DL/SCI/TM95T

technical memorandum

Daresbury Laborato

DL/SCI/TM95T

AN INTRODUCTION TO PARALLEL PROGRAMMING

by

R.J. ALLAN, SERC Daresbury Laboratory.

MAY, 1993

G93/95

Science and Engineering Research Council

DARESBURY LABORATORY

Daresbury, Warrington WA4 4AD

ISSN 0144-5677

An Introduction to Parallel Programming

R.J.Allan

SERC Daresbury Laboratory, Daresbury, Warrington, WA4 4AD, U.K.

These notes form material for a series of lectures to be given at the Curso Supercomputacion Vectorial y Paralela '93, 10-14th May, CIEMAT, Madrid, Spain, 1993.

Contents

Chapter 1 Abstract

This course is intended as an introduction to the general principles of parallel computing. It will enable a programmer already familiar with Fortran 77 and UNIX to adapt or write programs mainly for distributed memory systems. These include distributed workstations on a local area network (e.g. IBM, HP, SGI, SUN with Ethernet, FDDI, SOCC or Ultranet) or closely coupled parallel computers (e.g. Intel, NCube, Meiko, Parsytec, Thinking Machines).

The use of portable message-passing and other tools is illustrated (PARMACS, PVM, Fortnet, Linda) and reference is made to forthcoming international standards such as MPI (Message Passing Interface) and HPF (High Performance Fortran).

Some notes are included on programming shared memory systems using memory mailboxes or POSIX threads (e.g. Cray, Alliant, KSR, Convex and multi processor workstations such as Apollo, SGI, SUN, Stardent). Some final notes are included on designing parallel numerical algorithms and on benchmarking parallel systems.

Chapter 2 Acknowledgements.

I wish to thank the organising committee of the Curso Supercomputacion Vectorial y Paralela '93 for inviting me to present this tutorial material at the workshop to introduce the delegates to the concepts of parallel computing and parallel algorithm design. I especially thank Dr. Araceli Quintero for her help. I also thank Drs. R.J.Blake, D.R.Emerson, J.Carter for their continuing help and support and Mr. J.Garner for his work on a parallel load balancing algorithm.

Chapter 3 Introduction

The increase in speed of electronic computing machines in the period 1950–90 was attributed to improvements in electronic engineering and to the use of parallel computation. Until the early seventies the parallel computation was to a large extent transparent to the user. Since that time however this has ceased to be the case and the user has found it necessary to familiarise himself with some of the details of machine architecture in order to exploit the capabilities of a particular target machine effectively. This is because machines have become highly specialised to achieve high performance for specific types of computation, and to benefit from this there must be a close mapping of the user's code onto the hardware. There is so far no sufficiently general specification language available to allow this to be done simply by a compiler, except in a few specific cases. We thus have a problem that in programming parallel computers (and also vector computers) we must be aware of the machine architecture, and consequently programs, if they are tuned to be highly efficient, are unlikely to be easily portable to other architectures. Some of this lecture material will therefore deal with these topics.

We will here regard "conventional" architectures as being typified by the CRAY X-MP, Y-MP and C-90 range, which have a small number of powerful vector processors sharing a common memory. Programming these vector supercomputers will be dealt with by other lecturers in this course. On the other hand novel architectures, the main subject of the present lectures, usually have a large number of less powerful processors together with distributed memory, a part of which is physically attached to each processor.

Parallel processing has now been around for a long time, both theoretically and practically. Indeed vector computation is a form of parallel processing whereby a single instruction (such as a floating point add) is performed concurrently on many data elements — this mode of operation was given the acronym Single Instruction Multiple Data, SIMD. However most people think of parallel processing as using more than one CPU, enabling Multiple Instructions to be done concurrently with Multiple Data — MIMD.

On the architectural side there are two different types of architecture available for parallel processing distinguished by the interconnect topology of the processing units and memory. Many machines employ shared memory, whereby each processing unit (processor) is connected to the memory by a high-speed bus (bus based). More recently many machines have been built with a distributed memory arrangement. Each processor has a fixed memory size (say between 8 and 32 Mbytes, and there is no virtual memory paging to disk) and the programmer needs to send messages explicitly to communicate data to other processors across electronic links (link based). This is rather like having a set of workstations or PCs connected in a network with an Inter-Process Communication (IPC) interface. Data and files must be shared to collaborate in a large task.

Indeed the distributed memory architectures may be further sub divided into two classes. Firstly there are closely-coupled systems (such as Intel, NCube, Thinking Machines, Meiko or Parsytec) which have many processors in one box connected by special channels or links. Secondly there are UNIX workstation clusters using local area network technology (Ethernet, FDDI, SOCC or Ultranet) to connnect them. A number of manufacturers are now packaging

several of these systems into single cabinets, so the distinction is being lost. Market forces currently mean that workstation clusters are a very successful way of exploiting parallelism since the component workstations are highly developed, readily available, have good software and may be used as stand alone machines during the day and as a single parallel resource at night.

Details of interconnect and processor type will be discussed later. To date the typical computational scientist or engineer has been loath to get involved with the architectural aspects, and it has been primarily the computer scientists, along with some physicists, who have been at the forefront of building, and exploiting, the potential of parallel computers. The last four years have however seen a dramatic increase in the number of practitioners from other fields moving over to parallel machines as commercially available systems become more robust and user friendly. This is even more so now that workstation manufacturers, who can apply more resources to research and development, are becoming involved (e.g. IBM with their SP1 parallel system).

The explosion of the personal computer and workstation market means that high performance processors, multi-megabit memories and fast disk storage are now almost commodity items. The technology of building replicated parallel machines is well suited to robotic manufacture and the cost of Mips (Millions of Instructions Per Second) or Mflops (Millions of FLoating-point Operations Per Second) in these machines is falling rapidly. At the same time the cost of increasing the ultimate performance of a single supercomputer processor is very high requiring exotic integrated circuit and memory capability and crygenic plant to enable high component density with speed of data movement between processor and memory ultimately governed by the speed of light, $3*10^8 ms^{-1}$.

If a processor and memory are separated by 1cm it takes $1/(3*10^{10})$s for data to traverse the link. Assuming two memory access operations must be done for every floating-point operation the peak performance is limited to $1.5*10^{10}$flops, that is 15Tera flops. Such close coupling and fast enough processor cannot currently be contemplated, yet several manufacturers of parallel computers are predicting a Teraflop computer by 1995 and speed far in excess of this by the start of the next century.

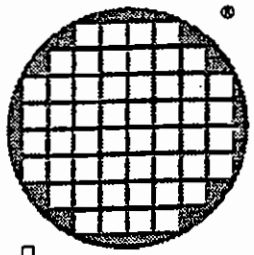
Section 1 Components for Parallel Computers

Harnessing the power of many simple and cheap processors is an area which can then draw on mass-produced components. The same ultimate performance (or more) can be available as in a supercomputer, but at a fraction of the cost. There is however the need to communicate data between the processors which adds design complexity, and programming complexity if we are to use the system effectively (i.e. map applications well onto the parallel hardware).

Some parallel computer manufacturers use commonly available components to reduce to cost of machine design. Others maintain that the standard components are unable to satisfy the enhanced requirements of a sophisticated memory and communication interface integrated alongside the processor. A few common machine types and their constituent components are listed here:

machine	processor	comms. device
Intel iPSC/2	80386 + 80387	Intel DMA device
Intel iPSC/860	Intel i860	Intel DMA device
Intel Paragon	Intel i860 XP	Intel DMA device + PMRC
Meiko i860	Intel i860	T800 + crossbar switch
Meiko CS2	SPARC + 2*Fujitsu	multi level switch
Thinking Machines CM5	SPARC + 4*Weitek (?)	Data Network
Parsytec Multicluster	T805	crossbar switches
Parsytec GC	T9000	T9000 VCP + crossbar
IBM SP1	RS/6000	SOCC and routers
CRAY MPP	DEC alpha	DMA
NCube	NCube processor	routing device
Kendall Square	KSR processor set	KSR cache devices

The need to move data rapidly and flexibly was for instance solved in an elegant way by integrating a communication device into the processor chip in the Inmos Transputer, Texas Instruments C40 and NCube processor. Data movement however still incurs an overhead and many other projects have failed simply because it was too slow in relation to the available processor speed. A block diagram of the new Inmos T9000 transputer is shown below. This represents the state of the art in such devices, having a highly modular parallel internal structure. It is also the highest current packing density with over 3.3 million transistors in just under 2 square centimetres.



inmos®

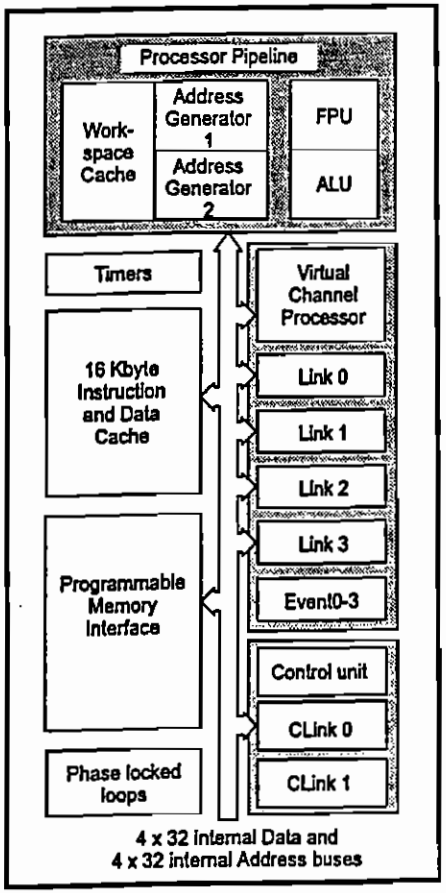
IMS T9000 transputer

Preliminary Data

The information in this datasheet is subject to change

FEATURES

Pipelined superscalar micro-architecture
Workspace cache
Programmable memory interface
4 Gbyte physical address space
16 Kbyte instruction and data cache
200 MIPS peak
>70 MIPS sustained
25 MFLOPs peak
>15 MFLOPs sustained
Sub-microsecond interrupt response
Per process error handling
Enhanced support for pre-emptive schedulers
Memory protection and address translation
64 K virtual communication channels
Support for message routing
80 Mbytes/s total bi-directional link bandwidth
Separate control system
Single 5 MHz clock input
40, 50 MHz speed options
Single 5 V ± 5% power supply



SGS-THOMSON MICROELECTRONICS

INMOS is a member of the SGS-THOMSON Microelectronics Group

August 1992
42 1460 03

Section 2 What are the goals of parallel computing ?

The answer to this question depends upon the point of view of the user. For a fixed size problem, the user would generally like to see the wall clock (elapsed) time reduced, e.g. if the problem took 1 hour on a single processor he would like to get the results back in 15 minutes on four processors. However there are many people who would prefer to increase the problem size and complexity of their simulations e.g. the inclusion of a more sophisticated turbulence model in computational fluid dynamics (CFD), the simulation of a complete aircraft, car or industrial process to improve design time and safety, or an all-forces molecular dynamics (MD) or computational quantum chemistry (QC) simulation of a large systems, finer grid mesh for accurate numerical weather prediction and climate modelling. Many people are looking for cost-effective computing. Parallel computing on novel architecture systems or networks of cheap workstations offers the possibility of more Mflops per unit cost than on current supercomputers such as the Cray. This is because many replicated off-the-shelf components are cheaper and the novel systems are usually air cooled, whereas supercomputers are hand built and refrigerated with expensive liquid plant.

Such cheap replicated systems have however not yet been perfected. I/O still remains a bottleneck on parallel machines and gives the conventional supercomputers a distinct advantage for certain applications. Parallel computers only have a limited market, for systems larger than attainable by other means at reasonable cost. The companies that produce them are small. Now that workstation vendors are moving into this market place however some of the remaining problems will be tackled.

Section 3 Some Facts and Figures

The table below gives some information about currently available parallel supercomputers. It quotes the maximum parameters of the system, which can be obtained from the manufacturers published information.

Make	Max procs	Total memory (a)	total max speed (b)	x-sectional bandwidth (c)
Intel Paragon	4096	128	300	12.8
Meiko CS2	256	36	51.2	13.1
Thinking Machines CM5	16384	512	2097	81.92
Parsytec GC/T9000	16384	512	400	102.4
CRAY MPP	2048		300	
CRAY C90	16	2	16	n/a
KSR KSR1	992	32	40	n/a

(a) in Gbytes

(b) in GFlops

(c) in Gbytes per second. This is the rate at which an arbitrarily chosen half of the total processors can send data over communication channels to the other half.

Chapter 4 Parallel Architecture Classifications

Section 1 Flynn's Taxonomy

Flynn (1972) introduced a classification of computer systems that is now widely used by the computing community. He divided machines into four categories based on how the machine relates its instructions to the data being processed. The categories are:

SISD	Single Instruction stream, Single Data stream
SIMD	Single Instruction stream, Multiple Data stream
MISD	Multiple Instruction stream, Single Data stream
MIMD	Multiple Instruction stream, Multiple Data stream

Here the term stream refers to the sequence of data or instructions as seen by the machine during the execution of a program. An instruction stream is a sequence of instructions as executed by the machine and a data stream is a sequence of data including input, partial or temporary results called for by the instruction stream. The SISD computer represents most serial computers available today. The original von Neumann concept mixes instructions and data (possibly referred to via indirect addressing) in one stream using a single program pointer and possible alternative registers used for fetch and put operations. The SIMD operation corresponds to a vector computer or array processor such as the AMT DAP (Distributed Array Processor). The MISD operation corresponds to a pipeline computer (at least in these notes !), this has met considerable success in modern processor chip design in which high floating point capability is achieved with three or four linked add and multiply units in a pipeline architecture. The MIMD classification includes all forms of multiprocessor configurations, from linked mainframe computers to large arrays of microprocessors. In practice many current machines are hybrids. For example the Cray X-MP has up to four processors (MIMD) with each processor using pipelining (MISD), and can no longer be classed as a von Neumann machine.

A subdivision of Flynn's SIMD and SISD classes was produced by Shore (1973).

We will also use a commonly met terminology to distinguish between shared memory and distributed memory MIMD computers. We will call the former multiprocessor computers and call this class SM-MIMD, and the latter multicomputers and call the class DM-MIMD.

Sometimes the term SPMD (Single Program Multipe Data) will be met in discussions of programming models. This is a corruption of Flynn's SIMD term. It adds nothing to our classification and is in fact misleading because even if we write a single program it will inevitably contain branch points which depend on a processor's state, so multiple instructions will be executed concurrently, and we are back to MIMD mode. We will however see that in programming these systems it is usual to have a single source code that is compiled and executed on all processors simultaneously. For reasons of scalability it would be impossible to do otherwise.

Chapter 5 Parallel Architecture — Topologies

One of the most important aspects of parallel computing is how the individual processors communicate with each other. This is particularly true for processors with only local memory. There are many different ways of connecting processors together. The main difference between the approaches, as far as the user is concerned is whether the memory is local (distributed, DM-MIMD) or shared (SM-MIMD).

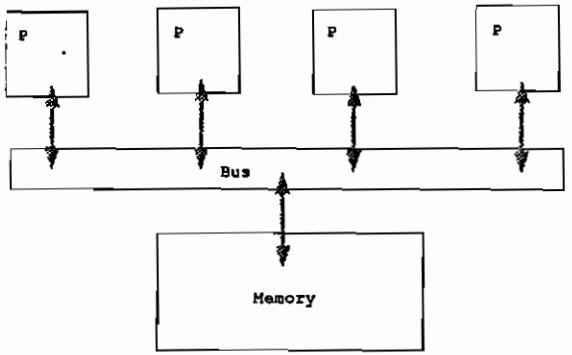
Section 1 Banked Shared memory

Many architectures use a shared-memory arrangement whereby each processor can access data from a common memory store. Any communication between the processors must be via the global memory. One problem that can arise with this system is that more than one cpu may wish to access the same memory area, or even that one cpu can access it multiple times from its many internal functional units. This is known as memory contention. A system of locks and semaphores controlled by software must be used to control coherency of access to the memory. A time delay is then incurred whilst one functional unit waits for the other until the memory is free, and it reduces the efficiency of execution. To reduce the probability of this occuring the the memory is split into "banks" which complicates programming. If two funtional units on one processor try to access the same bank a "bank conflict" arises. This must be handled by the compiler and operating system in dictating the data layout in memory and the order of access. A "simultaneous conflict" occurs when the same bank is accessed from different cpus. The operating system alternates the priority of the cpus every few clock periods. A "line conflict" occurs when two or more of the four lines into a cpu attempt to access the same bank. We will not discuss these problems any further here as they may be covered in other lectures.

Examples: Cray X-MP, Y-MP, C90, NEC SX3

Section 2 Bus-driven shared memory

Another shared memory approach is to connect processors with a high speed bus, as shown below. The processors are synchronised by reading and writing to the global memory. A problem that arises with this approach is "bus contention", which can become severe with large numbers of processors. This means that such architectures cannot be scaled over about 20 processors (The Alliant fx/2828 had 28 of them, of which 24 could be programmed and four were for i/o). Using a local cache memory can help alleviate this problem, but adds complexity in that cache coherency must be maintained across the processors either by hardware or software.

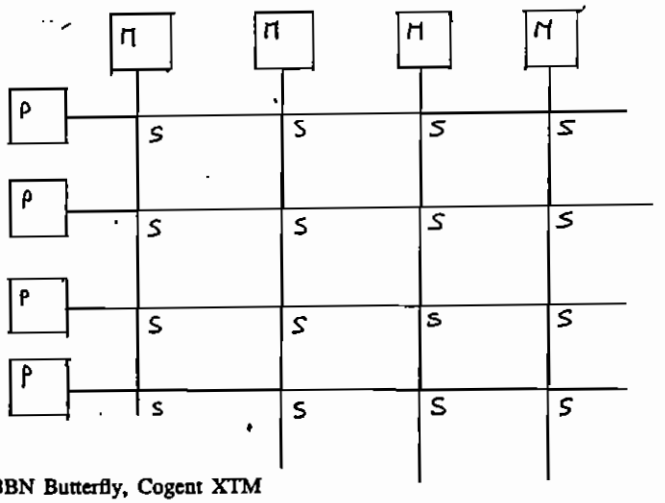


Examples: Alliant fx/8, fx/2800, Convex C2 and C3, Encore, Sequent, SUN, Silicon Graphics, Apollo, Stardent

Certain activities, such as updating variable, are not atomic and must be protected by explicitly calling locks. If this is not done two processors could change the value simultaneously and cause a loss of cache coherency. This is true in all shared-memory systems.

Section 3 Switch driven shared memory.

One method of obtaining a virtual shared-memory system with distributed memory blocks is to use a "crossbar" or "butterfly" switch. Each processor then has access to all the memory units with fewer connection links. However the number of switches required to connect p processors to m memory modules is p*m. Memory contention occurs as usual if two or more processors try to access the same memory. Switch contention occurs if two or more processors require the same switch. This is the same as memory contention in a full network, but may occur at other times in a lower order permutation network. There is an additional overhead in controlling the switches.



Example: BBN Butterfly, Cogent XTM

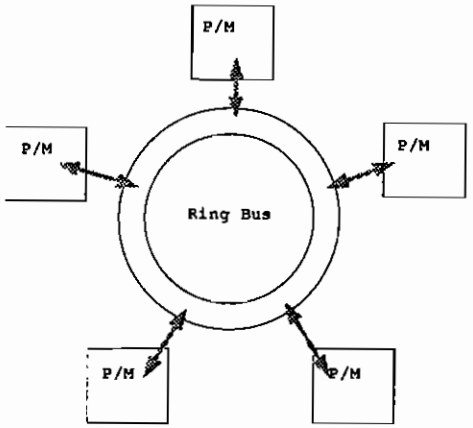
Section 4 Distributed memory Ring Network

A ring network, shown below, utilises a high-speed bus or network approach. Each message placed on the ring by a processor would contain a destination address and a source address. Hierarchies of rings can be built to produce a scalable design wich still has good local communication latency and overall bandwidth.

In the Kendall Square Research KSR1 cache coherency and messages are handled in special hardware. There is only cache memory in the system, each processor has 32 Mbyte locally attached. Pages of data migrate around the system following memory references in a data-flow fashion and the system is either programmed like a shared memory system using compiler directives, using POSIX pthreads and memory mailboxes (see below). Again locks, barriers and semaphores are necessary to control cache coherency.

A number of parallel applications in science and engineering are now being developed worldwide and are being run on UNIX workstations coupled by local area networks (LAN), either copper Ethernet or CDDI or optical fibre such as FDDI or the IBM Serial to Optical Channel Converter (SOCC), and Ultranet. This is a potentially attractive prospect since workstation cpus are often only partly used interactively, and may sometimes be entirely unused during night-time periods, especially in educational environments. A parallel background job can utilise wasted cpu cycles and return high effective performance, allowing useful research to be undertaken with existing equipment. Manufacturers such as IBM and Hewlett-Packard are now even offering clustered workstation solutions to high-performance computing requirements. A further incentive is to make use of different capabilities of networked UNIX hosts, perhaps combining a compute server, a disk server and a high performance graphics engine (e.g. Allan et al. 1991). A

number of *ad hoc* methods involving UNIX sockets have been used in the past to achieve this. Whilst Ultranet is a star network with an intelligent control hub (see below) the other LANs are topologically variations of a ring.



Example: Kendall Square Research KSR1, Ethernet and FDDI or IBM SOCC for connecting workstation clusters.

