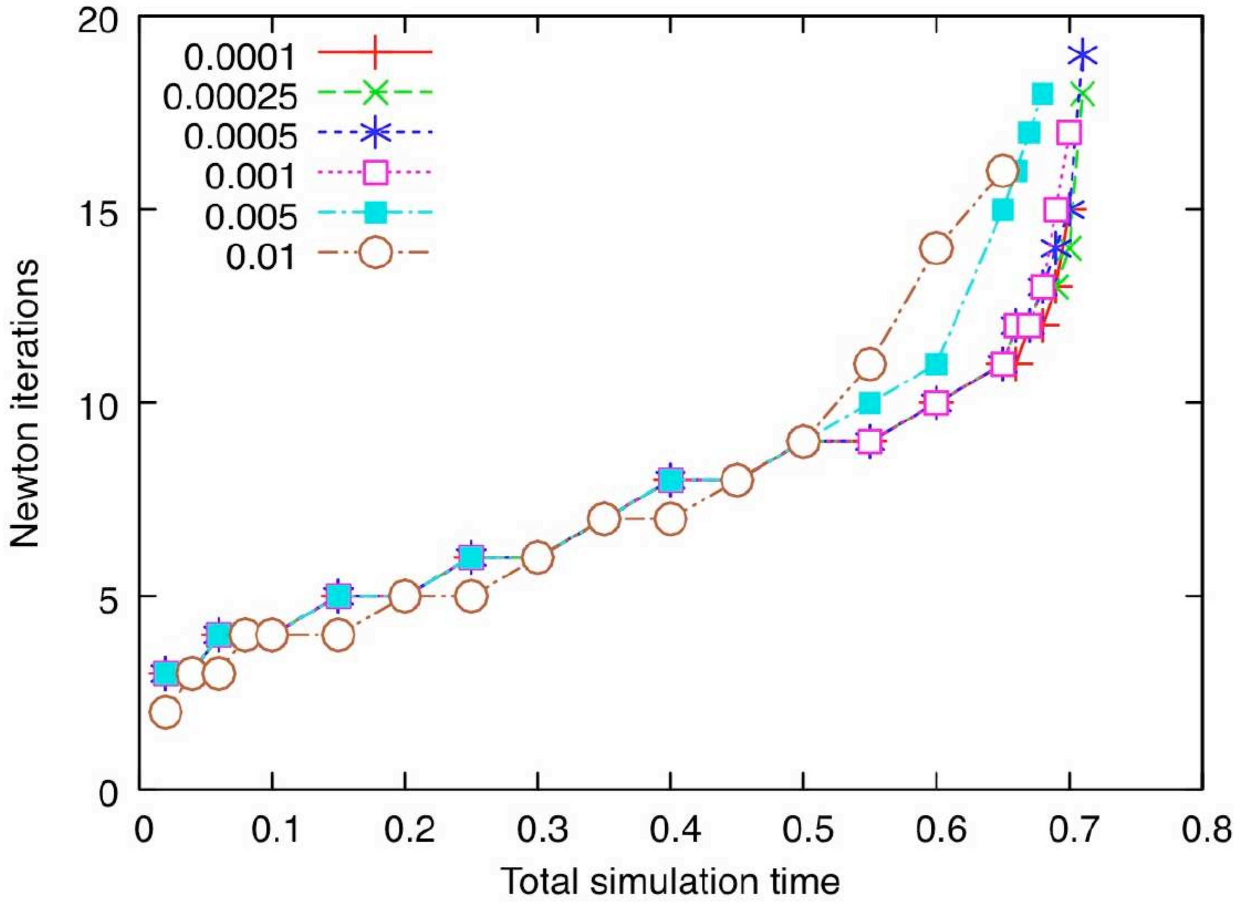


Figure 27: Number of Newton iterations required to solve the difference approximation to the Lorenz system (15.119) with the coefficients shown in (15.121) using various time steps. Horizontal axis is time. The initial guess for the Newton iteration is constant in time.