Section 5 Distributed memory with completely connected links

As previously mentioned, local memory systems need to communicate with each other. Therefore any inter-processor communication is done by message-passing, i.e. data or other information is explicitly transferred between processors. A completely connected system of p processors would require p-1 connections from each processor to all others. There are p*(p-1) connections in total. This is clearly impractical for a large number of processors. Thus some connections are removed and a lower order system such as a hypercubes or 2 or 3 dimensional grids are used (see below). Processors must explicitly communicate data which implies a form of synchronisation called "loose synchronicity" (terminology due to Fox). This involves one processor waiting, plus the transmission time which is slow compared to direct memory access because some form of network control protocol must be used. In addition contention occurs if two processors want to communication with a third — one of them must wait (it is "blocked").

There are ways to approximate complete connectivity by using multi-stage switched networks. Interconnect systems exist which approximate full switches using a number of levels of lower order interconnect. Examples of these are the omega, binary n-cube, banyan and R shuffle exchange networks and the Benes network which reduces an NxN crossbar to two N/2 x N/2 crossbars. In programming these machines permutation algebra is used to reduce the problem to a number of sub problems, each requiring a different setup (permutation) of the switches. An example of this system is the Meiko CS-2 which uses a Clos network:

Section 6 Star Network

A star network has a number of satellite nodes which are connected to a central hub. The hub arbitrates and routes packets of data between the satellites. The hub may in some cases be simply a crossbar switch, and communication can be very rapid, up to tens of Mbytes per second.

Example: Ultranet for connecting workstation clusters

Section 7 N-dimensional mesh or toroid topology

A popular connection scheme today is to employ a mesh arrangement. The simplest is a linear array. This is like the ring above but uses links rather than a full bus architecture. In this arrangement each processor is connected only to its nearest neighbours. The two end processors can be connected to form a closed ring but some facility must then be provided to talk to the outside world. Whilst this arrangement has the advantage of simplicity it has the disadvantage that data may need to be passed through up to (p-1)/2 processors. In modern designs the links are controlled by separate through-routing chips or special link engines with Direct Memory Access (DMA) using dual-ported memory so they can work in parallel and be integrated closely with the cpu, so whilst this design no longer suffers from increased transit times for increasing p it still means that only one message can be transmitted at a time.

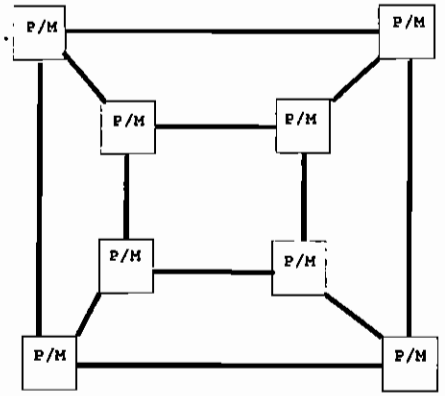
To improve this situation high dimensions of grids and tori are proposed. 2D and 3D systems have been built. The number of links on each processor is 2^d where d is the dimension of the mesh. It is independent of the number of processors so the system is physically scalable.

Example: Intel Paragon (2D), Parsytec Multicluster and GC-T805 (2D switchable), Parsytec GC-T9000 (3D)

Section 8 Hypercubes

One of the frequently used connection topogies in the past was the binary hypercube. This combines relative simplicity with relatively short paths between any two nodes. The number of processors in the system is $p = 2^d$ which d is the "dimension" of the hypercube. The number of links on a given processor is also $l_p = d$ so the total number of links in the system scales as $l = p*l_p/2$. This incurs a disadvantage that as p grows the number of links must be increased, and there is a practical limit to the size of cube which can be built.

To construct a hypercube of size d+1, take one of size d, replicate it and connect all equivalent processors together. A 3D hypercube on two-dimensional paper looks like this:



This just preserves the topology, but distorts the lengths of the links. A 4D hypercube just looks like a cube inside a cube in three dimensions.

Examples: Intel hypercube, NCube hypercube

It is clear that manufacturers who considered hypercubes in the past are now turning to 2D or 3D meshes, rings or hierarchies of these for reasons of reduced engineering complexity and a good compromise on average and maximum path lengths. Such systems are not however formally scalable in the mathematical sense since the average pathe length between pairs of processors increases faster than log(p).

Section 9 SIMD processor arrays

A final example is an SIMD architecture in which many processors, say a 64*64 array, have their own data memory, but execute instructions in lock-step mode with instructions broadcast from a central controller. Hardware exists to do this broadcasting rapidly, but it may still be a bottleneck in very large arrays of processors. Processors have their own local memory, and can only communicate with neighbouring processors, usually distributed in a 2D array, by left-right or up-down shift operations using special registers.

Example: AMT (Cambridge) DAP, Thinking Machines Corporation CM2, MasPar

Consider a vector dot product:

$x = \Sigma_{i=1,N}\ a(i)*b(i)$

this is executed sequentially as

```
x=0.0
do i=1,N
 x = x + a(i)*b(i)
end do
```

On a SIMD computer we assume that related elements of a and b are stored on the same processor. The multiplication step is then done concurrently to yield a vector of partial results

x = a*b

then shift and sum in vertical direction

```
sum = x
do i=1,nprocv-1
  shift_up(1, x)
  sum = sum + x
end do
```

then do same for horizontal shift and sum

The result is available on all the processors. There is one add step followed by nprocv+nproch–2 shift and add steps rather than N multiplies and N-1 adds.

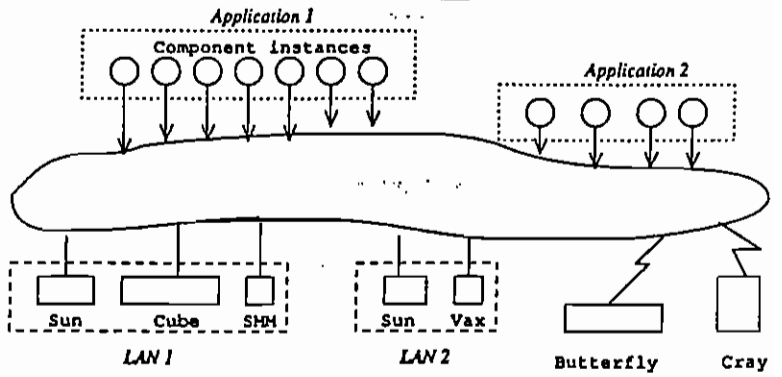
A special version of Fortran is usually used to express this in simple vector notation (e.g. Fortran+ on the DAP, Fortran–90 on the MasPar). Extensions leading to the High Performance Fortran (HPF) definition are discussed in a later chapter.

Examples: AMT (Cambridge) DAP, MasPar, Thinking Machines CM2

Section 10 Heterogeneous Systems

One example of a distributed memory system is of a computer built with specialised nodes to facilitate special jobs; for instance i/o or graphics. Enough nodes can be built into a complete system to balance the remaining numerical work. Communication is by sending messages, so all services can be provided this way on a client-server request basis. The idea is similar to doing graphics on an X-terminal with the computation being carried out elsewhere, but data is transferred in packets across the network. Indeed a sensible parallel computer can be constructed from a group of UNIX workstations with an Ethernet network. Another example is a shared-memory UNIX front end to a distributed memory compute server: the attached processor (e.g. CRAY Y-MP attached to a CRAY MPP). This opens many possibilities.

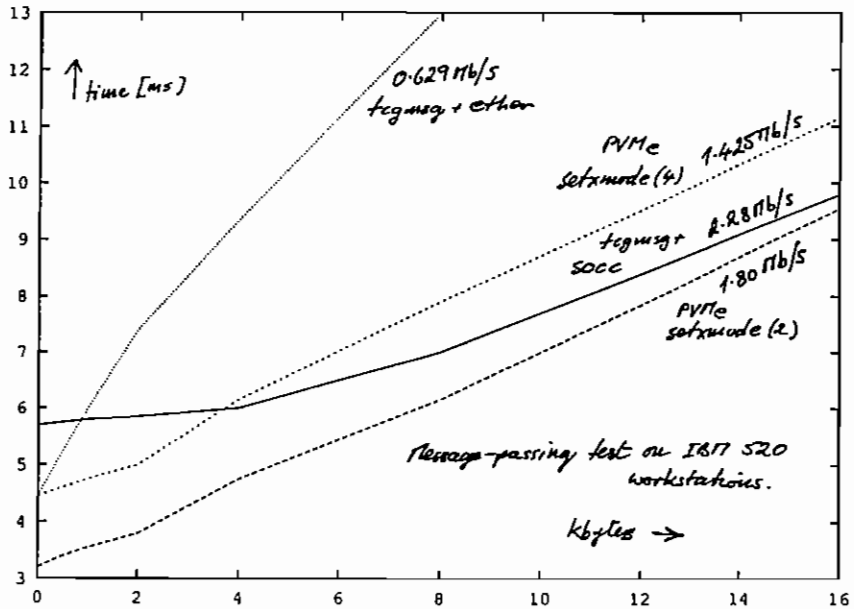
Example: network of workstations, control of a parallel processor from a graphical front end



Section 11 Link and Network Bandwidth

Some examples are given of currently available message passing speed which can be supported between neighbouring nodes of a parallel machine. Numbers (in Mbytes per second) are meant as a guide only, and may not be entirely accurate.

T9000 link	T800 link	Intel iPSC/860	Ultranet	FDDI	Ethernet
12.5	2.5	2.8	10	4	1



Chapter 6 Algorithm Complexity.

Section 1 General Principles

As previously mentioned, many people are interested in reducing the wall clock time required to obtain their results. This introduces the important concept of speed up. If the computations were carried out in p equal parallel parts then the wall clock time would be ideally reduced by a factor 1/p. If the wall clock time is denoted t_p for a given task executing on p parallel processors, then the speed up is defined as:

$S_p = t_1 / t_p$

The speed up is a measure of how the algorithm compares with itself on 1 and on p processors. For the fixed size example previously quoted the task took 1 hour on one processor (t_1 = 60 minutes) and 15 minutes on four processors (t_4 = 15). Hence in an ideal situation the speed up would be S_4 = 60/ 15 = 4 and is equal to the number of processors. In practice other factors influence the speed up ocurring. These factors are the extra time to load the program on four processors as against one, the extra time taken to communicate data between the processors which is slower than acessing it directly from memory, and increased time for i/o if processors clash on disk access (contention). We therefore introduce the concept of parallel efficiency:

$E_p = S_p / p$

For the case where the algorithm has a perfect speed up the efficiency is 1. In general the efficiency lies within the range $0 < E_p < 1$. (In some cases E_p may be >1, reasons for this are given below, but it is exceptional and will not be further considered.)

Section 2 Overheads

There are several reasons why perfect speed up is not attained.

1. the algorithm may not be perfectly parallel, i.e. certain parts may have to run serially on one processor and then have the result broadcast to the others, or it may be run on all of them duplicating the calculation. The result needs to be got before anything else can be done. Typical of this is the calculation of a global sum or summing points and weights of an integral quadrature.
2. Load balancing of the processors may not be perfect. When a code is run in parallel each processor is assigned an amount of work to do based on some estimated cost. If one processor is required to do more work than another in a particular part of the program it will take longer to complete its allocated job and the other processor may have to wait for the result before execution continues. Waiting increases the elapsed time without doing useful work
3. There is an extra cost of communication between processors which is not needed in the serial algorithm. In general when a program is run in parallel, each processor will have to send or receive data from other processors. Each time a message is sent a large overhead in timing cost is incurred because, in current machines, it is much slower than accessing main memory.

4. The code may need to be synchronised after certain operations. Synchronisation is an operation whereby two or more processors exchange information to coordinate their activity. This can lead to one of the processors having to wait for the other to complete its job. This is really the same as items 1 and 2 above.
5. If the nodes have vector capability a large number of them may perform badly because the vectors of data have been distributed over them and each processor is left with only a short vector to work on. Vector pipelining incurrs a startup cost and is less efficient on short vectors.
6. In many cases the serial algorithm will not be easily amenable to parallel decomposition, for the reasons itemised above. If we need to choose a different algorithm this is likely to be one which runs less efficiently on a serial computer. This on one hand complicates our simple-minded calculation of the efficiency because we are no longer comparing like with like, and on the other adds new overheads because the parts of the code which are distributed between the processors are themselves less efficient than the original serial code.

Section 3 Advantages ?

1. Some advantage and therefore *increased* efficiency can be gained from improved cache memory utilisation due to lower memory requirements per node when the data is distributed. This can result in the parallel efficiency of an application using a large amount of memory actually being greater than 1 on a small number of nodes. It will invariably drop below 1 as the number is increased.
2. This is also noticeable in practice on shared-memory systems which have processor caches. It is more efficient to program them using message-passing rather than using the concept of globally shared data. Because we are physically dividing the data, the full cache operation, as controlled by the compiler, is possible with no loss of coherency.

Section 4 A simple complexity model and Amdahl's Law

If we make the assumption that all of the operations are carried out at the same computational speed and that a fraction f can potentially be carried out in parallel on p processors whilst the rest of the work 1-f is carried out on 1 processor (or has to be repeated on p of them), then the elapsed time for the computation on one processor will be:

$t_1 = t_s + t_{||}$ and $f = t_{||} / (t_s + t_{||})$

now on p processors

$t_p = t_{||} + g(p)*t_{||}$ where $g(p) = P(p) + 1/(Ep)$

P(p) is the parallel overhead and E is the processor utilisation.

Hence the speed up is given by:

$Sp = t_1 / t_p = (t_s + t_{||}) / (t_s + t_{||} - t_{||}(1 - g(p)) = 1/(1 - f(1 - g(p)))$

It can be seen that the speed up is reduced by a factor approximately f + p(1 - f). This relationship for speed up is sometimes referred to a Ware's Law. From this expression for speed up the efficiency is given by:

$Ep = S_p / p = 1/(f + p(1 - f))$

Amdahl's Law can be applied to parallel processing. In most practical situations, as with vector computing, not all operations can be executed in parallel, and as the number of processors gets very large the serial part dominates. The speed up is bounded by:

$S_{p->inf} < 1/(1-f)$

Hence the speed up is limited by the proportion of work carried out in serial (including the overheads). It is therefore necessary to choose an algorithm for parallelisation in which f can be got very close to 1 to optimise the performance. This will in many cases certainly not be the original serial algorithm, which was chosen for entirely different reasons. Even with f close to 1 there are significant reductions in speed up for increasing values of p. The table below illustrates the performance deterioration for various values of p and f.

f	p=1	p=2	p=3	p=4	p=8	p=16	p=32	p=64	p=inf
1.0	1.0	2.0	3.0	4.0	8.0	16.0	32.0	64.0	inf
0.99	1.0	1.98	2.94	3.88	7.48	13.91	24.43	39.26	100.0
0.98	1.0	1.96	2.88	3.77	7.02	12.31	19.75	28.32	50.0
0.96	1.0	1.92	2.78	3.57	6.25	10.0	14.29	18.18	25.0
0.94	1.0	1.89	2.68	3.39	5.63	8.42	11.19	13.39	16.67
0.92	1.0	1.85	2.59	3.23	5.13	7.27	9.19	10.60	12.5
0.90	1.0	1.82	2.50	3.08	4.71	6.40	7.80	8.77	10.0
0.75	1.0	1.60	2.00	2.28	2.91	3.37	3.66	3.82	4.0
0.50	1.0	1.33	1.50	1.60	1.78	1.88	1.94	1.97	2.0
0.25	1.0	1.14	1.20	1.23	1.28	1.31	1.32	1.33	1.33
0.10	1.0	1.05	1.07	1.08	1.09	1.10	1.11	1.11	1.11
0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

No parallel processor actually has an infinite number of nodes, so it is the relative proportion of parallel to serial work which is important in a given case. It is still important to do as much parallel work as possible. Take a simple dot product for instance

$a = \Sigma_{i=1,N}\ x(i) * y(i)$.

We want to have the answer a on all the processors. This can be done as follows. Suppose we have a ring of N processors

1. do a = x(i) * y(i), one element on each processor
2. send local value of a to next processors
3. get value from previous processor, call it tmp
4. accumulate a = a + tmp
5. repeat steps 2-4 N-1 times.

This has reduced the problem from N multiplies and N-1 adds on a single processor to 1 multiply and N-1 adds plus N-1 sends and receives of data. This is ridiculous because:

1. it is as fast to multiply as to add on modern processors so f is limited to f = 0.5 only
2. there is a large additional overhead because it is very slow to send and receive data, currently equivalent to several floating point operations on modern processors.

All in all it would take longer to do this problem in parallel than on a single processor.

This discussion introduces us to the concept of "granularity". For a given set of p processors we want to make sure of several things:

1. fraction of parallel work is high
2. total work is much greater than communication overhead

On current machines each bit of parallel computation had better consist of hundreds of floating point operations. This is what we call coarse grained as opposed to fine grained which just has a few operations and then some data transfer as in the dot product case above.

On the Intel iPSC/860 hypercube for instance the asymptotic data transmission rate is 2.8 Mbyte/s. It takes 357 ms to send 1Mbyte of data. The i860 processors are each capable of say 60Mflops peak, so in this time 21.4 million floating point operations can be done. If this were the case (we do 7.7 times as much computation as communication) the implementation would still only be 50% efficient because half the time is spent in communication rather than all in computation in a serial code. This machine is badly balanced in this respect. If we carry out a similar calculation for a T805–30 transputer,with 4.4 Mflops peak performance and communication at 3Mbyte/s we could have to do only 1.5 times as much computation as communication to achieve 50% efficiency.

For a given size of problem it then depends on the number of processors because the amount of computation is divided between them, but the communication will roughly stay the same or increase as p increases, e.g. if we had only two for the dot product we could execute the algorithm as follows:

comments	proc1	proc2
Parallel	a1 = sum i=1,N/2 x(i)*y(i)	a2 = sum i=N/2+1,N x(i)*y(i)
overhead	send a1 to 2	receive a1 from 1
sequential		a = a2 + a1
overhead	receive a	send a to 1

If N is large this would not be too bad. It is of course possible to do more work than just adds and multiples in real applications, but this kind of worst-case problem could dominate the algorithm if we are not careful.

Conclusion: the bigger the problem, the better it will perform in parallel. But there are constraints about memory, so for a given problem there is a practical limitation to the number of processors that can be applied to it in a given architecture if sensible efficiencies are to be obtained.

Section 5 Gustafson's Law

Gustafson (1988) demonstrated that in fact the assumptions underlying Amdahl's 1967 argument are inappropriate for the current approach to massive parallelism. He did tests on an NCube–10 parallel machine with 1024 processors at Sandia Laboratories, USA. Each processor was capable of 0.17 Mflops and has 0.5 Mbyte local memory and 0.5 Mbyte/s interconnect. Amdahl's Law, based on the approximate equations above, contains the implicit assumption that the fraction f is independent of p. It says that asymptotically:

$S_p = 1/(1 - f)$

As demonstrated by Gustafson et al. (1988) this is virtually never true. They ran three applications and achieved speedups as follows: 1021 for beam stress analysis using conjugate gradients, 1020 for baffled surface wave simulation using explicit finite differences, and 1016 for unstable fluid flow using flux-corrected transport. The serial code has a parallel fraction f between 0.6 and 0.2 for these applications. The good results were achieved by scaling up the size of the problem roughly in proportion with the number of processors used. This gives an additional factor h(p) which explains the scale-up of the parallelism in the problem with processor size:

$t_s = t_s + h(p)*t_{||}$

$t_p = t_s + h(p)*g(p)*t_{||}$ where again $g(p) = P(p) + 1/Ep$

$S_p = (t_s + h(p)*t_{||}) / (t_s + h(p)*t_{||} - (1 - g(p))*h(p)*t_{||})$

so by analogy $f = h(p)*t_{||}/(t_s + h(p)*t_{||})$

clearly as h(p) is increased this pushes the parallel code fraction, and hence the efficiency, up.

Section 6 Valiant's Thesis

In an important paper (1990) Valiant introduced his Bulk- Synchronous Parallel (BSP) model of programming. Several conditions for optimal execution of a parallel computation were discussed and far reaching conclusions were drawn. Some of the conclusions are more relevant for the compiler or parallel system designer, but the programmer should also bear them in mind.

The most important concept is of "parallel slackness". This means programs are written for v virtual parallel processors to run on p physical processors where v is rather larger than p. Valiant suggests ideally at least $v = p * log(p)$. This is a non-trivial conclusion. The slack can be exploited by the compiler to schedule and pipeline computation and communication efficiently. Communication can be overlapped with computation and the system will scale up as p is increased with only constant factors missing from the formulae.

High level languages that could be compiled in this mode would allow a virtual shared memory address space, or sharing of global data structures. The program would have to be so expressed that v parallel instruction streams could be compiled with it. A PRAM (Parallel Random Access Memory) language would be ideal, but other styles are also suitable.

We will consider in later chapters some suitable programming styles, including High Performance Fortran (HPF) and PARLANCE (Parallel Library and Network Computing Environment), which are shared data models. To the programmer Valiant's thesis implies that if, for instance, he wishes to carry out a vector operation on n vector elements, it can run optimally on up

to p processors as given by $n = c * p * log(p)$. Some values (with a constant proportionality factor c) are:

p	2	4	8	32	64	512	16384
n	2	8	24	160	384	4608	229380

This implies that parallelisation of truly sequential code is, at the least, difficult.

For the hardware manufacturer there is also a message. The model relies on overlapping computation and communication. Within each parallel code segment communication must be scheduled by the compiler or run time stsrem. Using plog(p) processes on p processors enables such an overlap to be made and for the system to scale up in p to a constant factor. This means that processes are clustered together to increase effective granularity. Locality is exploited and the computation is viewed at a higher level of granularity. This is a direct deduction of Amdahl's and Gustafson's laws. However the constant factors in the system depend on the communication bandwidth which must be high enough. Valiant defines a parameter:

g = computation speed / communication speed

We can estimate some values of this from the above tables.

As we have already seen, having architectures with high bandwidth connectivity means extra manufacturing expense. Nevertheless Valiant's theses states that "if these costs are paid, machines of a new level of efficiency and programmability can be attained".

Most existing machines, except those using Transputers, have higher values of g than is ideal.

Valiant argues that the BSP model can be implemented efficiently on a number of technologies if they have a low value of g. He also argues that a large number of important high performance numerical algorithms can be implemented directly in the model. This includes: parallel matrix algebra; data broadcasting and prefixing; FFT on a butterfly graph; and parallel sorting.

If the concepts underlying Valiant's work are taken on board, parallel processing for high performance computation is here to stay!

Section 7 Heterogeneous and Multi-user Systems

A number of problems are apparent when considering heterogeneous and multi-user systems which are not addressed by the simple considerations above.

A networked workstation cluster is for instance an inherently dynamical multi-user system and the code developed faces several difficulties if good performance is to be obtained. Extra problems arise in a heterogeneous assembly. Important factors are:

1. Network traffic - which is dynamically varying can slow down communication between some or all processors
2. Differences in network bandwidth - on a multi-branch LAN
3. Differences in processor speed - in a heterogeneous or quasi-heterogeneous cluster

4. Local cpu load - may change dynamically, e.g. workstation owner may run a big sequential job at high priority
5. Local memory load - may change dynamically, e.g. workstation owner may start a big edit or backup job leading to disk paging and loss of performance
6. A workstation may be switched off or become inaccessible for some reason, requiring checkpointing and program restart with a different processor configuration. We suggest how the checkpoint file can be stored.
7. Different number representations - in a heterogeneous cluster translation between binary representations takes extra time

Constraints Consider a problem of size NMAX*NMAX distributed over p processors, and ignore communication costs. The speed of the i'th processor is s_i, so the time taken in the calculation on the i'th processor is $\propto w_i/s_i$ if it is given an amount of work w_i to do. Constraints are as follows:

$$\sum_{i=1}^{p} w_i = NMAX^2,$$

$$t_i \propto w_i/s_i,$$

$$t_p \propto max\{w_i/s_i\} = max\{t_i\}.$$

When balance is achieved

$$t_i = t_p \propto \frac{NMAX^2}{\sum_{j=1}^{p} s_j}; \forall i.$$

the virtual machine is behaving like a single processor, thus

$$w_i \propto s_i t_p; \forall i.$$

The aim is clearly to minimise t_p for a given problem by changing w_i.

Chapter 7 Programming by Passing Messages

Many real-world processes (simulations), and also computational problems based on mathematical algorithms are inherently parallel if coded in the right way. A message-passing architecture permits the following types of programming.

1. A single program can be divided up into parts, with nodes executing different parts of the program
2. several separate programs can be executed at once on different sets of nodes.
3. a single program can be executed simultaneously by different nodes emulating a SIMD machine.

Each processor has its own "private" memory, requiring the explicit transfer of data. When information that is stored on one node is required by another node, the originating node must send the information and the node that requires it must receive it.

There are three main components to the programming model for developing applications for a distributed-memory parallel computer:

1. computations are performed by a set of autonomous parallel processes with probably different clocks
2. data accessed by one process is private from other processes, and variables are not shared, and that includes common blocks
3. processes communicate and synchronise by sending messages

This is the basis of the "Communicating Sequential Process" CSP model of programming (Hoare 1978 and 1986) which is required at the lowest level for any "loosely coupled" architecture.

Section 1 Nodes appear fully connected.

We have previously discussed the kind of physical topologies used in parallel computers. None of them are completely connected because it would add enormously to the compexity of the system making it impossible to scale up. The compexity should be constant, that is as the system is extended no new links should be added to existing nodes, except those on the edges of the network.

However, as far as the programmer is concerned, most modern parallel computers *do* appear to be fully connected. This is done using communication logic similar to that in modern telephone systems. When you make a call a set of switches are closed which establish a dedicated circuit for your conversation. In a similar way a path can be set up between any two nodes of the parallel system, and data sent over it. No extra processing time is required on the intermediate nodes, only the overhead in establishing the path is increased (latency). This is illustrated below.

Our "virtual machine" for which programs are written will therefore be a fully connected all-to-all network. Bear in mind that this will be an even better model in the future with very fast virtual communication channels and well balanced computation to communication

speeds (e.g. on the Inmos T9000 Transputer with its internal Virtual Channel Processor (VCP), the Kendall Square KSR1 with data flow hardware, and the Meiko CS-2 with its multi-level switched network).

When we discussed Amdahl's and Gustafson's Laws we used the parallel overhead P(p). We saw that to keep the overhead low a large amount of floating point arithmetic at a rate t_p seconds per operation needed to be carried out for each communication at an asymptotic rate of t_c seconds per byte, because usually $t_p/t_c << 1.0$ for current systems. The time for communication is however not constant, but is made up of a constant latency (S) plus a term which depends linearly on the number of bytes (n) of data being sent.

$t = S + t_c n$

Some of the latency is due to hardware, and some is in software protocol and extra bytes sent as message headers.

In real systems, because of the physical interconnect and protocols used, the actual behaviour may be far from linear, especially if many communications are being done simultaneously.

Section 2 Message Passing in simple terms

Consider two office workers in different parts of the country. If office worker Jane needs to get information quickly to office worker Frank then she could use the telephone. To be able to make the call Jane needs to have Frank's telephone number, which depends on which office he is in. She must then dial him, wait for a reply, and include the information required in the conversation.

Using the Fortnet programming environment (Allan et al. 1988, Allan 1992) we would therefore have the following code on two processors

proc n	proc m
call check(m) - rings m	call wait(n) - waits for a call and picks up the phone only if it is from n
call send(m,nbytes,data,ti) - sends nbytes of data of type data ti to m	call receve(n,nbytes,data,ti) - receives
call receve(...) - receive reply	call send(...) - send reply

This would actually be coded as follows:

```
inode=jobid()
if(inode.eq.n)then
 call check(m)
 call send(m,nbytes,data,ti)
 call receve(...)
else if(inode.eq.m)then
 call check(n)
 call receve(n,nbytes,data,ti)
 call send(...)
end if
```

In the above subroutines ti indicates the type of data e.g. 'D' for double precision, 'I' for integer, 'S' for single precision real etc.

A "job" in a loosely-coupled parallel system is a sequential part of a complete parallel task (an application program). The job runs on a single system node. To make most efficient use of a loosely-coupled system you must design your task as a set of jobs that can be executed concurrently (in parallel).

Like people working in offices, each worker has his own files and cooperates on exchanging information rather than keeping multiple copies of the files. Multiple copies may be helpful to speed up a particular job, but waste space and cause problems of coherency if someone updates a particular piece of information but does not broadcast that fact.

Jobs are executed "asynchronously", i.e. there is no timing relationship between them, which is why heterogeneous processor architectures with a different clock speed on each node can be used. Communication serves both to exchange code and data and to synchronise jobs at the point of exchange. This is called the "loosely synchronous" programming model.

Section 3 Brief description of the Fortnet Harness

Fortnet (Allan et al. 1988) is a portable message passing interface which is designed to interract with other software, such as a vendors propriatory system supplied with hardware. It is a simple, efficient interface and has been found to be useful in a number of applications. Whilst these lectures use Fortnet syntax extensively, it is not unique. Other environments are discussed in a later chapter, and there is currently an initiative involving US and European users and vendors to define a standard Message Passing Interface (MPI) using experience gained of older software.

The Computation Model The abstract model which we use divides a parallel task into a number of computational jobs (concurrent sequential threads which send and receive data to other threads). The communicating sequential process (csp) paradigm (Hoare 1978) is used to transfer data between jobs. Jobs are logically connected to all other jobs and no fixed topology is assumed in the programming interface.

Single or multiple jobs may be placed on each processor. This placement can be controlled by a configuration file which is read at task startup time. In a UNIX environment several jobs may run on one (possibly multiprocessor) workstation and communicate through UNIX

shared-memory constructs (e.g. Silicon Graphics, SUN, Apollo or Stardent systems). Others communicate using UNIX 4.2BSD sockets. An introduction to UNIX interprocess control is given by Leffler et al. (19??). The UNIX interface software used in much of our work is tcgmsg (Harrison, 1991) which forms a communication layer below the Fortnet user interface. An alternative is the PVM harness (Geist and Sunderam 1991) or the PVMe implementation (Richelli 1992). The configuration file also indicates if the standard Ethernet driver or faster FDDI, SOCC or Ultranet network driver is to be used.

The Fortnet Interface. The Fortran-77 callable subroutines relevant to these lectures are listed below, others are documented in previous publications and in the user manual (Allan, 1992).

Message Passing and Strong Data Typing

WAIT(n) — wait for job n in the parallel task to set up a virtual channel of communication

CHECK(m) — set up a virtual communication channel to job m

SEND(n, nbytes, data, ti) — send nbytes of data from buffer data to job n along the existing channel.

RECEVE(m, nbytes, data, ti) — receive nbytes of data into buffer data from job m.

The routines SEND and RECEVE take a single character flag ti (the type indicator) which indicates the type of data being handled as follows: ti = 'A' (character), 'I' (integer), 'R' (real single precision), 'D' (real double precision), 'C' (complex single precision), 'Z' (complex double precision), 'U' untyped. Mixed data types in a single message are not yet available (but c.f. Geist and Sunderam, 1992).

Strong data typing is essential if data is to be transferred between systems with different number representations. Although many now adhere to the IEEE floating point standard there may still be problems with byte ordering in storage of integers. In the current work we have *not* used any data translation so as not to add extra overheads to the communication which might confuse our conclusions. We thereby restricted ourselves to use clusters of machines with a common data representation. The common ones are: highest significant byte first (SUN, IBM, HP, Stardent and Apollo) or least significant byte first (Silicon Graphics).

Timing

CPU(time) — returns current cpu useage for this job (user + system), in double precision variable time, to UNIX system accuracy, usually 1/60 th or 1/100 th of a second.

WALLTIME(wtime) — returns current elapsed (wall) time in double precision variable wtime.

STATS(flag) — controls the internal collection of event information. Flag is a character variable which can take the following values: flag = 'ON'| 'OFF'| 'PRINT'| 'RESET'.

The 'PRINT' option causes output to the standard output device of time in seconds, since 'RESET', used in message passing events by the calling job.

MSTATS(twait, tcheck, tsend, treceve, tseq) — returns the same information as above into the double precision arguments. This information can then be used by the calling program to make decisions about further execution as will be demonstrated below.

Event Tracing

debug(mode) — set the mode of event trace output to disk files. The mode can take values as follows: mode = 'ON'| 'OFF'| 'RESET'| 'TOGGLE'.

On each job that switches debug on, one line of output is put into a trace file at the beginning and end of each message-passing routine, plus some additional book-keeping information. The trace file name is fortnetnn.trf where nn is the jobid of the calling job. The Fortran-77 logical unit used to access this file is defined in the Fortnet parameter file so that it does not conflict with user defined units. The default is lutrf=9.

A static profiler has been provided to integrate the trace file event information. In addition to the profiler, an interface to the Oak Ridge National Laboratory Trace software (Heath 1990) has been provided. The latter is part of the ORNL PICL harness (Geist et al., 1990) and has been found to be very useful in studying the behaviour of a number of algorithms, and in static load balancing (Owens, 1991).

Profiling output of just part of a typical four-node run looks as follows:

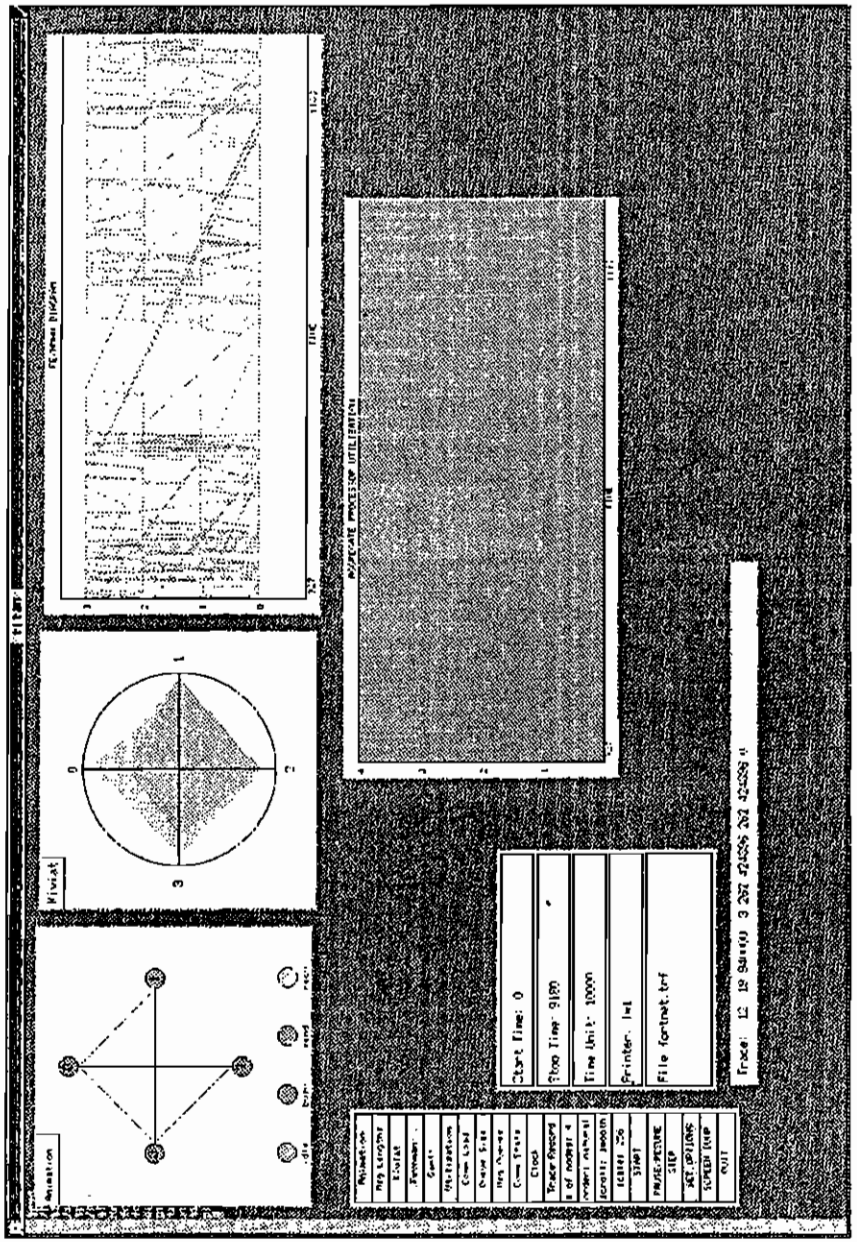
```
***** Fortnet Profile Facility *****

.....Fortnet timings in seconds
proc, receve,    send,      wait,     check, sequential
0,        0,       0,         0,         0,          0
1,   10.556,    7.431,    5.3807,    8.8028,     58.522
2,    9.577,    9.6561,   6.1389,    9.0043,     56.123
3,    9.8711,  11.332,    5.3873,    5.9699,     57.762
4,    7.2879,   8.8104,   8.1899,    7.3773,     58.460


.....number of calls
proc, receve, send, wait, check, sequential
0,      0,    0,    0,     0,          0
1,    260,  260,  260,   260,       1040
2,    371,  371,  371,   371,       1484
3,    371,  371,  371,   371,       1484
4,    260,  260,  260,   260,       1040

.....diagnostics
processor 0 terminated
processor 1 terminated  doing sequential code
processor 2 terminated  doing sequential code
processor 3 terminated  doing sequential code
processor 4 terminated  doing sequential code
```


Figure 3



Section 4 Blocking/non-blocking

Blocked asynchronous message passing

In the Fortnet programming environment jobs are explicitly "blocked" (synchronised) by using the CHECK/WAIT routines shown above. After doing this a "virtual channel" exists between the two synchronised processors and transmission of data can take place. The virtual channel exists until another different one is established with one of the two jobs as its end point. This virtual channel mechanism can be implemented on a great variety of computers of the architectures discussed previously. It can be done by setting switches, routing a path through links, setting semaphores, or by mapping directly onto virtual channel hardware such as in the Inmos T9000 transputer.

On other systems there are several different types of message passing available. What we have just illustrated is referred to as blocked asynchronous message passing with virtual channels, and is the type preferred on transputers. When a process issues a request to receive a message it then halts further execution until a message of the correct type has been received. If the message is not yet available the process calling RECEVE will remain idle until the message arrives. If the message has already been sent then the process only halts whilst it is accepted from the network into memory. SEND just puts the data from the source node into the transmission network and then returns control to the program. This is why we say the execution is asynchronous, because it does not have to wait for the receiver to pick the message up. This means that the source does not know that a message has arrived, only that it is on its way. The area of memory used to hold the data can be used again.

Getting the correct message, i.e. one from the correct processor, is assured in Fortnet by using CHECK/WAIT. This is what provides the loose synchronicity of the application. Additionally we could also ask for a further piece of information identifying the type of message using TCHECK/TWAIT (see Allan 1992a).

Non-blocked asynchronous message passing

If we reconsider the office worker analogy then a blocked message would correspond to Jane stopping work until she has been able to phone Frank and obtain the information she requires. A more realistic situation is that she would continue working on another part of the job, and then return to the original problem and try to phone again later. A better way, which does not require continual polling, is to leave a message for Frank to call back. This corresponds to non blocked message handling.

In some situations Jane could want information from several sources and would post a number of requests. Message types are used by some systems to put messages into pigeon holes so that the receiving processor can take them out in a different order. The message type is just a user-defined integer number, it is not strictly required in a CSP programming style. This is like an office memo system where instead of telephoning, one worker leaves a message for another to act on at a later time, it may be sufficient to identify the sender but a context could be useful too.

It is usually more efficient to handle message without blocking, but complicates the scheduling of work. A process notifies the transmission network of its request. A process can continue

work until it requires the information it has requested. If at that point the message has still not arrived it will have to wait in the usual way. Sending can also be done in an unblocked way. The source notifies the network of its wish to send data from a particular area of memory. It can then continue useful work in the knowledge that the data will be sent some time in the future, but it does not know when. If the area of memory needs to be used for some other purpose however it must wait until the message has been copied into the network otherwise a coherency problem will arrive.

If the system has DMA hardware to do communication, non-blocked transmission can actually be done in parallel with other work, and the overheads may be hidden effectively. In Fortnet there is currently no provision for unblocked sending of this sort. It is however possible to post the CHECK operation ahead of time using RCHECK or RTCHECK. This sends a request message into the network. At the point when real communication is required a blocking CHECK or TCHECK routine must still be used, which just examines the network to see if the correct virtual channel has been established and the target process synchronised, after which communication can proceed.

Interrupt-driven message handling

Another method of handling messages is called interrupt driven message handling. This is similar to the unblocked mode because the calling process continues to execute until the message arrives. At that time however it is interrupted and execution switched to a specially written handler process which should deal with the incoming message immediately.This type of message handling is not present on all MIMD machines at the user level, but is typical of the behaviour of many systems at the operating system or hardware level where an atomic response to certain signals is assured.

Handling messages in spawned threads

An alternative to the non-blocked and interrupt driven message handling is to use threads. A thread is another executing job which is started concurrently with one on the calling node. It is typically started with a command such as the UNIX FORK or SPAWN which replicates the parent process. The child can then conditionally do some alternative work, such as waiting for data communication or i/o which can appear to be completely non-blocked since the parent still carries on executing. A JOIN or similar command (WAITCHILD) would normally cause the parent to wait until the child has finished its job and then combine the two jobs once more.

In Fortnet a completely non-blocked receive of data would be coded this way:

```
...
inode=JODIB()
call NXTAG(newjob)
call SPAWN(newjob,'receve_nb',5,inode,m,nbytes,data,ti)
c do some more work
...
c now wait for data to be there
call WAITCHILD(newjob)
```

```
...
```

The source code for receve_nb would be

```
subroutine receve_nb(parent,m,nbytes,data,ti)
call TCHECK(m,parent)
call RECEVE(m,nbytes,data,ti)
end
```

The source code for the sending job would be

```
...
call TWAITANY(iproc,n)
call SEND(iproc,nbytes,data,ti)
...
```

The spawned process would start on any available processor, but in preference should be not too far removed from its parent. A similar programming style is used in the Mach or OSF/1 operating system with pthreads (parallel threads), and in the 3L Parallel C and Parallel Fortran-77 languages for transputers.

Section 5 Some examples using blocked asynchronous message passing.

Example: calculating pi

To illustrate how code is modified to execute in parallel we will look at a simple integration procedure using the trapezoidal rule method. Consider the following example.