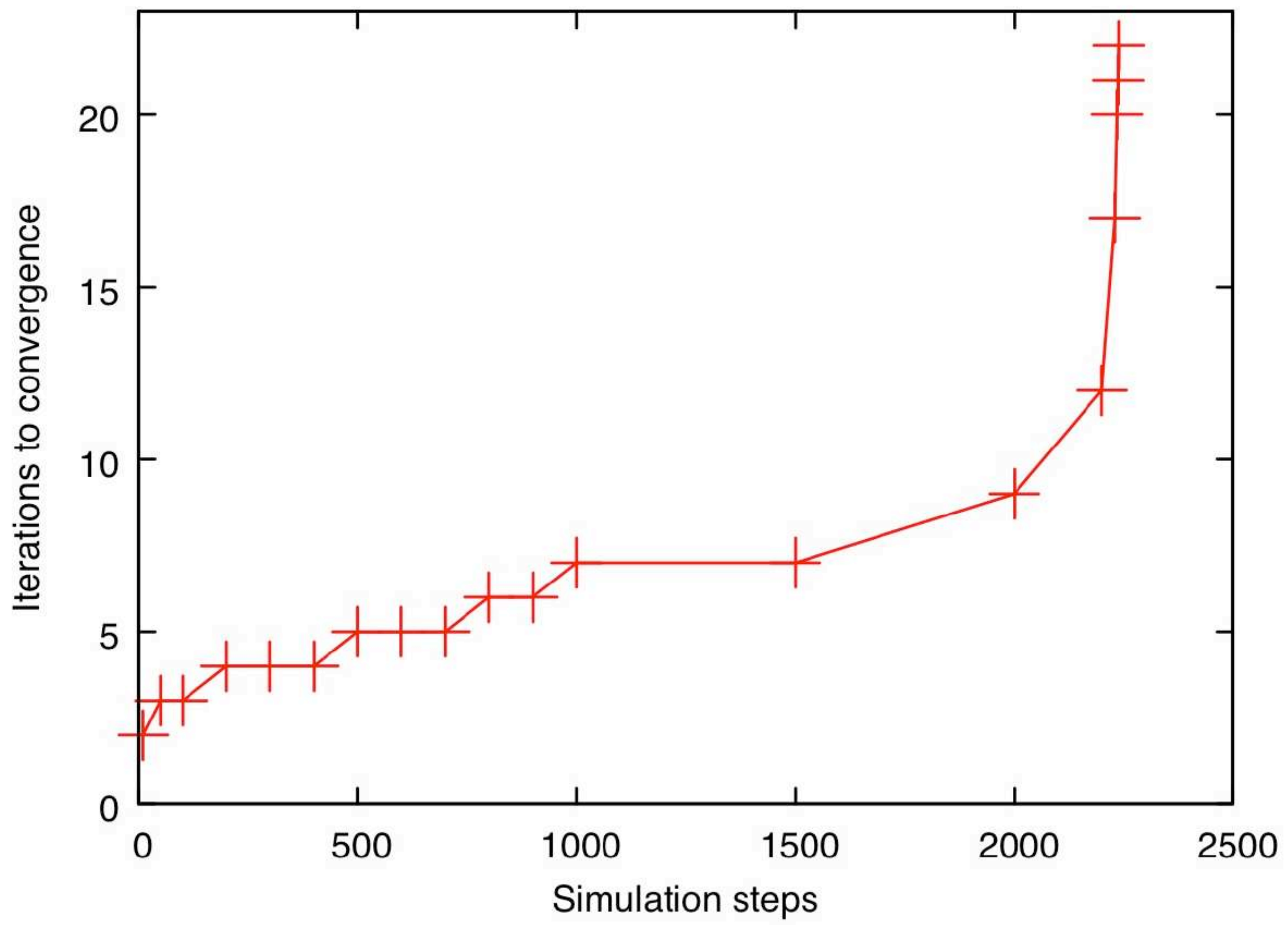


Figure 28: Number of Newton iterations required to solve the difference approximation to the Lorenz system (15.119) with the coefficients shown in (15.121) using a time step of $\Delta t = 0.001$. Initial guess based on a (ten times) coarser time step.

15.18 Other approaches

Many high-order methods have been proposed that exhibit parallelism in the high-order steps [2].

"Parareal" methods provide efficient parallelizations of non-linear, time-dependent problems via domain decomposition in the time domain [1, 13, 14].