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

The approach would be to divide the area into a number of equally spaced intervals and assume each strip forms a rectangle. The value of the function at the mid point of the strip is taken to be the height of the rectangle. The more strips used, the more accurate the calculation will be as illustrated in the diagram:

The sequential Fortran-77 code to integrate the statement function f(x) would be:

```
integer n,i
real w,x,sum,pi
f(x)=4.0/(1.0+x*x)
open(1,file='in.dat',status='old')
read(1,'(i5)')n
w=1.0/float(n)
sum=0.0
do i=1,n
 x=w*(float(i)-0.5)
 sum=sum+f(x)
```



```
end do
pi=w*sum
write(6,'('' pi = '',g12.5)')pi
end
```

In parallel the code would look something like this:

```
integer n,i,nnode,inode,nbytes
real w,x,sum,pi,temp
data nbytes/4/
f(x)=4.0/(1.0+x*x)
inode=JOBID()
nnode=NUMJOB()
if(inode.eq.1)then
 open(1,file='in.dat',status='old')
 read(1,'(i5)')n
 do iproc=2,nnode
 call send(iproc,nbytes,n,'I')
else
 call receve(1,nbytes,n,'I')
end if
w=1.0/float(n)
sum=0.0
do i=inode,n,nnode
 x=w*(float(i)-0.5)
 sum=sum+f(x)
end do
if(inode.ne.1)then
 call send(1,nbytes,sum,'S')
else
 do iproc=2,nnode

  call receve(iproc,nbytes,temp,'S')
  sum=sum+temp
 end do
 pi=w*sum
 write(6,'('' pi = '',g12.5)')pi
end if
end
```

The routine JOBID returns the id of the local instance executing process which calls it, this is numbered from 1 upwards. NUMJOB returns the total number of executing jobs.

Example: Broadcasting data

In the above example the value of n needed to be broadcast to all nodes. This part of the

code is serial in that processor 1 must read in the data and then send it to each node in turn. To simplify the sending step, and to allow it to be optimised for different types of processor networks the concept of broadcasting is introduced. The BRCAST subroutine requires to be told a root, which is the jobid of the processor doing the sending. All other processors which call BRCAST with inode != root will receive the data. The initial loop can therefore be replaced by a single call

```
call brcast(1, nbytes, n, 'I')
```

On a particular machine the BRCAST function will be implemented in the most efficient way, and the programmer does not have to worry about it, for instance on a hypercube architecture it will almost certainly use a binary spanning tree to reduce the complexity from n operations to log2(n). In this way the message passing is "encapsulated" or hidden from the programmer. All other "global" operations could also be provided in this way, as we shall discuss later.

Example: loop-level decomposition

Many coarse-grained applications can be parallelised at the loop level. They either have a single outer loop which can be decomposed, or a loop further insied the program which has enough iterations to keep all the processors busy. Examples would be in calculating molecular integrals in quantum chemistry, or doing partial wave sums and angular momentum coupling in atomic collision theory.

In the pi example we unrolled the calculation in the main loop simply across the processors

```
do i=inode,n,nnode
  ...
end do
```

This could alternatively be coded as:

```
do 5 i=1,n
if(mod(inode,nnode+1).ne.i)goto 5
  ...
5 continue
```

This is alright if each iteration of the loop takes the same time. In some instances we require to have an iterative procedure which may take more or less time depending on the loop index encountered. It would therefore be bad if one processor executed all the longest calculations and some form of load balancing is required. The simplest method is to use a so called "farming" technique. As a processor wants to do an iteration, it requests the next value of a loop counter which has not already been used. A central service must be available to maintain this loop index and increment it at each request. Fortnet provides such a service in the form of a Server process running concurrently with the application and the loop could then be re-coded as follows:

```
integer counter
6 i=COUNTER()
if(i.gt.n)goto 5
...
goto 6
```



```
5 continue
```

As an example we consider a Gaussian type integration with the computation of the integrand done by subroutine integl() at each point being of unknown duration. N points are used, the weights W and points X are computed by *all* processors at the start in routine gaussl (duplicated work), and the results are collected by process 1 at the end (sequential work).

```
integer counter, myi(n), y(n)
...
call gaussl(n,x,w)
inode=jobid()
myk=0
6 i=counter()
if(i.gt.n)goto 5
myk=myk+1
myi(myk)=i
c do integrand at points
y(myk)=integl(x(i))
goto 6
5 continue
collect results
if(inode.eq.1)then
  do i=myk+1,n
    call waitany(iproc)
    call receve(iproc,4,k,'I')
    call receve(iproc,8,y(k),'D')
  end do
else
  do i=1,myk
    call check(1)
    call send(1,4,myi(i),'I')
    call send(1,8,y(myi(i)),'D')
  end do
end if
c sum up results
if(inode.eq.1)then
  do i=1,n
    summ=summ+w(i)*y(i)
  end do
end if
c broadcast result
call brcast(1,8,summ,'D')
...
```

Another example is when records of a database have to be searched or matched against special features. The time taken can vary with the record (e.g. we may use pattern recognition

software for matching finger prints). It is then important that nodes process records in order rather than being given pre-allocated sections of the database.

Domain partitioning in Explicit Equation Solvers as applied to a CFD problem

The easiest methods to develop in parallel are those which have only coupling between neighbouring computational cells. Therefore, in principle, any explicit time-marching scheme can be modified to execute in parallel. There are many schemes which fall into this category e.g. Lax-Wendroff schemes, MacCormack's method, Runge-Kutta schemes, explicit TVD (Total Variation Diminishing) schemes. In general implicit schemes are more difficult because of the strong non-local coupling throughout the computational domain. Spectral methods and ones using FFTs also present difficulties. However many implicit methods have also been implemented in parallel e.g. incomplete LU factorisation, ADI (Alternating Direction Implicit) etc. These may be discussed in other lectures. With the explicit schemes the speedups obtained have been approximately linear to quite a large number of processors, this is not the case with implicit schemes at present.

There are several problems in implementing parallel solvers. In general iterative schemes can give rise to bad load balancing because it is not possible to know a priori where all the time will be spent. Some processors will be idle while others finish their iteration similarly to the loop case discussed above. This could happen in a shock-shock fluid interaction whereby the processor trying to resolve the interaction may need more iterations to converge. Adaptive grid methods, particularly those schemes which add or remove grid points can also cause problems. In general any method that would be adding or removing grid points to resolve flow features will cause bad load balancing unless some form of dynamic allocation if introduced. This is discussed in a later chapter.

By far the most popular technique for solving CFD problems is partitioning the grid into subdomains and allocating each subdomain to a different processor. This grid partitioning approach is frequently called domain decomposition. However this is a misuse of the terminology because a true domain decomposition solution would require no inter-process communication. This is not the case here because data must be exchanged at subdomain boundaries at each time step.

With most CFD problems it is also necessary for each process to receive global information. For example with explicit time marching codes the timestep is limited by the Courant-Friedrich-Lewy (CFL) condition. On a distributed system each process would determine the minimum time step required for its subdomain. However only one subdomain has the globally smallest timestep. As it is not possible a priori to know which this is, it is necessary to broadcast a value from every node to determine the smallest one. Of course if the problem were using local time-stepping to accelerate to a steady state solution it would be unnecessary to broadcast the global time step. However it would then be necessary to synchronise the solution at the end of any advance in time otherwise subdomains would begin to contain incorrect halo data.

Numerical schemes must be used to solve the complex set of equations governing the model. The usual prescription is to impose a grid of points on the geometry and then seek solution in terms of values of the quantities which control the flow, e.g. local velocities, temperature and

pressure, at the grid points. The partial differential equations can be translated into sparse algebraic equations using a suitable discretisation scheme. In the example code we used a finite-volume, operator split approach for discretisation of the 2D Navier-Stokes equations.

The field values between neighbouring grid points are coupled and the system is solved by an explicit time-marching procedure once the initial conditions have been specified. Such time-marching schemes are numerically unstable when regions of high gradients, such as shocks, are encountered. To stabilise the solution a Total Variation Diminishing (TVD) scheme is therefore incorporated into the McCormack predictor-corrector method by invoking a flux-limiting calculation before the corrector stage. Further details are described by Emerson and Poll (1990). In the benchmark we march for one hundred time steps, whereas real calculations would require in excess of 10,000 time steps for periodic motion to be set up in the cavity.

A typical TVD scheme would contain a dot product such as:

$$r_i^{\pm} = \frac{< \Delta Q_{i+1/2}, \Delta Q_{i-1/2} >}{< \Delta Q_{i\pm1/2}, \Delta Q_{i\pm1/2} >}$$

where $\Delta Q_{i+1/2} = Q_{i+1} - Q_i$ etc. This stabilisation term would involve knowing the values of r^{i-1}, r_i and r_{i+1} and results in a 5-point stencil. Many other schemes require a 5-point stencil is used for stabilisation too. For example the Runge-Kutta method popularised in CFD by Jameson uses a pressure sensor of the form:

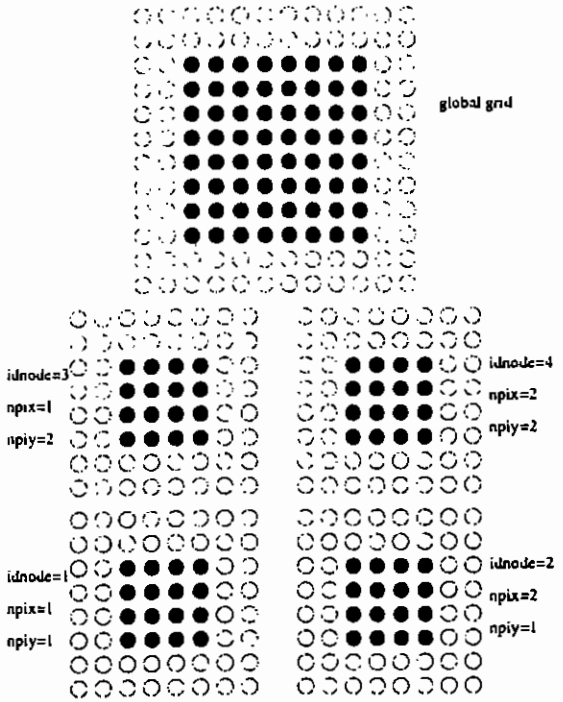
$$\nu_{i,j} = \kappa \frac{|p_{i+1,j} - 2p_{i,j} + p_{i-1,j}|}{p_{i+1,j} + 2p_{i,j} + p_{i-1,j}}$$

and it is necessary to evaluate the $\nu_{i+1,j}$, $\nu_{i,j}$, $\nu_{i-1,j}$ terms. This has important implications at a subdomain boundary because a so called "halo" of two points in each direction around any computational point must be known for the stencils. If the neighbouring points are on a different processor, which is the case near a domain boundary, data must be exchanged.

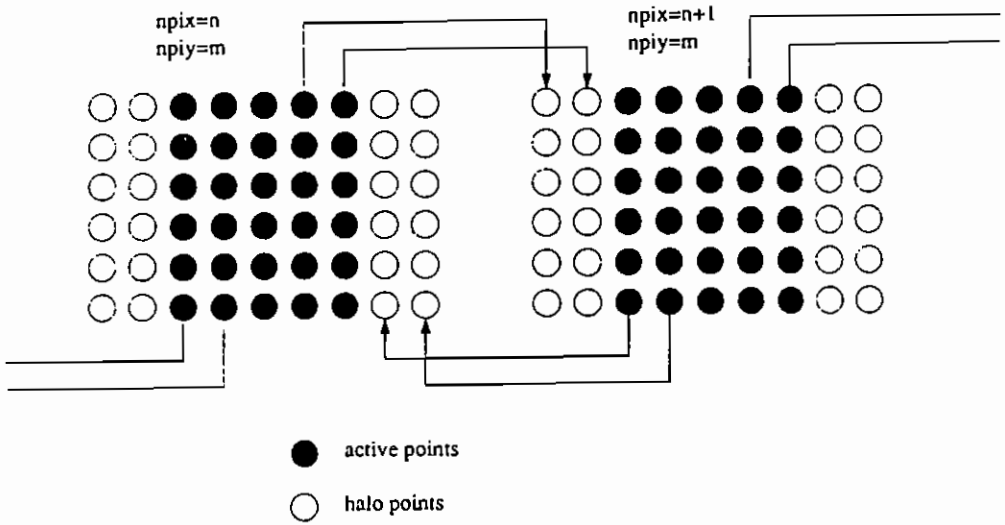
The application is unsteady compressible Navier-Stokes equations for high-speed flow over a rectangular cavity. The approach used to share the work out is to determine the number of active grid cells and allocate equal "areas" to each process. In the numerical model the physical geometry is inscribed within a logical rectangle of points. To ease the computations a buffer of two grid points is added to the rectangular domain to accommodate the finite volume computational stencils on all boundaries. This buffer of data replicated from a neighbouring domain is referred to as "halo" data. In the code we use a grid which is refined in the regions where we expect the flow variables to change rapidly, e.g. near solid boundaries and in the cavity shear layer. This is topologically equivalent to a regular grid.

The computational process involves: establishing a grid of points fitted to the cavity geometry; initialising the flow field and then calculating the new allowable time step; applying the prediction, stabilisation and correction steps along the y and then x directions. These procedures are explicit in that the calculations of the new value at each grid point can proceed independently. The computations are highly vectorisable and highly parallelisable. Synchronisation of the computation is required before starting another iteration either to calculate global norms of the solutions or to calculate the time step for the next iteration.

The code is parallelised using the grid partitioning strategy. The computational grid contains NMAX active points (excluding the buffer region) along both the x and y axes. The grid is initially cut into equal-area squares of size m=NMAX/NPROCX points along the x axis and n=NMAX/NPROCY points along the y axis. The total number of processors available is given by the product NPROCX*NPROCY. In the figure we illustrate how a small global grid might initially be distributed amongst 4 processors assuming two processors are allocated to both the x and y directions. The partitioning is further discussed below.



The computations require details of both global and local data. The time step for the explicit scheme is calculated by determining the largest flow velocity at any of the grid points in the computational domain. The largest permissible time step is first calculated for each of the subdomains. These values are then circulated to the other subdomains, compared, and the minimum permitted global time step adopted on all subdomains. In a similar manner, an estimation of the norms is made which requires calculation on a single subdomain before being circulated to the other subdomains and accumulated into global values. This would be used as a test for convergence a steady-state flow.



How do we share out the work so that each process gets an equal amount of work to do and an evenly balanced load is achieved ?

As an illustrative example we will consider the cavity problem. We will assume that the number of "active" cells is m*n and that p processors are available (the term active is used to describe cells used at the inflow, outflow and solid boundaries). The simplest approach would then be to determine the number of active cells and divide this up into p equal pieces. It is then necessary to decide which way to split up the computational domain for allocation to each processor. In general there will be several alternatives. The computational domain could be split into p horizontal strips or p vertical strips or a combination of both. There are advantages and disadvantages to all approaches and it may be necessary to try each combination to find the best approach to a particular problem. In general the treatment of boundaries causes a load imbalance which is difficult to correct for. The inclusion of the cavity causes a significant imbalance on the loading because of the solid wall boundary. In practice if the loads are balanced to within 10% we feel we are doing well. This is further discussed below.

The subdomains are labelled according to two conventions which depend on whether grid based or global data are being exchanged. For the halo exchange the subdomains are mapped onto a two dimensional logical mesh of size NPROCX by NPROCY with indices npix and npiy which describe their relative positions in the global geometry. Exchange of halo data along the x direction takes place between subdomains whose npix indices differ by one. Similarly, exchange of halo data along the y direction takes place between subdomains whose npiy indices differ by one in the initial decomposition. There is no halo data exchange between the left hand and right hand subdomains or the top and bottom subdomains in the two dimensional logical mesh.

Implementing a 2D domain partitioning scheme is more difficult than emplying just a vertical or horizontal decomposition approach. There are however advantages. The numerical computation is better with nearly square blocks, and less halo data is involved so communication delays are minimised. The amount of data to be transferred in a 1D layer decomposition is $\propto NMAX * (p-1)$. In a 2D regular block decomposition however it is only $\propto 2 * NMAX * \left(\sqrt{p}-1\right)$ in the symmetric case. This is important since current processor speeds greatly exceed communication speed as will be seen from the results presented below. This argument extends to higher dimensional systems.

As a codeing example the norm subroutine is written for message-passing. For global data circulation the subdomains map well onto a logical ring of processors. In passing the halo data for the finite-volume template we attempt to do parallel communication in two steps for each direction. This works if the underlying network has sufficient bandwidth. The code assumes a logical ring of processors to, for instance, calculate the global norm as follows:

```
SUBROUTINE NORM
INCLUDE 'parms'
INCLUDE 'comms'
Check if norms of solutions are to be computed this
c iteration
IF(MOD(Ntim,Nresid).NE.0)RETURN
c The temporal L1 norm (the biggest change in a
```



```
c variable at any grid point over an iteration)
c is calculated for each of the conserved variables.
rnorml(1:mmax)=0.0
rnormo(1:mmax)=0.0
rnormg(1:mmax)=0.0
c rnorml: local L1 norm for grid points in this subdomain
do 210 m=1,mmax
  DO 150 nrec=1,Nrect
    DO 120 j=Jstar(nrec),Jstor(nrec)
      DO 110 i=Istar(nrec),Istor(nrec)
        erra=Qold(i,j,m)-Q(i,j,m)
        erra=ABS(erra)
        rnorml(m)=max(rnorml(m),erra)
110     CONTINUE
120    CONTINUE
150   CONTINUE
210 continue
c rnorml: global L1 norms initialised to local value
c Rnormg(1:mmax)=Rnorml(1:mmax)
circulate the local norms around the subdoamins
c assuming a ring topology and determine the global norms
c rnorml: local norm to be sent to next subdomain in ring
c rnormo: local norm to be recived from previous subdomain
c in ring
IF(NPROC.GT.1)THEN
  DO 350 np=1,NPROC-1
   CALL CPU(timsta)
   IF(MOD(Idnode,2).EQ.0)THEN
    CALL WAIT(Nodem)
    CALL RECEVE(Nodem,MMAX*Nbyter,Rnormo,Typer)
    CALL CHECK(Nodep)
    CALL SEND(Nodep,MMAX*Nbyter,Rnorml,Typer)
   ELSE
    CALL CHECK(Nodep)
    CALL SEND(Nodep,MMAX*Nbyter,Rnorml,Typer)
    CALL WAIT(Nodem)
    CALL RECEVE(Nodem,MMAX*Nbyter,Rnormo,Typer)
   ENDIF
    CALL CPU(timsto)
    Timcomm=Timcomm+timsto-timsta
c compute global norm and copy local norm ready
c for forwarding to next processor
    Rnormg(1:mmax)=MAX(Rnormg(1:mmax),Rnormo(1:mmax))
```

```
    Rnorml(1:mmax)=Rnormo(1:mmax)
350     CONTINUE
   ENDIF
   IF(Idnode.EQ.1 )WRITE(6,99001)Ntim,Rnormg(1:mmax)
99001 FORMAT (' iter ',i3,':',4F12.8)
call forflush(6)
END
```


Chapter 8 Benchmarking using the CFD code.

The FLOW code is one of a set of reduced application code benchmarks, or kernels being developed in collaboration with the U.K. National Physical Laboratory as part of their EC Esprit III funded PEPS project. It can be used to assess the suitability of a range of parallel processing systems for solving demanding application problems. The kernels all have their origin in production codes that are used in fields such as computational fluid dynamics (CFD), financial transactions, image processing, process control and neural networks.

The CFD benchmark is controlled by a single parameter, NMAX, defined above, scaleable in powers of 2, that sets the problem size which can be changed so that breakpoints or threshold points under which the behaviour of the architecture changes can be tested. The code was implemented for concurrent execution from the start, using subdomain partitioning and the communicating sequential process (csp) model of programming (Hoare 1978). It is implemented in the Fortnet harness to be portable.

Preliminary results have been obtained on a number of multicomputers. The results confirm the expected behaviour that as more processors are thrown at a given size of problem, the communication costs increase relative to fixed total computation costs. On the other hand, increasing the job size for a given number of processors increases the efficiency (Gustafson 1988).

Section 1 Results

A Measure of performance In this application the execution time of the sequential code scales with the size of the grid, NMAX, so $t \propto NMAX^2$. In the ideal parallel implementation, on p identical processors, $t \propto 1/p$ thus in the absence of parallel overhead $\overline{M_p} = NMAX^2/(t_p * p)$ where $\overline{M_p}$ is a constant proportional to the single processor performance. With parallel overhead $\overline{M_p}$ decreases as p increases, and increases as NMAX increases, reflecting Amdahl's and Gustafson's Laws respectively (Amdahl, 1967, Gustafson, 1988).

The best that we can do therefore is to compare $M = p * \overline{M_p}$ for different machines and look for the highest value for a given size of problem.

In considering the times for parallel execution the best performance (i.e. speedup with 100% efficiency) should correspond to using the measured value of M for the single processors as a measure of the speed, s, of that processor. Thus the measured value is

$$\overline{M_p} \leq \sum_{i=1}^{p} M_i/p.$$

and the actual efficiency is obtained by comparing the measured performance (computed as above) to this ideal best performance.

Homogeneous Workstation Clusters. We have observed behaviour of the code on three types of workstation cluster: a system with three IBM RS/6000 model 520s; a system with four HP/9000 model 720s and one HP/9000 model 735; and a system of two Apollo DN10000s having a total of five processors. All systems are linked by a local area network (Ethernet) which has high nsf traffic. The IBM 520 workstations are in addition coupled by dedicated optical fibre SOCC links, and Ultranet. The HP model 720 systems are coupled by FDDI.

It has been possible to measure both the cpu time used by each UNIX process, and the elapsed time from the system clock on which the process is running. Measurements of elapsed time may include interference from other users and network traffic which will cause the process to be interrupted, unless we can guarantee single-user access. This was not possible and we in fact aimed to study a more typical situation in this work. The measured cpu time does not however reflect time spent waiting for system resources, such as sockets, semaphores, i/o etc. We measured this independently in ping-pong tests, and took the best results approximating single user conditions. These are reported in the next table using Fortnet with the indicated LAN and interface, together with the ratio of elapsed time to cpu time for short messages (of relevance in this application) and the asymptotic message passing rates (c.f. manufacturers' literature). All times are in microseconds unless stated otherwise.

system	a	b	c	d	asymptotic rate [Mb/s]	ratio elapsed/ cpu
IBM system 520s						
ethernet	1540	0.469	1571	0.794	0.792	1.02
SOCC	2010	0.096	1595	0.289	2.60	0.793
PVMe setxmode(1)	630	0.186	???	1.004	0.840	1.64 *
PVMe setxmode(2)	1620	0.259	1220	0.272	1.88	0.753 *
PVMe setxmode(4)	1300	0.219	2650	0.463	1.466	2.04 *
Ultranet	4996	0.0371	3444	0.0803	8.74	0.689
HP system 720s						
ethernet	361	0.149	375	0.917	0.939	1.04
FDDI	348	0.147	1135	0.206	2.83	3.26
other systems						
apollos	2010	0.548	7370	2.46	0.33	2.82
multi-processor systems using UNIX shared memory						

iris2	995	0.128	410	0.0999	4.39	0.78
apollo2	905	0.0838	285	0.0861	5.88	1.03

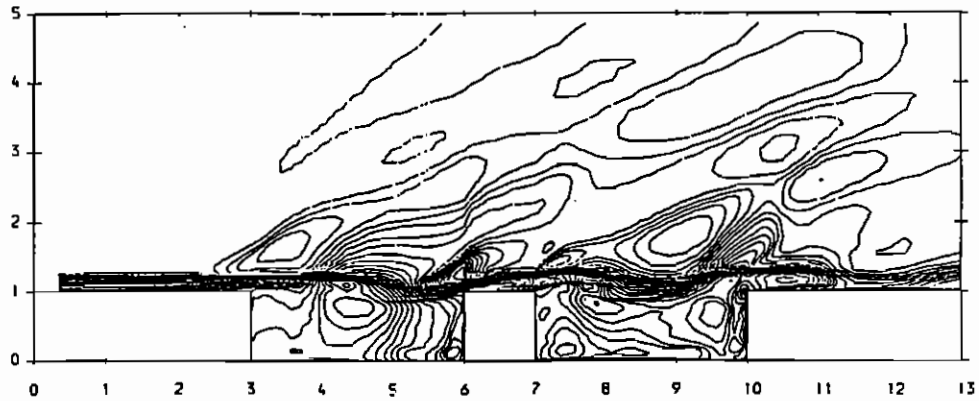
PVM (Geist and Sunderam, 1991) is one of a number of software harnesses like Fortnet (Allan et al. 1988) and tcgmsg (Harrison 1991) which give an easy and reliable interface to UNIX sockets and shared memory system constructs. The SETXMODE command was introduced into PVMe (Richelli, 1992) to bypass PVM daemon processes and control the synchronicity of access to the SOCC driver in IBM installations.

These parameters will be used in a system scalability model.

Overall time for message transfer in one direction is

$t = a + bn + c + dn$ microseconds

where n is the number of bytes in the message, a and b parameterise the cpu time, and c and d the elapsed time. This assumes a straight line fit to the data. A straight line is however not strictly correct, and the best performance peaks at between 32k and 1M bytes, roughly suggesting a useful upper limit to the size of system buffers. In the current application messages are less than 4k bytes, and a straight line fit is optimistic, the curve being quite flat until the data terms exceed the latency.



On a busy sysem it is impossible to get an exact picture of the time spent in message passing in a real application, because of the variable elapsed time component. The above results were a best estimate. We therefore quote both average cpu time and total elapsed time in the next table for the FLOW program to show the state of system load. It gives average performance measure and efficiency for homogeneous workstation clusters in runs of 100 iterations of a problem of size nmax=64. Columns are comp: average computational cpu time; comm: average comunication cpu time; elaps(m): measured total elapsed time; elaps(e): total run time estimated using table 1 (see text); %util: ratio elaps(e)/elaps(m); %eff: parallel efficiency; Mbar: average performance value; M total perfomance value as described in the text. All times are in seconds.

	comp	comm	elaps (m)	elaps (e)	% util	% eff	Mbar	M
HP/9000 systems with Ethernet								
tcs-hp1	41.34	0.04	80.30	41.4	51.6			98.99
2x1	20.56	2.76	32.05	26.2	81.2	79.0	78.2	156.4
3x1	13.34	3.5	29.87	20.5	68.6	67.3	66.7	200.0
4x1	9.94	3.68	35.1	20.2	57.5	51.2	50.7	202.8
2x2	9.44	4.1	26.4	17.8	67.3	58.1	57.3	230.1
HP/9000 model 720 workstations with FDDI								
2x1	20.46	2.33	29.79	30.39		68.1	67.4	134.8
3x1	13.36	3.2	22.7	26.9		51.3	50.8	152.3
4x1	9.9	3.26	25.3	23.79	94.0	43.5	43.0	172.2
2x2	9.59	3.93	21.5	26.3		39.3	38.9	155.7
IBM RS/6000 model 520 workstations with ethernet link								
tcs-ibm2	61.57	0.18	92.65	61.75	66.6			66.33
2x1	29.31	6.72	59.2	42.9	72.4	72.0	47.8	95.52
3x1	19.95	9.79	59.2	39.7	67.1	51.9	34.4	103.1
IBM 520 workstations with Serial to Optical Channel Converter								
2x1	29.26	5.83	59.0	39.7	67.3	77.7	51.6	103.2
3x1	19.24	8.64	92.6	34.7	37.5	59.3	39.4	118.0
IBM 520 workstations with Ultranet								
2x1	29.16	12.84	52.07	50.9	97.7	60.7	40.3	80.6
3x1	19.76	16.99	70.37	48.5	68.9	42.5	28.2	84.5
Apollo DN10000 multi-processor workstations with ethernet link								
apollo1	56.46	0.14	56.6		100			72.37

apollos 1+1	29.57	10.07	75.88	68.0	89.7	41.6	30.1	60.2
apollos 2+1	19.17	9.87	67.58	56.9	84.2	33.2	24.0	72.0
apollos 3+1	15.52	10.79	71.27	56.7	79.6	29.4	18.0	72.2
apollos 2+2	14.87	9.58	66.09	57.5	77.9	27.5	19.9	79.6
apollos 3+2	14.82	10.73	84.28	55.8	66.2	20.3	14.7	73.4

Key to systems:

tcs-hp0 — Hewlett Packard model 750
tcs-hp1 to 4 — Hewlett Packard models 720
HPs — cluster of one Hewlett Packard model 750 and the four model 720 machines linked by Ethernet and FDDI. All i/o goes to a RAID disc system on tcs-hp0 resulting in heavy ethernet traffic under typical conditions
apollo1 — Apollo DN10000 3 processor model
apollo2 — Apollo DN10000 2 processor model
Apollos — cluster of the two DN10000 systems linked by Ethernet, with a total of 5 processors
tcs-ibm2 to 4 — IBM models 520
IBMs — cluster of the three IBM model 520 machines linked either by Ethernet, SOCC optical connection or Ultranet

Code on all IBMs was compiled using xlf -O. Code on HPs uses f77 +O3 +OP4 +OPunroll. Code on Apollos f77 –O.

Typically the elapsed time in communications should be related to the cpu time used by the ratios given in the first table, if there was no other network traffic. The relationship is not however a direct one due to the fact that average cpu times are reported. For a layer decomposition, outer layers pass data only to one other processor, whereas inner layers pass to two neighbours, thus average cpu time is proportional to 2*(p-1). The elapsed time however is the maximum rather than the average result and is therefore related to twice the two-processor time for p>2. Here we assume that pairs of processors can interact independently on small systems with no collisions between messages. This assumes that packets are moving very fast on the network and all the overheads are in the device drivers and protocol. Similar arguments apply to the 2D block decomposition and are summarised:

p	comm cpu	best elapsed
2x1	1	1
3x1	4/3	2

4x1	3/2	2
5x1	8/5	2
6x1	5/3	2
2x2	1	2
3x2	7/6	3

In column 5 of the next to last table the cpu component of the communication is multiplied by by the ratios given to represent the true elapsed time and then the computation time added to give an estimated total run time. This represents the best that can be expected in a single user environment from the given application. The percentage utilisation is then the estimated run time compared to the measured run time, and reflects *other* system or network useage if all our assumptions are true. This is justified by the close agreement between some of the values in columns 4 and 5 of the second table. Only the values for the FDDI are difficult to explain.

In a real system the effect of elapsed time can, in principle, be reduced by allowing the processors to work on other tasks but, if these use the network the situation is made worse. The estimated run time was used to compute the performance measure M reflecting single user and therefore best results.

Column 7 of the table contains an estimate of parallel efficiency calculated for this application. It decreases very rapidly as more processors are added due to high inherent network latency.

Multi-processor Workstations A number of manufacturers offer multi-processor workstations to increase potential cpu power. Performance is achieved by using the shared memory to transfer data, which can be done at effective speeds up to 10Mb/s, allowing for software overhead and internal bus contention, or by using parallel compiler technology to unroll the program structure, typically by distributing (tiling) loops, see below.

We have observed behaviour of the code on two systems: a three-processor Apollo DN10000; and a four-processor Silicon Graphics 4D/220 GTX.

Results are now presented in a similar way to those above. Fortnet again is used as an interface to the tcgmsg software (Harrison 1991) which controls the UNIX shared memory data transfer to simulate message passing between separate jobs in the parallel task. Scaling of such systems is typically better than the network solution due to apparent improved communication performance via shared-memory access.

	comp	comm	elaps (m)	elaps (e)	% util	% eff	Mbar	M
Silicon Graphics system, iris2								
1x1	90.08	0.14		90.22				45.47
2x1	49.1	5.84	63.19	59.5	94.2	75.7	34.4	68.8
3x1	36.21	6.94	54.17	48.6	89.7	61.8	28.1	84.3

2x2	27.38	4.12	93.07	34.7	37.3	64.9	29.5	118.0
4x1	32.60	9.38	58.35	49.3	84.5	45.7	20.8	83.1
Apollo DN10000 system apollo1								
1x1	56.46	0.14		56.6				72.37
2x1	29.94	4.70	39.78	39.5	99.2	71.7	51.9	103.7
3x1	18.97	5.24	31.54	29.6	93.9	63.7	46.1	138.4

Key to systems as in the previous table except: ris2 — Silicon Graphics 4D/220 GTX four-processor model. Code on SGIs f77 -O2 and on Apollos f77 —O. The SGI operating system automtically balances processor load by swapping all jobs. The system has four processors of differing speed, but this cannot be seen from any tests we made.

Scaling Laws. If the scaling behaviour of the code is known, as it is in the present case, speedup can be estimated based on observation of a two-processor system. The computation time of the code must be known, together with the communication time and communication parameters for the appropriate message sizes. In the FLOW code the amount of computation is approximately constant, but the message passing scales in a known way. Based on measured parameters for two processor systems it is therefore possible to predict the performance of systems with more processors, and for larger problem sizes. For the HP systems with FDDI with 3, 4 and 5 processors predictions yield M values of 142.7, 173.6 and 197.9 respectively. This is quite close to the best estimated values. Since the computational work in the code scales as NMAX*NMAX, whereas the communication scales as NMAX, clearly performance is better for larger problem sizes. It is estimated for instance to yield M values of 160, 193, 242, 283 on 2, 3, 4, and 5 HP processors connected with FDDI for a problem of size NMAX=128.

Some differences between the predicted and measured performance arise from imperfect load balancing due to boundary and geometry effects. This is the subject of other work (Garner et al., 1993), see below.

Closely Coupled Systems Some similar measurements were made on closely coupled systems as indicated in the table below:

NMAX	p	comp	comm	elaps(m)	% eff	Mbar	M
Parsytec Multicluster T805							
128	2x2	1625	352.5	1948	85.5	2.1	8.41
	1x4	1623	175	1798	92.6	2.27	9.11
	3x3	700	210	910	81.3	2.0	18.0
	3x4	534	165.8	700	79.3	1.95	23.4
Intel iPSC/860							
128x64	2x1			178	90	23.0	46.0

	4x1			103	78	19.9	79.5
	2x2			102	78	20.1	80.3
	8x1			58	69	17.65	141.2
	4x2			59	68	17.35	138.8
KSR KSR1 using loop tiling (early system)							
128	2			254	98	32.3	64.5
128	4			143	87	28.6	114.6
256	2			1001	115 (*)	32.7	65.5
256	4			518	112	31.62	126.5

Conclusions It was found that in the FLOW code, even with a small grid size, useful performance could be obtained from a small number of identical workstations. Performance is limited by both the ultimate speed and the latency of the connecting network and better speedup was found in multiprocessor systems with shared memory and in closely coupled parallel systems. Some systematic measures of system performance have been given for this code which can be adapted to others with a similar data decomposition and message passing scheme. Such parameters allow performance of workstation clusters to be predicted from a knowledge of processor performance and simple message-passing 'ping-pong' tests between two processors.

Whilst it is tempting to make comments about the relative performance of workstation clusters of different types, this would not be entirely fair to the manufacturers concerned, due to the hidden effects of the loaded network and multiple users discussed in the text. We therefore leave it to the readers to draw their own conclusions.

Some comments are however relevant to the so-called high-speed networks offered at not negligible expense. Whilst it is true that asymptotic performance of such a network can be good (several times better than Ethernet), there is a large class of application which transfer many relatively short data packets between the parallel components, say less than 2k bytes. For these applications the high message latency or startup cost of the higher speed network can mean little or no improvement over the standard Ethernet connection.

As far as absolute performance goes, only a comparison of the run times with contemporary supercomputers is meaningful. The FLOW code, taking the same parameters as above, delivers a flow mark of M=1591.7 on a single Cray Y-MP/8I processor. This is equivalent to 129.3 Cray MFlops, a ratio of 13:1. It is expected that on workstations a ratio of around 10:1 is likely due to the lack of square root hardware etc. The Cray is delivering only around half its peak performance, probably because of indirect addressing in certain areas of the code which is not fully optimised for vector hardware.

Chapter 9 Programming environments.

Until now we have given examples of programs written using a CSP (Communicating Sequential Process) model with a portable harness interface called Fortnet which was developed at Daresbury Laboratory from 1987 until the present date. This is typical of other initiatives of computational scientists both in Europe and in the USA. Some computer manufacturers have also provided their own programming environments using message-passing via a library of subroutines which must be linked to the application code. A list of such software, and other software which works in a somewhat different way, follows (not exhaustive) :

Fortnet - Daresbury Laboratory U.K. (Allan et al. 1988; Allan 1992)

PARMACS - GMD mbH W.Germany and Argonne National Laboratory USA (Bomans and Hempel, 1990; Hempel 1991)

Express - Caltech and Parasoft Corp. USA (Kolawa)

PICL - Oak Ridge National Laboratory USA (Geist et al. 1991)

PVM - Emory University, Rice University and ORNL USA (Sunderam 1991; Geist and Sunderam 1992)

P4 - Argonne National Laboratory USA (Lusk et al.)

tcgmsg — Argonne National Laboratory and Pacific Northwest Laboratories, USA (Harrison 1991)

MPI — draft standard of a Message Passing Interface currently being discussed by a consortium on US and European members (see below)

Schedule - Argonne National Laboratory USA (Dongarra and Sorenson 1986, 1987)

Mach and POSIX threads — IEEE and Standards committees USA

Linda - Yale University USA (Leler 1990; Carriero and Gelernter 1988)

LGDF2 - Oregon Graduate Institute USA (Babb)

Other environments have taken the form of new languages for parallel programming:

Occam - Inmos Ltd., designed for the Transputer (Inmos)

Strand_88 - Strand Ltd. (Foster)

Vienna Fortran — University of Vienna, Austria

Fortran-D — DEC, USA

High Performance Fortran — USA and European computer consortium draft standard for a data parallel language based on Fortran 90 (see below)

We give only a brief introduction to a few of these environments. The first is parallel POSIX threads in shared memory as implemented by Kendall Square Research. The second is the Linda "tuple" space environment. The third is High Performance Fortran (HPF) as implemented on the CRAY MPP system.

Section 1 Mach and POSIX threads and virtual shared memory as implemented on the KSR1

We illustrate some concepts of virtual shared memory and Mach-style threads as defined in the POSIX draft standard P1003.4a (1990) and implemented on the Kendall Square Research KSR1 computer.

Tiling over DO-loops was done with the c*ksr*tile compiler directive. To do this one must indicate which loop index is to be tiled, and which variables are local to the loop iteration, all others being shared between iterations and processors. Some rearrangement of the loop ordering may be required for good performance but this is normally trivial. Such fine-grained programming techiques are capable of giving good performance. The programming style is moreover easy to use, and is very close in concept to proposed standards such as the High Performance Concurrent Fortran standard.

At the start of the program there is a single thread of execution (a UNIX process). If any parallel work is to be done another thread must be started either explicitly using a pthread library routine such as

```
call pthread_create(id, pthread_default,
&     function_name, args, ...)
```

or implicitly by starting a team of threads and using a compiler directive to invoke parallel do-loop unrolling. The latter technique is know as "tiling". Threads share all variables in common blocks, unless told otherwise, but other variables are local to the thread if started explicitly. Furthermore the i/o system is shared between threads. This is completely different to the CSP model of programming.

We illustrate implicit thread utilisation by re-coding the norm subroutine from the CFD example:

Codeing example of a how to initialise a team of threads on which to execute the application:

```
subroutine ksrinit
c Written by S. Breit, KSR, Jan. '92
include 'parms'
include 'comms'
character*80 commandline
call getarg(1,commandline)
read(commandline,'(i2)') nthreads
call pthread_procsetinfo(ncells,idummy,1,istatus)
if (istatus.ne.0)
& stop 'Bad status after call to pthread_procsetinfo'
if (nthreads.le.ncells) then
  print *,' Executing with',nthreads,' threads'
else
  stop 'Number of threads exceeds number of cells'
```



```
end if
Create team of pthreads, maximum number to be used
call ipr_create_team(nthreads,idteam)
end
```

Codeing example of a subroutine, summing of the global norms:

```
SUBROUTINE NORM
INCLUDE 'parms'
INCLUDE 'comms'
parameter (maxthreads=32)
c*ksr* subpage /ksrnorm/
common /ksrnorm/ redary(4,maxthreads)
Check if norms of solutions are to be computed this iteration
IF(MOD(Ntim,Nresid).NE.0)RETURN
c The temporal L1 norm (the biggest change in a
c variable at any grid point over an iteration)
c is calculated for each of the conserved variables.
rnorml(1:4)=0.0
do ithread=1,nthreads
  redary(1:4,ithread)=0.0
end do
c rnorml: local L1 norm for grid points in this subdomain
DO 150 nrec=1,Nrect
c itilsiz has been initialised in the main program
c*ksr* tile( j, teamid=idteam,
c*ksr*&  tilesize=(j:itilsiz(nrec)),
c*ksr*&  private=(ithread,m,i,erra,redvar))
DO 120 j=Jstar(nrec),Jstor(nrec)
  ithread=ipr_mid()+1
  do 210 m=1,mmax
    redvar=0.0
    DO 110 i=Istar(nrec),Istor(nrec)
      erra=Qold(i,j,m)-Q(i,j,m)
      erra=ABS(erra)
      redvar=MAX(redvar,erra)
110   CONTINUE
    redary(m,ithread)=max(redary(m,ithread),redvar)
210  continue
120 CONTINUE
c*ksr* end tile
150 CONTINUE
do ithread=1,nthreads
  rnorml(1:4)=max(rnorml(1:4),redary(1:4,ithread))
```

```
end do
c rnormg: global L1 norms initialised to local value
Rnormg(1:mmax)=Rnorml(1:mmax)
WRITE(6,99001)Ntim,Rnormg(1:mmax)
99001 FORMAT (' iter ',i3,':',4F12.8)
END
```

Explicit use of threads

In addition to implicitly using threads via compiler directives we can directly start a number of thread processes and then allow them to communicate, via a mailbox in shared memory. This is done with the help of "spin lock" constructs and barriers illustrated in the following piece of code which starts threads and then synchronises them:

```
program test
implicit real*8 (a-h,o-z)
implicit integer*8(i-n)
parameter (nprocmax=32)
c defaults for threads - this is provided by ksr
common/pthread_defaults/ipthread_attr_default,
&     ipthread_mutexattr_default,ipthread_condattr_default,
&     ipthread_barrierattr_default
logical checkd
common/flags_g/checkd(nprocmax)
integer bar_g
common/locks_g/mut_g(nprocmax),bar_g(nprocmax)
common/talk_g/nnode_g
character*80 commandline
integer work
external work
call getarg(1,commandline)
read(commandline,'(i3)') nthreads
nnode_g=nthreads
write(6,'('' starting mutex and barriers '')')
do i=1,nthreads
  call pthread_mutex_init(mut_g(i),ipthread_mutexattr_default,
&       istatus)
  call pthread_barrier_init(bar_g(i),
&       ipthread_barrierattr_default,2,istat)
end do
write(6,'('' starting threads '')')
do i=1,nthreads
  call pthread_create(inode_g(i),ipthread_attr_default,work,
&       i,istatus)
end do
```



```
write(6,'('' all threads started, waiting '')')
do i=1,nthreads
  call pthread_join(inode_g(i),ireturn,istatus)
end do
write(6,'('' threads finished '')')
end
c
integer*8 function work(inode)
implicit real*8 (a-h,o-z)
implicit integer*8(i-n)
if(inode.eq.1)then
  call waitany(iproc)
  call check(iproc)
else if(inode.eq.2)then
  call check(1)
  call waitany(iproc)
end if
work=0
END
c
subroutine check(n)
implicit real*8 (a-h,o-z)
implicit integer*8(i-n)
parameter (nprocmax=32)
logical checkd
common/flags_g/checkd(nprocmax)
integer bar_g
common/locks_g/mut_g(nprocmax),bar_g(nprocmax)
c set semaphore in global memory
checkd(n)=.true.
c barrier to synchronise threads
call pthread_barrier_checkout(bar_g(n),1,istat)
call pthread_barrier_checkin(bar_g(n),1,istat)
end
c
subroutine waitany(iproc)
implicit real*8 (a-h,o-z)
implicit integer*8(i-n)
parameter (nprocmax=32)
logical checkd
common/flags_g/checkd(nprocmax)
integer bar_g
common/locks_g/mut_g(nprocmax),bar_g(nprocmax)
```

```
common/talk_g/nnode_g
c spin lock waiting for any semaphore
5    continue
do iproc=1,nnode_g
  if(checkd(iproc))then
    call pthread_barrier_checkout(bar_g(iproc),0,istat)
    checkd(iproc)=.false.
    call pthread_barrier_checkin(bar_g(iproc),0,istat)
    return
  end if
end do
goto 5
end
```

Section 2 Linda

Linda is provided as an extension to standard imperative programming languages. It includes the concept of a "tuple space" into which data can be put together with certain key words (names). Each piece of data is of a particular type, integer, real etc. (strong typing, as in Fortnet). Data can be put into the tuple space and is then available for any other process to fetch or copy. Tuple space is therefore an implementation of virtual shared memory. There are only four extra commands in Linda:

out(t) - put the tuple of numbers t into tuple space

in(s) - fetch the tuple matching the template s from tuple space into the formal parameters in s (see below)

rd(s) - same as above, but reads the data, leaving a copy still in tuple space

eval(t) - This is the same as out(t) except that t is evaluated after, rather than before it enters tuple space. Eval therefore implicitly forks a new process to do the evaluation.