- One limitation is the inclusion of a serial section which plays the role of a preconditioner.
- Techniques presented here can be integrated with those of [1, 13, 14] to potentially improve scalability.
- The techniques here may provide a scalable parallel algorithm for this or a similar preconditioner.

15.19 Conclusions (and Perspectives)

We have demonstrated that the Newton parallel method can provide significant speedup for solving ODE's.

The efficiency is limited by the number of required Newton iterations, but the speedup can be arbitrarily large as the time step is decreased.

The main reason that efficiency is limited by the number of required Newton iterations for the problems considered here is that we focused on explicit time-stepping schemes.

Thus the sequential algorithm gets replicated in each Newton iteration.

But for problems requiring implicit time-stepping schemes, a Newton (or similar) iteration might be required even in the sequential case.

In this scenario, the efficiency might be significantly better.

15.20 (Conclusions and) Perspectives

The Newton parallel method benefits greatly from a good initial guess.

What is the best way to use P processors to collectively approximate a solution to an ODE?

- Sounds like original question but now Newton parallel method computes refinement, so initial step need not be convergent.
- One idea: each processor solves slightly different problems and then we use this data to synthesize an approximation.
- Something like the data assimilation problem?

Applying these ideas to ODE's arising from the semi-discretization of PDE's will be a significant challenge.

- Molecular dynamics is typically done with explicit time-stepping methods (e.g., Verlet)
- flow problems (Navier-Stokes) often involve implicity methods (Euler or backward differentiaion) [10].

References

[1] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah. Parallel-in-time molecular-dynamics simulations. *Phys. Rev. E*, 66:057701, 2002.

[2] Kevin Burrage. *Parallel and sequential methods for ordinary differential equations*. Oxford University Press, 1995.

[3] Terry Clark, Reinhard von Hanxleden, J. Andrew McCammon, and L. Ridgway Scott. Parallelization using decomposition for molecular dynamics. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 95–102, Knoxville, Tennessee, May 1994. IEEE Computer Society Press.

[4] Xie Dexuan and L. R. Scott. An analysis of the parallel U-cycle multigrid method. Research Report UC/CS TR-2010-03, Dept. Comp. Sci., Univ. Chicago, 2010.

[5] Y. Duan and P Kollman. Pathways to a protein folding intermediate observed in a 1-microsecond simulation in aqueous solution. *Science*, 282:740–744, 1998.

[6] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM J. Sci. Stat. Comput.*, 9:639–649, 1988.

[7] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comp.*, 9(4), 1988.

[8] A. Ilin and L. R. Scott. Loop splitting for superscalar architectures. *J. Supercomp. Appl. and High Perf. Computing*, 10:336–340, 1996.

[9] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods*. John Wiley and Sons, New York, 1966.

[10] Hector Juarez, L. R. Scott, R. Metcalfe, and B. Bagheri. Direct simulation of freely rotating cylinders in viscous flows by high–order finite element methods. *Computers & Fluids*, 29:547–582, 2000.

[11] A. Karp. Gordon Bell Prize for 1997. *NA Digest*, 97(12):226–319, March 1997.

[12] F. Thomson Leighton. *Introduction to Parallel Algorithms and*

Architectures: Arrays, Trees, and Hypercubes. Morgan Kaufmann, San Mateo, CA, 1992.

[13] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'EDP par un schéma en temps "pararéel". *C. R. Acad. Sci. Paris*, 332(7):661–668, 2001.

[14] Yvon Maday and Gabriel Turinici. A parareal in time procedure for the control of partial differential equation. *C. R. Acad. Sci. Paris*, 335(4):387–392, 2002.

[15] U. Schendel. *Introduction to numerical methods for parallel computers*. Chichester: Ellis Horwood Limited, 1984.

[16] L. R. Scott, T. W. Clark, and B. Bagheri. *Scientific Parlallel Computing*. Princeton University Press, 2005.

[17] L. R. Scott and Dexuan Xie. The parallel u-cycle multigrid method. In *Proceedings of the 8th Copper Mountain Conference on Multigrid Methods, April 6-11, 1997*, 1997.

[18] D. P. Siewiorek, C. G. Bell, and A. Newell. *Computer Structures: Principles*

and Examples. New York: McGraw-Hill, 1982.