Tuple space is an associative memory. Tuples have no addresses; they are selected by IN or RD on the basis of any combination of their field values. Thus the five element tuple (A,B,C,D,E) may be referenced as the five element tuple starting with A, the five element tuple with B in the second place and E in the fifth, or any combination of appropriate element values.

To read a tuple using the first description we would write RD(A, ?w, ?x, ?y, ?z). This makes A an actual parameter, it must be matched, and w to z formals, whose values will be filled from the matched tuple. To read using the second description we write RD(?b, B, ?x, ?y, E) and so on. Associative matching is actually more general than this. Formal parameters or wild cards, may appear in tuples as well as match templates, and matching is sensitive to both the types and the values of tuple fields.

Consider distributing a simple loop

```
do i=1,n
  call work()
end do
```


This would be written in Linda using the following parallel construct

```
do i=1,n
  call eval('work loop',i,work(i) )
end do
do i=1,n
  call in('work result',i)
end do
```

This first loop farms out n copies of the WORK subroutine, each of which must terminate with the call

```
call out('work result',i)
```

This is a kind of semaphore which flags that the work has been done. The second loop is waiting for n of these to appear in tuple space which means all the work is finished. This is equivalent to starting n threads (see above) and then blocking while they terminate.

There are much more sophisticated things which can be done with Linda, however this brief review will suffice for now. A good reference is the one by Carriero and Gelernter (1988). Linda was modified for use in writing operating system code by Wm. Leler and the parallel PIX window system of the short-lived Cogent XTM Transputer-based workstation as well as its O/S was written in this so-called kernel-Linda.

Section 3 High Performance Fortran (HPF)

I will describe some aspects of HPF as implemented on the CRAY MPP system. HPF is an emerging shared memory programming standard. It is based on FORTRAN-90 and deriving from earlier experiments such as FORTRAN-D and Vienna FORTRAN among others. It will be released mainly as extensions for Fortran-90 compilers. The CRAY implementation will differ slightly from the final standard due to their early involvement and input to the MPI committee.

The model uses standard Fortran-90 with compiler optimisation directives. It is intended to provide an incremental approach to writing parallel code on MPP systems. Similar approaches with dialects of HPF are available also on KSR, DEC, Thinking Machines CM5, Convex, Intel, IBM MIMD systems and DAP, MasPar and Thinking Machines SIMD systems. The full HPF specification is somewhat more sophisticated than the examples below, it includes a multi-layer abstract machine concept for mapping arrays onto processors and controlling memory allocation.

The system assumes a data sharing model, and programmers must add data sharing directives to distribute data among the processing nodes. A work-sharing model can also be specified using the normal Fortran-90 array syntax. If the data parallel directives are too restrictive a DO SHARED directive can be used to control the indices and work distribution within DO loops. Low level synchronisation through barriers, locks and events is also available.

The programming model includes:

1. control for communication and synchronisation
2. specification of how and where data should be located and how and where work will be performed
3. work sharing and data sharing models with control of data and work alignment

Data sharing Data objects can be either "shared" or "private". Shared objects are accessible to all tasks and are generally distributed with a piece of the object in each processor's memory. Private objects are generally replicated with an independent copy in each memory. Large matrices should be shared, temporary variables such as loop counters private. For example:

```
common/xxx/A(131072),B(131072)
cdir$ shared A(:block)
cdir$ shared B(:block)
```

specifies that A(1) and B(1) are on the first node, and that A and B are aligned so that A(i) and B(i) are always on the same node. This would of course be a very good arrangement for vector style operations involving A and B which could be carried out in parallel.

One may optionally specify a block size as follows:

```
cdir$shared A(:block(32))
```

which implies distribution is required to be by blocks of 32 elements. This may mean that some processors are allocated a different number of elements if the size of A divided by 32 is not equal to the number of processors. A colon alone implies that the dimension should not be distributed, for instance in a two-dimensional array we may want the data distributed by rows:

```
dimension A(8192,512)
cdir$ shared A(:block(1),:)
```

Work sharing Work can be shared across the nodes with either Fortran-90 data parallel statements, or with standard Fortran-77 DO loops. Some algorithms are easily expressed using array syntax. In these cases parallel directives simplify the programming effort. In others DO loops may be simpler. DO SHARED directives applied to standard DO loops allow the programmer to directly control the indices and to align the data distribution with the work distribution.

Data parallel Using Fortran-90 array syntax, and the vector distribution mentioned above, we can easily distribute computational work:

```
dimension A(1024),B(1024),C(1024)
cdir$ shared A(:block),B(:block)
cdir$shared C(:block)
A=B*C
```

multiplies all elements of B by corresponding elements of C and stores the results in A elementwise, and does it in parallel.

DO SHARED directive DO loops can be either shared or private, just like variables. If loops are shared, then iterations are divided among the nodes. All iterations of private loops are executed on all the nodes. The programming model permits programmers to direct loops to be shared across a number of processors, and provides different scheduling mechanisms. These might be pre-scheduled with aligned work distribution, or self-scheduled.

In a pre-scheduled loop, iterations are done on nodes which contain the required data. This is the best option if there is no difficulty of load balancing. For example:


```
cdir$ doshared (K) on C(K)
do K=1,n
  A(K)= B(K)*C(K) + D(K)
end do
```

In a self-scheduled loop iterations are allocated adaptively at run time to idle processors. It is best used in MIMD style loops where load balancing is important. For instance:

```
cdir$ doshared (K,L) chunksize(64)
do K=1,n
  do L=1,Z(K)
    A(K,L)=B(K,L)*C(K,L) + D(K,L)
  end do
end do
```

Regardless of the value of Z(K) the chunks of work will be serialised and allocated to the nodes in chunks of 64 iterations. When a processor finishes its work it will request another chunk until all is completed. This is equivalent to the KSR Fortran concept of loop 'tiling'.

Section 4 Message Passing Interface (MPI)

The Message Passing Interface is a proposed interface standard to support a library of point to point message passing functions. This will be used for Fortran-77 and C language programming , and will form the basis of a standard high-level communication environment featuring collective communication and data distribution transformations. The standard proposes both blocking and non-blocking message passing between pairs of processes, with message selectivity by source process and message index. Provision is made for non-contiguous messages. Context control provides a convenient means of avoiding message selectivity conflicts between different phases of an application. The ability to form and manipulate process groups permits task parallelism to be exploited, and is a useful abstraction in controlling certain types of collective communication.

The MPI definition will address programming of MIMD distributed memory systems. The main advantage of establishing an independent worldwide message passing standard are full portability and ease of use. In a distributed memory communication environment in which the higher level routines and or abstractions are built upon lower level message passing routines the benefits of standardisation are particularly apparent. Furthermore the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide by hardware support, thereby enable scalability.

The design of MPI is making use of experience of existing message passing systems, rather than selecting or adopting any one of them. It has been influenced by work at IBM (Bala and Knipis, 1992; Bala et al. 1992), Intel's NX/2 (Pierce, 1999), NCube's Vertex (NCube, 1990), PARMACS (Bomans and Hempel, 1991; Hempel 1991; Hempel et al. 1992). Other contributions come from Fornet (Allan et al. 1988; Allan 1992), Zipcode (Skjellum and Leung, 1990; Skjellum et al. 1992), Chimp (EPCC 1991; 1992), PVM (Geist and Sunderam 1991); Sunderam 1990) and PICL (Geist et al. 1990)

Chapter 10 General programming techniques

Writing parallel algorithms is more difficult that writing sequential ones, therefore anything which can be done to make the job easier is important.

Section 1 Modular Programming.

Modular programming means breaking down the application into smaller parts, rather than just having a monolithic main program. The parts should be given sensible names which relate to what they are intended to do. For instance you might have a input phase, various computational phases and an output phase. You might also have a message-passing phase. Comments should be written into the code at each major step to explain what is happening. This might sound obvious, or redundant if you wrote the program yourself, but it is easy to forget what was intended and make a mistake.

Data should also be divided into structures (vectors or arrays) with sensible names. It is preferable to keep the same name in each subroutine if possible, this is especially true if common blocks are used and greatly aids readability and will make the job of parallelising a program much easier.

One of the wonderful capabilities of Fortran is to have common blocks with different data types, i.e. complex*8 in one part of the code and real*4 in another part. Fortran does not check strong typing. I admit this can be a useful trick for certain algorithms (e.g. complex fast Fourier transform), but from the point of view of modularity it should definately be avoided. It is better to pass it as a subroutine argument in this case with a clear comment about what is happening.

Section 2 Domain Decomposition.

Domain decomposition strictly refers to geometric decomposition of the data into separate independent computation regions. The term is however used loosely and in fact any geometric partitioning in which message-passing of boundary information will be referred to as domain decomposition. This was the case in our CFD example above.

Section 3 Control Decomposition.

There are two simple types of control decomposition:

1. Function distribution
2. Master-Worker distribution

In the function distribution the application is divided into functional entities on a coarse scale rather like a very heavyweight pipeline. This is sometimes referred to as a "conveyor belt". It is only possible to do this when there are a small number of "fat" processors which can be loaded with different executable modules (or told to execute different subroutines within the same executable), and data flows between. It is important that there be a large number of computational cycles to absorb startup overheads. There will inevitably be a serious problem with load balancing because it is highly unlikely that the different functions take the same length of time to execute.

In the Master-Worker distribution workers run on all nodes except number 1. They are sent data and receive requests for work from the Master process, and send their results back to the master. This is what we did in the Gaussian integral example, except that the Master did some of the work itself and the Slaves made their own choice of which points to integrate.

Section 4 Object-oriented Programming Techniques.

An object-oriented approach to programming provides a formal basis for organising the data structures and threads of execution of which a program consists. It is then possible to express decompositions clearly and avoid errors in algorithm design. A good introduction to the concepts is given by Martin (1993).

Object-oriented programming was developed as a technique to avoid problems associated with software complexity on sequential machines. The approach is easily applicable to large applications on parallel systems. Certain special languages, such as C++ and ADA, are available to support object-oriented programming although the design techniques can be used in any language, in our case Fortran.

The data structures and subroutines present in a large application must be decomposed into smaller parts, each of which strictly holds a certain type of data or performs a certain simple function on that data. This is called data abstraction. Once the application is decomposed in this way it is possible to state a set of rules about how the data and functions behave and are related. It is then possible to make changes to both the structure of data, and to the algorithms underlying the functions, in order to implement them optimally on new architectures, provided the rules are still obeyed. This is known as encapsulation. Debugging a program written in this way can be reduced to testing each function in turn, since it has a cleanly defined interface which can be rigorously tested.

Only a functional object can access the data objects, so information about the status of the data can be passed from one function to another (inheritance) and rules can control access to classes of data so that functions may only operate in a strict sequence (hierarchy). Functions may also call one another to build more complex operations (another form of inheritance whereby new functions or object classes are built from old ones).

Encapsulation means that the user of the program can treat each function as a black box and just hook them together in the right order. In fact the application programmer does not generally need to know either how functions are implemented, or how data is distributed. The functions can also be re-used in a different context, for instance a matrix multiplication subroutine may be required in a number of different applications. To enable this, functions can be built into a

library with a strictly defined interface. We have done this in our PARLANCE system (Allan 1992b), which will be briefly described in the last chapter of this tutorial.

Surprisingly the overheads associated with an object-oriented style are not too large. Calling functions with several arguments is very fast with modern compilers. I have written Fortran programs with several thousand lines just composed of a few hundred functions (rather than subroutines) which are linked in a hierarchical sense to, for instance, compute mathematical functions required in a certain type of integral evaluation.

Section 5 Reducing the Parallel Overhead

In writing parallel code it is important to minimise the parallel overhead so that good speedup and scalability can be obtained. Several general techniques can be applied to this:

1. Structure the algorithm so that only "local" communication is done and only pairs or small sets of processors need to communicate at any one time
2. Avoid any broadcast or global communication steps if possible — it may sometimes be faster to re-calculate rather than transmit data (the so called "direct" approach). Avoid implicit use of central services such as access to shared devices such as terminal screens or disks unless the system has a parallel i/o subsystem
3. Overlap communications by getting several sets of processors to communicate in parallel — this may require some awareness of the underlying network to achieve optimal behaviour. Clearly it is easier if the sets are small (e.g. pairs of processors)
4. Overlap communication with computation using non-blocked transmission or threads — this assumes the transmission is performed by separate DMA devices
5. Send or receive data in advance — this may require re-structuring the algorithm and including temporary variables as buffer space. Buffering to send long messages is also a good idea for both communication and i/o.
6. Avoid imbalance of computational load so that processors are not kept waiting
7. Write algorithms which can use underlying vector hardware (if available) on the longest possible vectors and high-level optimised BLAS

We illustrate some, but not all, of these techniques in the following examples.

We consider that a modular approach to designing applications, and the use of object-oriented techniques is extremely important. Not only the design of applications, but also the readability and useability of the code is improved. If you are working as part of a research team and the application is to be of long-term significance this is crucial.

Chapter 11 Examples of Load Balancing

The loop farming technique was already mentioned above where we used a counter service to decide which was the next available loop iteration to calculate. It was also used in the HPF language. For this to work the iterations must be independent and only the end results need to be accumulated.

We have more complex problems to deal with in the case of geometric domain decomposition, as in the CFD example above, which is now explicitly treated.

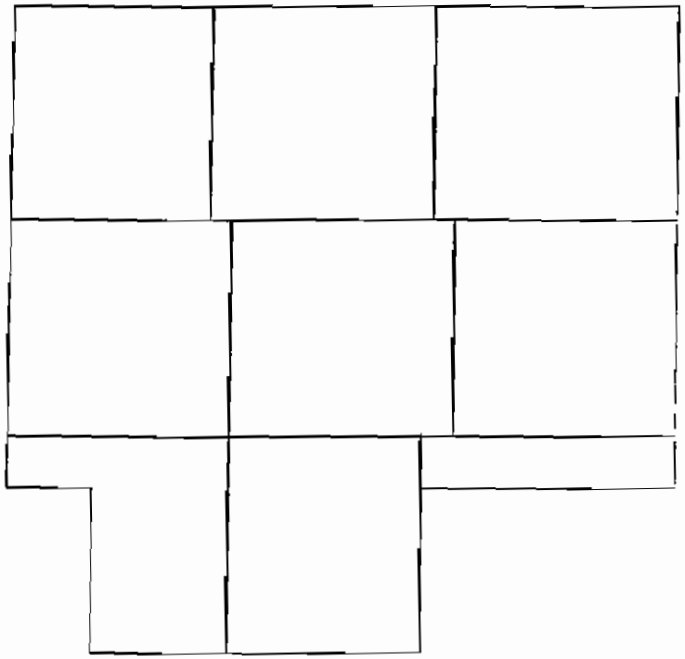
Section 1 Load Balancing Strategy for Domain Decomposition.

We have adopted an hierarchical approach to load balancing in a multi-dimensional domain decomposition. In 2D this amounts to alternately balancing in the x and y directions. The y dimension is divided into full layers (slabs), whilst the x dimension is divided into rectangular blocks within each layer. Balancing in x involves moving the vertical block boundaries in a horizontal direction independently in each layer, a parallel activity. Balancing in y involves exchanging information within a complete slab to find the total time spent in the slab calculation, and then moving the horizontal slab boundaries in the vertical direction to balance. In both cases the process is similar, and can be extended to more than two dimensions.

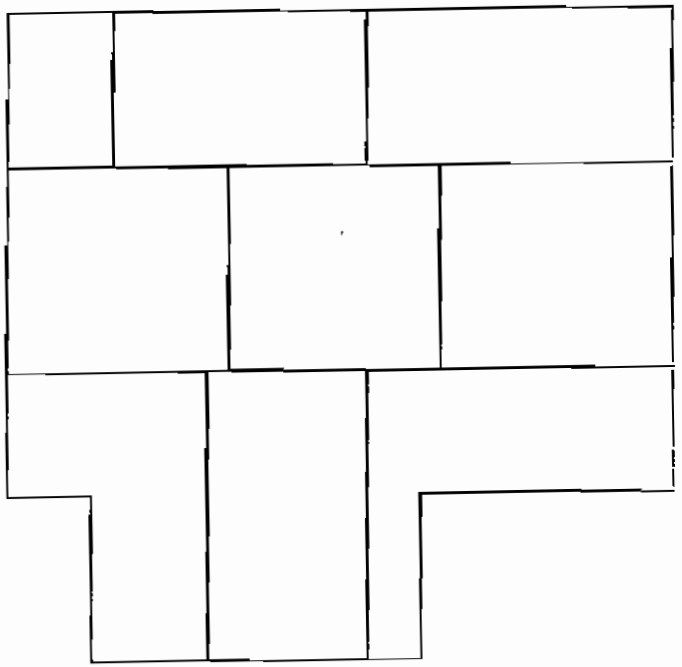
To determine which subdomain boundaries are required to move, the neighbouring domains communicate the elapsed (wall) time spent in calculating the previous iteration, including communication in sending and receiving data. The difference between this time and the average time for each three neighbouring domains indicates the amount of work (number of cells) which must be gained or lost before the next iteration. Domain boundaries are thus moved accordingly in each dimension independently of the others, halo data is updated, and the process continued. For simplicity of the implementation boundary movement is restricted to at most two grid points, so that existing routines can be used to build new halo data (see above).

A distribution of physical domains of a 2D problem near the start of a computation is shown below.

It can be seen that the domains in the lower corners have little work, since many of their grid points lie outside the computational region. In addition we ran the system on a Parsytec T805 Multicluster and simulated a system in which processor number 7 does some extra work to simulate a slower processor (which may be encountered in a heterogeneous or multi-user distributed environment).



After 100 calls to the balancing routine the situation had stabilised as shown below. The balancing could then be switched off in a production run. However in a multi-user environment it would be still necessary to test for imbalance every few iterations and switch the balancing operation on again if required.



Section 2 Mapping to unstructured meshes, and mesh refining

It is possible to map unstructured meshes into the abstract block decomposition used in the current problem. This is considered to be an extension of the orthogonal recursive bisection method discussed by Williams (1991).

Mesh refining techniques consist of adding or subtracting mesh points in physical regions of the problem with high or low activity, such as where turbulence starts or shock waves form. In the current method this is equivalent to adding more work into a domain handled by one processor. Clearly work must be migrated onto other processors in the manner suggested to give a new optimal distribution.

Section 3 Checkpointing and Fault Tolerance.

Fault tolerance is different from load balancing in that it indicates that one or more faulty processors can no longer participate at all in the solution and the connections to them are permanently discarded. This raises two problems:

1. How to detect that a processor is faulty — usually by a timeout on communication with it
2. How to restart the calculation without going right back to the beginning

The solution to the second problem is usually to invoke checkpointing; that is, data and program status are written to disk every N iterations so that the entire problem may be re-started from the last saved solution. In a workstation environment a simple objection to this scenario is that the disc which holds the data may be attached to the faulty processor, and writing to disc on all participating workstations is impractical.

We propose two solutions to this, analogous with current parallel disc and backup strategies. One solution uses cyclic checkpointing to discs on different workstations. If there are p discs the worst case is for the most recently used one to become inaccessible, a 1:p chance, and restart would have to take place from the dump made 2N iterations previously. The second solution uses a method similar to RAID technology. All p discs (or a subset of them) are used at every N iterations, but one of them contains checksum information so that, if any one becomes inaccessible, the entire data set can be rebuilt and only N iterations are lost. This however involves writing extra data every N iterations. The former strategy is therefore preferable on most systems since i/o with Winchester discs can be quite slow compared to processor speed and failure is in any case a rare event. Even if all except one processor is removed there is still enough information to restart without going right back to the beginning.

Chapter 12 Matrix Algorithms

We can try to think about some parallel algorithms with real square matrices. Starting with simple ones such as multiplication and transposition, and then progressing to inverse and the eigenvalue problems. We will continue to use the Fortnet environment for this study.

It is clear that there are various algorithms suitable for different distributions of data. The efficiency of a given sort of algorithm depends critically on the actual data placement. The algorithm can nevertheless be made to work in some form for completely general distributions. It is also clear that the best format of data for one algorithm will not necessarily yield good results for another.

Section 1 Matrix Multiplication

The following example is the kernel of a matrix multiplication A=B*C where all three square matrices of order N may be distributed across the whole machine.

In simple terms the serial algorithm is as follows, using Fortran-90 notation

```
do i=1,n
  v1(1:n)=b(i,1:n)
    do j=1,n
    v2(1:n)=c(1:n,j)
    a(i,j)=ddot(n,v1,1,v2,1)
  end do
end do
```

Ddot is a vector dot product basic linear algebra (BLA) routine (see VecLib documentation for instance). Note that the above separates moving the data into two vectors from the actual vector multiplication. The dot routine is clever enough to execute the following code:

```
do i=1,n
  do j=1,n
    a(i,j)=ddot(n,b(i,1),n,c(1,j),1)
  end do
end do
```

where the second argument n is the vector stride in the memory containing B.

In order to translate this into a parallel decomposition we should analyse the steps which are taking place. Going back to the first code, which has the explicit vector gather, we can see what is going on. The row and column which are used to form one element of the result matrix are first gathered into temporary vectors. If the matrix elements are distributed across several processors, with their local memory in an DM-MIMD system, then this may involve communication. We should ensure that the vectors are gathered onto the same processor, possibly the one which is to be used to store the result A(i,j). Let us first consider the case of matrices distributed columnwise, order N on n processors. Matrix A is replaced by a local column vector of which element A(i) represents A(i,j) on processor j. This is similar for B and C.

```
do i=1,n
  c ----- start gather
  do k=1,n
    do j=1,n
      if(inode.eq.k.and.inode.eq.i)then
        v1(k)=b(i)
      else if(inode.eq.j)then
        call receve(k,8,v1(k),'D')
      else if(inode.eq.k)then
        call send(j,8,b(i),'D')
      end if
    end do
    c ----- end of gather
    end do
  a(i)=ddot(v1,1,c,1,n)
end do
```

Note that it is better to make a separate gather routine as follows:

```
do i=1,n
  call gather(n,i,inode,v1,b)
  a(i)=ddot(v1,1,c,1,n)
end do

subroutine gather(n,i,inode,v1,b)
dimension v1(*),b(*)
do j=1,n
  do k=1,n
    if(inode.eq.k.and.inode.eq.j)then
    v1(k)=b(i)
    else if(inode.eq.j)then
      call receve(k,8,v1(k),'D')
    else if(inode.eq.k)then
      call send(j,8,b(i),'D')
    end if
  end do
end do
end
```

This means that the computational code looks very similar to that found on a vector machine using basic linear algebra routines (BLAS). We have again separated movement of data into a communication "skeleton" and the dot product into a computation skeleto. The gather routine could be easily modified for different distributions of the array B. If the array C is stored other than by columns it could also be called to gather elements of B into the "intermediate vector" V2.

The above code is somewhat complex. Almost all the code is doing communications, which proves that we must produce powerful skeleton routines to simplify this. It is also not very

parallel.

Consider now the case if array B is transposed so that it is stored with a row on each processor. The gather routine would then not need to do any message passing other than shifting a whole row from one processor to another. Let us investigate this strategy further:

```
subroutine gather(n,i,inode,v1,b)
dimension v1(*),b(*)
do j=1,n
  if(inode.eq.i.and.inode.eq.j)then
    v1(1:n)=b(1:n)
  else if(inode.eq.j)then
    call receve(i,8*n,v1,'D')
  else if(inode.eq.i)then
    call send(j,8*n,b,'D')
  end if
end do
end
```

The overhead used in transmitting a whole vector is less than doing it element-wise. At least n steps of this algorithm are now perfectly parallel. For the other steps a vector of data must be transmitted between two processors. It is clearly also possible to do these steps in parallel.

```
do i=1,n
a(i)=ddot(b,1,c,1,n)
call shiftup(b,n)
end do
```

The shiftup routine moves all rows of the distributed B array up one, and puts the top row on the bottom (called "wrap around"). This is done as follows:

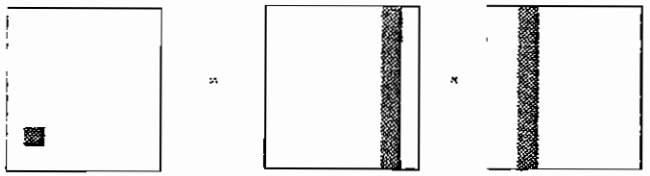
```
subroutine shiftup(b,n)
dimension b(*)
dimension temp(large enough ?)
ppl=inode+1
pml=inode-1
if(pml.eq.0)pml=nnode
if(ppl.gt.nnode)ppl=1
c do transfer in two steps
if(mod(inode,2).eq.0)then
  call send(ippl,8*n,b,'D')
  call receve(ipml,8*n,b,'D')
else
  call receve(ipml,8*n,temp,'D')
  call send(ippl,8*n,b,'D')
  b(1:n)=temp(1:n)
end if
end
```

Our matrix multiply routine is now very parallel, and it works in the case that A and C are stored column-wise and B is stored row-wise, This is equivalent to the matrix equation

$A = (B^{\dagger}) * C$

which is commonly met in quantum mechanical calculations in physics and chemistry.

Diagramatically the operation looks like:



```
transpose B
do i=1,n
  multiply columns on this processor to get
           one result element
  roll B horizontally to left
end do
```

The compexity of the algorithm (time) is:

$$Max\left[\frac{N^2(N-1)T_p}{p}, \frac{2N^2T_c}{p} + \frac{2NS}{p}\right]$$

where t_p is the average time in seconds for a + or * operation if the processor is scalar, t_c is the communication throughput between two processors in seconds per word (8 bytes ?), and S is the startup time for the parallel communication. In deriving this formula it was assumed that communication and computation can be overlapped and that all processors are able to communicate simultaneously.

There exist variants on this algorithm for matrices which are stored as square blocks. These are represented diagramatically below. We will suppose that the matrix of order N is distributed in square blocks, m*m of them, on p processors. Each block has (N/m)**2 elements of the matrix. One strategy due to Geoffrey Fox et al. (1988) is shown in the first two steps as follows:

```
broadcast B horizontally to right
do i=1,m-1
  multiply sub matrices on this processor
  broadcast B horizontally to right
  roll up C vertically
end do
```


Figure . Fox's Parallel matrix multiply algorithm

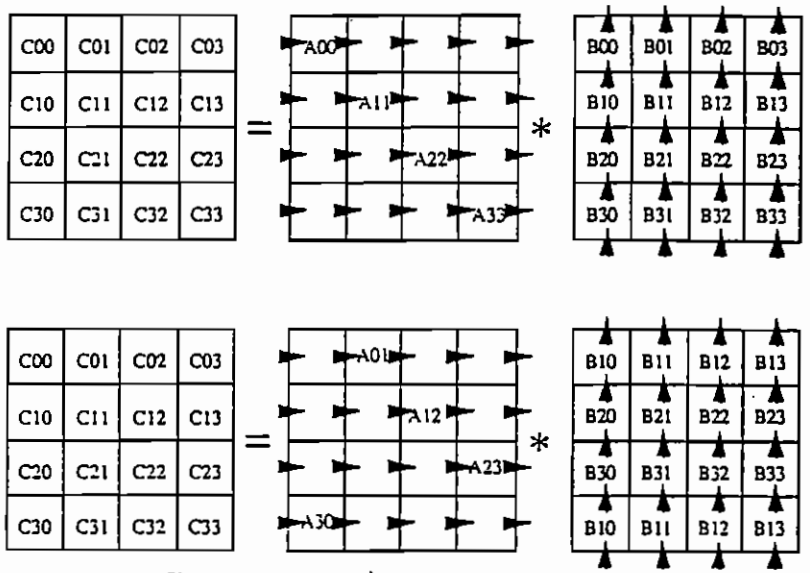


Figure 1st et 2nd cycles of Fox algorithm

It relies on broadcasting blocks from one processor to all others at each step for the B matrix, and rolling up the C matrix.

It is assumed that communication can be done in parallel so there is a total overlapping of the rolling up by the broadcast. Horizontal broadcasts can be optimised using a pipeline, and both directions go simultaneously. The complexity of this algorithm is:

$$Max\left[2\frac{N^3}{m^2}t_p, \frac{N^2}{m}t_c + \frac{m^2}{2}S + 2N\sqrt{\frac{m}{2}St_c}\right]$$

Another algorithm is due to Bernsten (1989), and is somewhat simpler:

```
do i=1,m
multiply sub matrices on this processor
roll B horizontally to left
roll C vertically upwards
end do
```

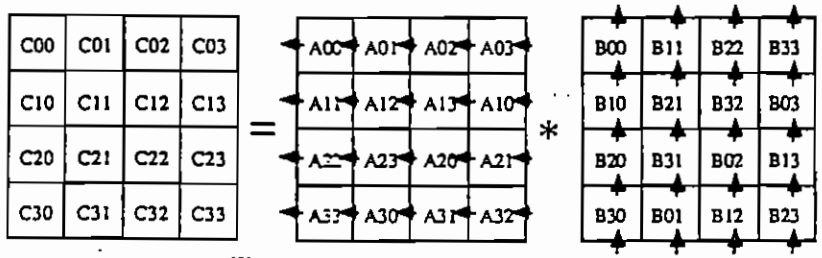


Figure 1st step data repartition for Bernsten's Parallel matrix multiply algorithm

The complexity is:

$$Max\left[2\frac{N^3}{m^2}t_p, \frac{N^2}{m}t_c + mS\right]$$

One lesson learned from the above analysis is that the way data is distributed in the machine is important. It is also possible to break the algorithm up in different ways and do steps in a different order to the intuitive one. Clearly the old sequential steps are usually not the best and new ones need to be invented. The matrix multiplication is a kernel code used in many applications. We would hope to extract maximum parallelism from such kernels, without having to re-write the entire application!

Section 2 Matrix Transpose

Transposition of matrices is also important in a number of applications, partly because it makes them simpler to do in parallel. It is for instance used commonly in doing a parallel 2-dimensional FFT based on having the 2D data again distributed columnwise across the processors. A 1D FFT is first done down each column in parallel. The matrix is then transposed and the FFT step repeated and the results transposed back. The backward transpose is not always needed if the data is to be back transformed at a later stage, intermediate steps could use the transposed matrix. The serial algorithm is illustrated as follows, you can imagine how to make it parallel using a transpose step.

```
c down the columns
ii=1
do 1 j=1,ny
  call tfft(A(ii),work,nx,trigx,ifax,nfacx,1,isign)
  ii=ii+nx
1    continue
c along the rows
do 2 i=1,nx
  call tfft(A(i),work,ny,trigy,ifay,nfacy,nx,isign)
2    continue
```

A general matrix transpose program is difficult to write, partly because the sequential version looks so simple, partly because it ought to involve no computation so that the overhead in doing parallel communications and computing offsets are disasterously highlighted. This appears to be a clear case where a hand-coded solution with knowledge of the distribution of elements is needed. Let us however examine the more general problem in a series of steps. Firstly the sequential version:

```
do i=1,n-1
  do j=i+1,n
    temp=a(i,j)
    a(i,j)=a(j,i)
    a(j,i)=temp
  end do
end do
```

Next a vector version:

```
do i=1,n-1
  k=i+1
  v1(k:n)=a(i,k:n)
  a(i,k:n)=a(k:n,i)
  a(k:n,i)=v1(k:n)
end do
```

If we again store the matrix A by columns this can trivially be expressed in a simple parallel form using our Fortnet routines, but there is no overlapped communication!

```
do i=1,n-1
do j=i+1,n
  if(inode.eq.i)then
    call send(j,8,a(j),'D')
    call receve(j,8,a(j),'D')
  else if(inode.eq.j)then
    call receve(i,8,temp,'D')
    call send(i,8,a(i),'D')
    a(i)=temp
  end if
end do
end do
```

Whilst the above program should execute and carry out all the communications required, it would be hideously slow because for each iteration of the loop on i all other processors are trying to communicate with the ith one, i.e. there is no overlapped communication. An improvement can be made by reversing the order of the do loops so that a number of sends (which do not block) are initiated while the receives are completing. One truly parallel matrix transpose can be done by employing a trick. That is to create an identical copy of the storage for symbol 'a' and call it 'b', this is done using the REALLOCATE routine from the PARLANCE library (see below). Now we can use dyadic (pairwise) communications to put the transpose of a into b, then do one big copy at the end to move it back.

```
call reallocate(b,a)
do j=1,n-1
  do i=j+1,n-1
    if(inode.eq.j)then
      call receve(i,8,b(i),'D')
```



```
    else if(inode.eq.i)then
      call send(j,8,a(j),'D')
    end if
  end do
  do i=1,j-1
    if(inode.eq.j)then
      call receve(i,8,b(i),'D')
    else if(inode.eq.i)then
      call send(j,8,a(j),'D')
    end if
  end do
end do
call pcopy(n,a,1,1,b,1,1)
call deallocate(b)
```

This version of the code should also work but it wastes memory and has an unwanted extra copy at the end. The interests of parallel programming at the algorithm level are to handle large systems, so this should be avoided.

This overlaps some communication because whilst processor 1 is talking to processor 3, processor 2 (which has already sent its data to 1 into the communication network) will be talking to processor 4 etc.

Another way to tackle this problem is to use a "wavefront" decomposition. This is of general use in a number of algorithms. Again we keep the matrix A stored by columns, although the technique would also work well if it were stored by square blocks (see below).

```
c run down diagonal
do i=2,n-1
iproc=i
  do jproc=1,i/2
    if(inode.eq.iproc)then
      call receve(jproc,8,temp,'D')
      call send(jproc,8,a(jproc),'D')
      a(jproc)=temp
    else if(inode.eq.jproc)then
      call send(iproc,8,a(iproc),'D')
      call receve(iproc,8,a(iproc),'D')
    end if
    iproc=iproc-1
  end do
end do
```

This algorithm indexes i down the diagonal starting from the top left-hand corner of the matrix. In the first step (i=2) processors 1 and 2 exchange data. In the second step (i=3) processors 1 and 3 exchange data. In the fthird step (i=4) processors 1 and 4 exchange data and

at the same time processors 2 and 3 exchange data. The wavefront gradually builds up so that at its peak the exchange is simultaneously between processor pairs (1,n), (2,n-1), (3,n-2) ...

This is clearly a good strategy except at the beginning and end of the algorithm (startup and shutdown overheads). It is also good for algorithms involving banded matrices.

In a transpose with column storage every node must actually communicate with every other one. At any one time there is a set of pairs of processors communicating. This should be a set which "covers" all n processors to reduce the overheads. There is a number of such sets and each will be referred to as a k-fold covering of n processors.

The number of different ways of covering p processors with k of them is

$M_k(p) = p!/((k!)^{**}N^*N!^*m!)$; N=p/k; m=(p-Nk)

where the exclamation sign denotes a factorial.

Higher schemes are possible for other algorithms. Of course if p is large, then ncover is *very* large illustrating a fundamental difficulty of global communications schemes - the inflation in data movement. Further solutions to this problem are outside the scope of this tutorial.

If permutation theory could be used to find connections (in the common block), the algorithm could be re-written as:

```
integer connect
common/cover/connect(p)
ncover=dyadic(p)
do i=1,ncover
  call covering(i,2,n)
  iproc=connect(inode)
  if(iproc.gt.inode)then
    call send(iproc,8,a(iproc),'D')
    call receve(iproc,8,a(iproc),'D')
  else
    call receve(iproc,8,temp,'D')
    call send(iproc,8,a(iproc),'D')
    a(iproc)=temp
  end if
end do
```

This has transformed the problem of devising communication strategies into one of sets and permutations. It still has the disadvantage however of sending single words in each transfer. It is therefore fairly certain that the best distribution of a matrix, if a transpose is required, is in square blocks so we can extend the above algorithm or the wavefront one to send a block at a time between processors and do a local transpose of each block.

Matrix transposition is nevertheless still a bad algorithm for parallel computation. It involves long-range communication which will not scale well as we increase the number of processors.

Section 3 Gaussian Elimination

Gaussian and Gauss-Jordan elimination are very fundamental matrix manipulations used in

solving a full system of simultaneous linear equations.

$\Sigma\ c(i,j)^*x(i) = y(j)$

(Note other methods are used for tridiagonal systems, which we do not discuss here). These algorithms are also used in finding the inverse of a matrix as will be further discussed below. It is useful to illustrate how the Gauss and Gauss-Jordan schemes can be implemented in parallel.

In the Gauss-Jordan method an appropriate multiple of the first equation is added to each of the other equations so that the resulting n-1 equations have zero coefficients for the x(1) term. (If the first equation has a zero coefficient for the term involving x(1), we must first interchange two equations to obtain one with an x(1) term as the first equation). An appropriate multiple of the first equation is then added to all equations to eliminate the x(2) term from all but one equation. This process is continued until each equation contains only one unknown, and the equations are solved. At each step the coefficient being used to eliminate other coefficients is called the pivot.

This algorithm has some inherent global properties, not easily suited to a distributed architecture. This can be demonstrated by noting that if we change any one coefficient in the matrix, we may change the nature of the whole solution dramatically. The algorithm can however be structured so that information can be made to move around a network of processors so that repeated local movement of data satisfies requirements both for global information and an efficient parallel implementation.

For reasons of numerical stability, as well as the reason mentioned above, the equations are re-ordered at each step so that the diagonal element, which we are going to use to eliminate the other elements in the column, is the largest coefficient in its column and in its row. This is referred to as "full pivoting". It is usual to compare only elements of a column on and below the diagonal or of a row on or to the right of the diagonal. An alternative "partial pivoting" is used in which only elements in the same column are compared below the diagonal. This is mathematically almost as good as full pivoting and is clearly easier to implement in parallel.

The alternative Gaussian method is also used in which only sub-diagonal values are eliminated. This reduces the matrix to upper triangular, rather than diagonal form. In the Gaussian method therefore the solution is not found directly. Only the last equation will have been reduced to just one coefficient, i.e. its solution has been found. The others must then be solved, starting with the (n-1)th equation by so called "back substitution". The solution of equation n is substituted into equation n-1 which is thus solved, and so on. This is some three times faster than full Gauss-Jordan elimination on a sequential computer when the inverse is *not* required. This is however not favoured as a method to use in practice, LU decompositions being much better to do the same job. The back-substitution phase is moreover difficult to implement in parallel. We will discuss it further below when considering the inverse problem.

In parallel the matrix can be distributed by columns "wrapped around" the available processors. This is illustrated in the diagram. The right hand side is included as an extra column of the matrix.

In step one processor one must find the pivot element (the largest absolute value) in column one. It must broadcast the position of this element and the entire first column to all the other processors. Because each processor has one or more whole columns the rows can then be interchanged (moving the pivot row up to the diagonal position) in parallel. The processors must now work down the column to be eliminated, starting from the diagonal, and for each row subtract its coefficient in the first column times the coefficient in the column being worked divided by the pivot value.

There is a long time spent waiting in this algorithm. For instance processor 1 will have finished step one before processor two, and then has to wait for processor two to find and broadcast the pivot position for step two and column two. This waiting time can be partly eliminated using a method suggested by F.Wray (1992). Processor two can be made to find the step-two pivot and do its broadcast before doing its step-one elimination. This can of course only be done approximately. Every processor should then receive a pivot and column, and if possible immediatly compute the next pivot and broadcast it with its related column.

In the final stages of elimination only a few sub-diagonal elements need to be eliminated on a few processors. The reduction in the amount of work at each step thus reduces the parallel efficiency (shutdown overhead). The algorithm is of course better for an order of matrix much larger than the number of processors involved.

Chapter 13 PARLANCE (PARallel Library And Network Computing Environment)

Which of the various algorithms will prove to be the most efficient depends on several issues. Firstly the size of the matrix compared to available memory dictates whether it must be handled in situ or not, secondly the speed of data transfer to other nodes compared to the individual processor speed will govern the choice of an algorithm which does redundant data movement or one which has expensive index calculations, loops and communications overheads for each matrix element. Thirdly the mapping of data onto the physical machine architecture is important. These issues cannot currently be decided due to the rapid evolution of hardware.

For the above and other reasons we are designing a user interface for parallel numerical algorithms which is called PARLANCE (PARallel Library And Network Computing Environment). It takes some ideas from object-oriented programming techniques, such as encapsulation, hierarchy and inheritance.

Take the different ways to transpose a matrix for instance. We can provide a single routine, called PMTRANS, which looks at how the matrix is distributed across the processors and just calls the most appropriate method. In order to do this a record must be kept of the physical data distribution of each object to be handled in this way. This is done by the PARLANCE data manager, which is called implicitly by each routine. The data manager can also control message passing, so that need no longer be done by the application programmer - it is encapsulated in the PMTRANS routine. At the start of the program however it is still necessary to specify the initial distribution of data, and to tell the system that matrix A is a distributed data object.

Matrix A can only be referenced implicitly through the PARLANCE library routines. Because this is done we can also refer to any element of matrix A at any time using its real two-dimensional index. This gives us back the concept of a virtual shared data space (memory) and we no longer have to remember which block of A is on which processor. Furthermore the numerical algorithms are free to internally change the storage distribution of A which would then be inherited by further algorithmic objects. For instance it might be that the block transpose were always chosen to be the most efficient method, and the matrix could be re-distributed by the data manager before it were invoked. PARLANCE can also be implemented directly on virtual shared-memory computers.

The data structures in PARLANCE are objects, and algorithmic methods are invoked by the library system to perform optimised functions which may be dependent on the object distribution or state of the run time system. Techniques of object oriented programming are used in building PARLANCE.

PARLANCE currently contains a large number of data management routines, communication "skeletons" and vector BLAS as well as a number of matrix algebra routines the internal workings of some of which were illustrated above. It is being extended. These routines constitute the mechanism by which PARLANCE addresses its data objects and encapsulates numerical operations.

We do not have time to go fully into the structure of PARLANCE, it is probably inappropriate in this tutorial anyway. I do however give a short example of some code written in this way.

The matrix multiply A=B*C

```
character*8 A,B,C
data A,B,C/'A       ','B       ','C       '/
call storebycol(A,...)
call reallocate(B,A)
call reallocate(C,A)
call pmtwist(B)
...
compute data in C
...
compute data in B
call extracta(A)
do j=1,n
  joff=index2d(1,j,n)
  do i=1,n
    ioff=index2d(i,j,n)
    call onea(ioff,iaproc,iacore,ti)
    if(inode.eq.iaproc)
  &    temp=pdot(n,b,joff,1,c,joff,1,ti)
    call puta(inode,1,temp,1,1,A,ioff,1,ti)
  end do
end do
```

The symbols A, B and C are names rather like the key words in Linda. They act as pointers to the data in the virtual shared memory space. STOREBYCOL does an initial distribution of A. reallocate copies the format for B and C respectively. PMTWIST changes the format so that it columns of B are referenced as rows — data will therefore be stored in B in the transposed position. EXTRACT and ONE interact with the data manager to set up local tables indicating the structure of A which are then inherited by the other routines. INDEX2D changes a 2D index into a contiguous 1D vector offset used by the data manager. PUT copies local data from a given processor into the global data space.

Because of the way the data is set up this will be parallel. It resembles the sequential algorithm, makes no explicit reference to message passing and will still work for another data distribution, although less well!

A final example is the Gauss-Jordan elimination scheme with partial pivoting for a coefficient matrix distributed by columns, one per processor. It is used to solve a set of linear algebraic equations of order N, as discussed above. B is the single right hand side and upon exit contains the solution vector.

```
subroutine gaussj(n,A,B,ndim)
implicit real*8 (a-h,o-z)
character*8 A
```



```
c extend A to add on one RHS vector for elimination steps
if(n.le.ndim.and.inode.eq.1)
&  call allocate(A,ndim,'D')
call pcopy(B,1,1,A,n*ndim+1,1,n)
call extracta(A)
do 5 i=1,n
  call brall('ON')
  ioff=index2d(ndim,i,i)
  koff=index2d(ndim,1,i)
  j=pmax(n,A,koff,1)
  joff=index2d(ndim,j,i)
  call pswap(n-i+1,A,ioff,ndim,A,joff,ndim)
  call fetcha(inode,1,t,1,1,A,ioff,1,'D')
c this is the pivot, scale it to 1.0
  s=1.0/t
  call pscal(n-i+1,A,ioff,ndim,s,'D')
  do 10 j=1,n
    if(j.eq.i)goto 10
    joff=index2d(ndim,j,i)
    call fetcha(inode,1,s,1,1,A,joff,1,'D')
    v=-s
    call paxpy(n-i+1,v,A,ioff,ndim,A,joff,ndim,'D')
10 continue
5 continue
copy result vector back to B
call pcopy(A,n*ndim+1,1,B,1,1,n)
c no re-ordering has been done!
end
```

Again this will work for any distribution of data, but is only efficient for columns as discussed above.

It can be seen that PARLANCE is in fact very similar in concept to the proposed HPF language standard. It may however encompass extensions to it and is thus complementary, being simply a high-level library interface. PARLANCE uses message passing in its underlying routines, although it could equally well use shared memory without changing the user interface. An environment of this kind can therefore bridge the gap between: object-oriented databases and numerical algorithms; distributed and shared memory; coarse and fine grained systems; MIMD and SIMD computing. It embodies our experience of parallel computing so far!

Chapter 14 Bibliography and Further Reading.

R.J.Allan, L.Heck, and S.Zurek "Parallel FORTRAN in scientific computing: a new occam harness called FORTNET" Computer Physics Comms. 59 (1990) 325-44

R.J.Allan, and L.Heck "Parallel FORTRAN in scientific computing: a new occam harness called FORTNET" in 'Applications of Transputers 1' Proceedings of the 1st International Conference on Applications of Transputers, Liverpool 23-25 August 1989 ed. Len Freeman and Chris Phillips (IOS Press: 1990) ISBN 90 5199 025 1

R.J.Allan "FORTRAN-77 Programming of Parallel Computers" (1989) Daresbury Laboratory DL/SA/TM61T

R.J.Allan, E.L.Heck, R.K.Cooper, R.J.Blake and D.R.Emerson "FORTNET: design and experience of a portable message-passing harness for high performance transputer-based systems" Journal of the Transputer Consortium vol. 1 (1993) submitted

R.J.Allan "FORTNET v4.0 - The Parallel Programming Software" Daresbury Laboratory, revision 1 (June 1992)

R.J.Allan "PARLANCE v2.0 – Numerical Algorithms in a Virtual Shared Data Space" Daresbury Laboratory, revision 0 (June 1992)

R.J.Allan "Portable Message-passing Tools" DL Technical Memorandum DL/SCI/TM71E

R.J.Allan "A Tutorial in Parallel Programming - Parallel Algorithm Design and Implementation" 5-hour course presented at the Physics Computing '92 Conference, Prague (24/8/92) Daresbury Laboratory DL/SCI/P824T

R.J.Allan, M.F.Guest and P.J.Durham "Networked Computer Power" Physics World 4 (February 1991) 51-54

R.J.Allan, P.J.Durham and M.F.Guest "Networked Computer Power" Physics World 4 (1991) 51-4

G.Amdahl "Validity of the single-processor approach to achieving large-scale computer capabilities" AFIPS Conference Proc. vol. 30 (1967)

E.Anderson, Z.Bai, C.Bischof, J.Demmel, J.Dongarra, J.DDuCroz, A.Greenbaum, S.Hammarling, A.McKenney and D.C.Sorenson "LAPACK Working Note 20: A Portable Linear Algebra Package for High-performance Computers" University of Tennessee CS-90–105 (May 1990)

P.K.Andleigh and M.R.Gretzinger "Distributed Object-oriented Data Systems Design" (Prentice Hall: 1992) ISBN 0 13 174913 7

M.J.Aslett "A Knowledge based Approach to Software Development" (North Holland, 1991)

V.Bala and S.Knipis "Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library" Technical report, IBM Watson Research Center (October, 1992)

V.Bala, S.Knipis, L.Rudolph and Marc Snir "Designing efficient, scalable, and portable collective communication libraries" Technical report, IBM Watson Research Center (October, 1992)

H.-J.Bast, M.Gerndt and C.-A.Thole "SUPERB - the Suprenum Paralleliser" Supercomputer 30 (1989) 51-7

A.Beuglin "Sched3 notes for users" AMOCO Oil Company (1987)

J.Bernsen "Communication efficient matrix multiplication on hypercubes" Parallel Computing 12 (1989) 335-42

R.J.Blake, D.R.Emerson and R.J.Allan "FLOW: a Parallel Benchmark Code for High-speed Air Flow: version 1" Daresbury Laboratory. Documentation produced under contract to NPL (1992)

L.Bomans and R.Hempel "The Argonne/GMD macros in Fortran for portable parallel programming and their implementation on the Intel iPSC/2" Arbeitspapiere der GMD 406 (Gesellschaft fur Mathematik und Datenverarbeitung; Sankt Augustin) 1989, and Parallel Computing 15 (1990) 119-32

J.Boyle, R.Butler, T.Disz, B.Glickfeld, E.Lusk, R.Overbeek, J.Patterson, R.Stevens "Portable programs for parallel processors" (Holt, Reinhart and Winston, 1987) ISBN 0-03-014404-3

N.Carriero and D.Gelernter "How to write parallel programs, a guide to the confused" Research Report YALEU/DCS/RR-628 (May 1988)

N.P.Clancy "Integrated software system ..." M.Sc. Thesis, Bristol University (1990)

R.K.Cooper and R.J.Allan "FORTNET 3L (v1.0): A message- passing harness for transputers using 3L Parallel FORTRAN" for Computer Physics Communications (1991) submitted

Distributed Software Limited "PIPPA the Philips Parallel Processing Assistant" DSL (1992)

J.J.Dongarra and D.C.Sorensen "Schedule: Tools for developing and analysing parallel Fortran programs" Argonne National Laboratory, Technical Memorandum number 86 (1986)

J.J.Dongarra and D.C.Sorensen "Schedule User's Guide" Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Ill 60439

J.J.Dongarra and D.C.Sorensen "A Portable Environment for developing Parallel FORTRAN Programs" Parallel Computing 5 (1987) 175-86

J.J.Dongarra, J.J.Du Cruz, I.Duff and S.J.Hammarling "A set of level 3 basic linear algebra subproblems" ACM Trans. Math. Soft. (December 1989)

J.J.Dongarra, J.J.Du Cruz, S.J.Hammarling and R.Hanson "An extended set of FORTRAN basic linear algebra subproblems" ACM Trans. Math. Soft. 14 (1988) 1-32

J.J.Dongarra et al. "Solving Linear Systems on Shared Memory Computers" Pub. SIAM (1991) ISBN 0-89871-270-X

J.J.Dongarra, J.R.Bunch, C.B.Moler and G.W.Stewart "LINPACK User's Guide" (SIAM Press, 1979)

P.Dzwig et al. "The Ei3 Report on the Future of High Performance Computing in Europe" (Ei3: July 1992)

D.R.Emerson, R.J.Blake and R.J.Allan "Implementation of a Navier-Stokes Solver using Fortnet: Implementations and Results" "Parallel CFD '91" (Elsevier Science Publishers bv: 1992)

D.R.Emerson, R.J.Blake and R.J.Allan "CFD Experiences on a Range of Novel Architecture Systems: Implementations and Results" "Parallel CFD '92" (Elsevier Science Publishers bv: 1992) submitted

D.R.Emerson and D.I.A.Poll "High-speed Laminar Flow over Cavities" in Applications of Supercomputers in Engineering II. eds. D.ABrebbia, D.Howard and A.Peters (Elsevier, Applied Science Series 1991) ISBN 1–85166–695–8

EPCC "CHIMP concepts" Edinburgh Parallel Computing Centre (June, 1991)

EPCC "CHIMP version 1.0 Interface" Edinburgh Parallel Computing Centre (May, 1992)

S. Even, B. Monien "On the number of rounds necessary to disseminate information" Proceedings ACM Symposium on Parallel Algorithms and Architectures, ACM 1989, S. 318-327

J.Fairclough et al. "Review of Information Science and Engineering in SERC" (SERC, Swindon: April 1992)

R. Feldmann, B. Monien, P. Mysliwietz "A fully distributed chess program" in 'Advances in Computer Chess 6' ed. D. Bial (1990) 1-27

R. Feldmann, P. Mysliwietz, B. Monien, O. Vornberger, "Distributed Game Tree Search" in 'Parallel Algorithms for Machine Intelligence and Vision' ed. V. Kumar, P.S. Gopalakrishnan, L.N. Kanal (Springer Verlag: 1990) 66-101

M.J.Flynn "Some Computer Organisations and their Effectiveness" IEEE Trans. C-21 number 9 (sept. 1972) 948

G.Fox, M.Johnson, G.Lycenza, S.Otto, J.Salomon and D.Walker "Solving problems on concurrent processors" vol 1

G.Fox - loose synchronicity

Garner, J., Allan, R.J., Blake, R.J. and Emerson, D.R. (1993) "Dynamic Load Balancing in heterogeneous workstations clusters" for submission to Journal of Scientific Computing

D.Gannon, D.Atapattu, M.H.Lee and B.Shei "A software tool for building supercomputer applications" presented at the Butterfly Fall 1987 user group meeting, October 26-28, 1987, Cambridge Mass. USA

P.W.Gaffney and E.N.Houstis "Programming Environments for high-level Scientific Problem Solving" IFIP Transactions A-2 (North Holland: 1992) ISSN 0 444 89176 5

J.Garner "Development of a two-dimensional load balancing strategy and its implementation" M.Sc. Thesis, University of the West of England (1992)

J.Garner, R.J.Allan, R.J.Blake and D.R.Emerson "Dynamic load balancing in multi-dimensional domain decomposition: appication to heterogeneous workstation clusters" Journal of Scientific Computing (1993) in preparation

G.A.Geist, M.T.Heath, B.W.Peyton, P.H.Worley "PICL a Portable Instrumented Communication Library" Oak Ridge National Laboratory Technical Memorandum ORNL/TM-11130

G.A.Geist and V.S.Sunderam "Network-based concurrent computing on the PVM system" Concurrency 4 (1992) 293-311

D.Gelernter "Getting the job done" Byte (Nov. 1988) 301-7

J.L.Gustafson "Re-evaluating Amdahl's Law" Comm. ACM 31 number 5 (1988) 532-3

W.Handler "The Impact of Classification Schemes on Computer Architecture" Proc. Int. Conf. on Parallel Processing (1977) 7-15

R.J.Harrison "Documentation of tcgmsg toolset" Argonne National Laboratory and private communication (1990, 1991)

Helios "The CDL Guide" Distributed Software Ltd. (1990) part number H09014

R.Hempel "The ANL/GMD Macros (PARMACS) in Fortran for Portable Parallel Programming using the Message Passing Programming Model: User's Guide and Reference Manual, v5.1" GMD mbH, Sankt Augustin, W.Germany (November 1991)

C.Herzfeld et al. "Grand Challenges: High Performance Computing and Communications: The FY 1992 U.S. Research and Development Program" (National Science Foundation, Washington D.C., USA: 1992)

C.A.R.Hoare "Communicating Sequential Processes" (1978) Comm. ACM 666-77

C.A.R.Hoare "Communicating Sequential Processes" (Prentice Hall: 1985)

R.W.Hockney and C.R.Jesshope "Parallel Computers 2" (Adam Hilger: 1988) ISBN 0–85274–812–4

J. Hromkovic, B. Monien "The Bisection Problem for Graphs of Degree 4" Proceedings 'Mathematical Foundations of Computer Science' (1991)

Y.-F. Hu and R.J.Blake "Numerical Experiments with Partitioning of Unstructured Meshes" Parallel Computing (1993) submitted for publication

Y.-F. Hu and R.J.Blake "Algorithms for Scheduling of Message Passing" Parallel Computing (1993) submitted for publication

K.Hwang and F.A.Briggs "Computer Architecture and Parallel Processing" (McGraw Hill, 1984) ISBN 0-07-031556-6

Inmos "The T9000 Transputer Hardware Reference Manual" Inmos Ltd. document 72 TRN 23800 (August, 1992)

A.Kolawa "Express User's guide and reference manual" ParaSoft Inc.

C.Lawson, R.Hanson, D.Kincaid and F.Krogh "Basic linear algebra subprograms for FORTRAN usage" ACM Trans. Math. Soft. 5 (1979) 308-25

S.J.Leffler, R.S.Fabry and W.N.Joy (19??) "A 4.2BSD Interprocess Communication Primer" Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University opf Berkley, California

Wm. Leler "Linda meets UNIX" IEEE Computer Magazine 23 (Feb. 1990) 43-54

M.Lenzenin, D.Nardi and M.Simi "Inheritance Hierarchies in Representation and Programming Languages" (Wiley: 1991) ISBN 0 471 92741 4

R.Luling and B.Monien "Load balancing for distributed Branch & Bound Algorithms" in Proc. Int. Parallel Processing Symposium (1992)

R.Luling, B.Monien and F.Ramme "Load balancing in large networks: a comparative study" Proc. 3rd IEEE Syposium in parallel and distributed computing, Dallas (1991) 686-89

R.Luling, B.Monien, M.Racke and S.Tschoke "Effective parallelisation of a Brahcn & Bound algorithm for the symmetric travelling salesman problem" European Workshop on parallel computing (EWPC) Barcelona (1992)

R.Luling and B.Monien "Dynamic distributed load balancing with proven good performance" ACM Symposium on parallel algorithms and architectures, Velen, Germany (1993)

E.Maim et al. "Restricted Abduction in Constraint Logic Programs" Proc. SEKE'93 (International Conference on Software Engineering and Knowledge Engineering), San Francisco, USA, (1993) to appear

E.Maim et al. "Abduction and Constraint Logic Programming" Proc. 10th ECAI (European Conference on Artificial Intelligence", Vienna, Austria (1992)

E.Maim et al. "Dealing with Time Granularity in the Event Calculus" Proc. FGCS'92 (International Conferenc eon Fifth Generation Systems), Tokyo, Japan (1992)

E.Maim et al. "Recognising Objects from Constraints" Proc. SEKE'92, Capri, Italy (1992)

E.Maim et al. "Dealing with Granularities using Uniform Event Calculus", in 'Artificial Intelligence, Expert Systems and Symbolic Computing' ed. E.Houstis (North Holland: 1992)

J.Martin "Principles of object-oriented analysis and design" (Prentice Hall: 1993) ISBN 0–13–720871–5

T.Merrow and N.Henson "System design for parallel computing" High Performance Systems (Jan. 1989) 36-44

B. Monien, I.H. Sudborough "Min Cut is NP-complete for Edge Weighted Trees" Theoretical Computer Science 58 (1988) 209-229

B. Monien, I.H. Sudborough "Simulating binary trees on hypercubes" in 'Proceedings 3rd Aegean Workshop on Computing' LNCS 319 (1988) 170-180

B. Monien, I.H. Sudborough "Embedding one Interconnection Network in Another Computing Suppl. 7 (1990), 257-282

B. Monien "Simulating binary trees on X-trees", Proceedings ACM Symposium on Parallel Algorithms and Architectures, ACM (1991) 147-158

NCube "NCube 2 programmers guide r2.0" NCube Corporation (December, 1990)

C.Owens "Static and Dynamic Load Balancing" M.Sc. Thesis, Bristol Polytechnic (1991)

Parsytec "GC documentation" Parsytec GmbH, Aachen

P.Pierce "The NX/2 Operating System" in 'Proc. 3rd Conference on Hypercube and COncurrent Computers and Applications', pp384–90 (ACM Press, 1988)

C.V.Ramamoorthy and H.F.Li "Pipeline Architecture" ACM Computing Surveys 9 number 1 (March 1977) 61-102

G.Richelli "PVMe User Guide, Aix 3.2 version" ECSEC Rome (1992)

C.Rubbia et al. "Report of the High Performance Computing and Networking Advisory Committee" Vol. 1 (CEC: October 1992)

P.Sherwood "DISPLAY User Manual version 6.1" Daresbury Laboratory

J.E.Shore "Second thoughts on parallel processing" Comput. Elect. Eng. 1 (1973) 95-109

B.Smith, J.Boyle, J.J.Dongarra, B.Garbow, Y.Ikebe, V.Klema and C.B.Moler "Matrix Eigenvalue Routines — EISPACK Guide" (Springer Verlag, 1976)

A.Skjellum and A.Leung "Zipcode: a portable multicomputer communcation library atop the reactive kernel" in 'Proc. 5th Distributed Memory Concurrent Computing Conference' eds. D.Walker and Q.Stout, pp767–76 (IEEE Press, 1990)

A.Skjellum, S.Smith, C.Still, A.Leung and M.Morai "The Zipcode message passing stsrem" Technical Report, Lawrence Livermore National Laboratory, USA (September 1992)

V.Sunderam "PVM: a framework for Parallel Distributed Computing" Concurrency 2 (1990) 325–39

D.J.Wallace, R.D.Kenway et al. "Proposal for a European Teraflops Initiative" (Steering and Technical Options Commitees of the ETI: September, 1991)

Ware's Law

K.H.Werner, U.Brass and E.Thomas "The Suprenum User Interface" Supercomputer 4(March 1989) 20-24

R.D.Williams "Performance of dynamic load balancing algorithms for unstructured mesh calculations" Concurrency 3 (1991) 457–81

F.W.Wray "High-preformance Numerically Intensive Applications on Distributed Memory Parallel Computers" in 'Scientific Computing on Supercomputers III' ed. J.Z.Devreese and P.E.van Camp (Plenum Press: 1991)