THE INFLUENCE OF COMPUTER ARCHITECTURES ON ALGORITHMS

David Dent

ECMWF

Reading, UK

Summary: Various aspects of computer architectures are considered in relation to meteorological applications.

1. INTRODUCTION

Since the title of this paper covers a very large subject, discussion will be limited to

(1) operational environments which execute time critical codes
(2) running on 'supercomputers'
(3) Technical problems as well as the algorithms themselves will be considered; this includes coding style and portability aspects.

Figure 1 shows a schematic relationship between the scientific algorithm as the top layer (equations, finite difference techniques, etc.) and the implementation on to a computer underneath, usually in Fortran. By means of vendor-supplied compilers, libraries etc., this layer provides separation from the specific hardware. Naturally, we hope for complete portability at this interface, but it is worth observing that portability without reasonable performance is useless.

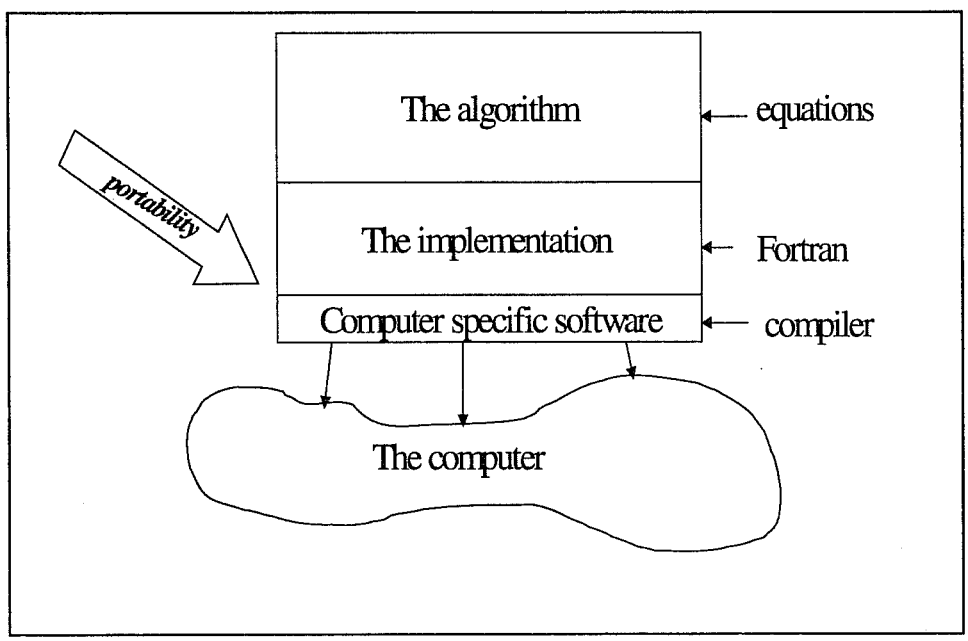

Figure 1: Relationship between algorithm and computer hardware

The ways in which computer architectural features influence algorithms will be considered by looking separately at:

- processor design (vector/scalar)
- memory access (vector/caches)
- memory size
- multiple processors
- memory organisation (shared/distributed)
- I/O

2. PROCESSOR DESIGN

2.1 Vector architecture

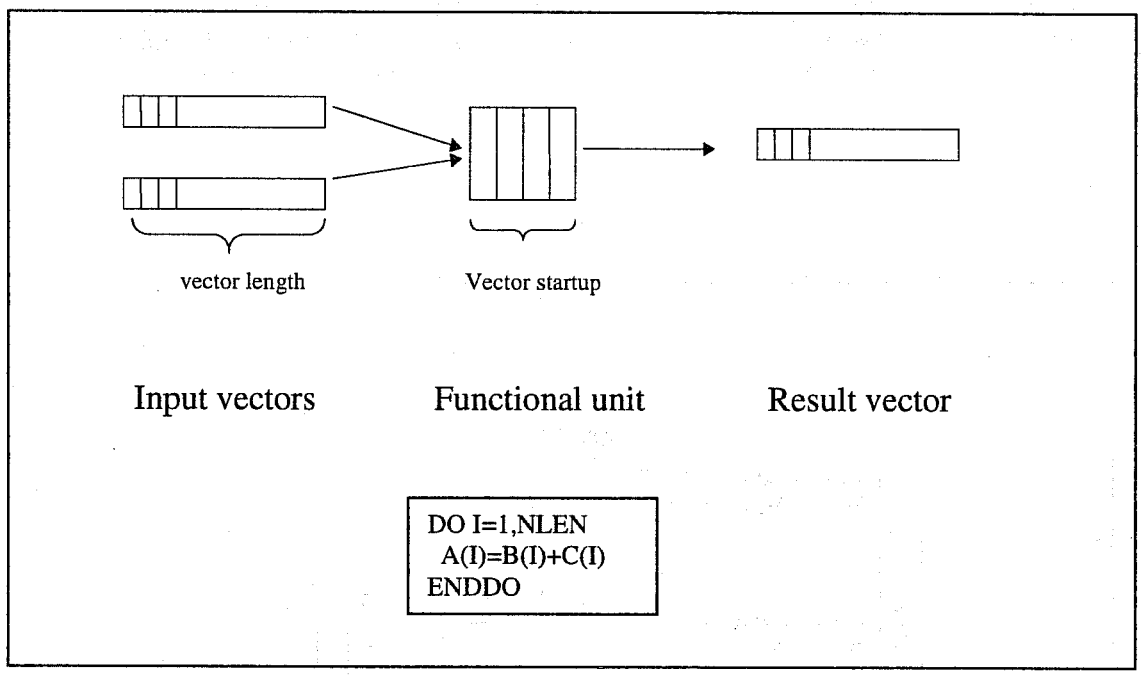

Figure 2: Vector architecture

The fundamental technique utilised in vector processors is 'pipelining'. The logic which performs a specific operation (e.g. a floating-point addition) is split into several independent segments, each of which can be completed in one clock period. Thus, a pair of operands can proceed to the second segment while a new pair of operands commences on segment one. In addition, the functional units usually contain several pipes so that more than one partial computation is performed at each step. The pipelining of functional units allows the result rate to be fixed (usually 2,4 or 8 results/clock). The vector start-up cost is the time taken before the first result becomes available and depends on the number of segments in the pipe.

One (vector) instruction produces a vector of results. The application code, the vector registers available, and the compiler determine the length of the vector operation.

A vectorizing compiler carries out the process of generating vector instructions. Vector operations are typically generated to execute the contents of a Fortran DO loop. They are possible only where there are no dependencies between DO loop iterations. The necessary dependency analysis is performed by the compiler and can usually be supplemented by user directives.

Many processors allow floating point add and multiply operations to proceed in parallel. This implies either the existence of two independent functional units, or one 'combined' unit capable of computing linked add/multiply operations. Computations such as this usually define the theoretical floating point computational rate for the processor (e.g. a vector-matrix product of infinite size).

Memory access is also pipelined, allowing operands to be transferred into and out of vector registers at the same rate as the computational pipes. This often means that the load/store pipes are several words wide, i.e. more than one word is transferred to/from memory simultaneously.

Note that vector processors also contain scalar processing capability in order that non-vectorizable parts of an application can be computed. The contrast in execution speed between vector and scalar code is often quite high (greater than 20) so that the tuning of applications to increase the vector component is an important optimising phase.

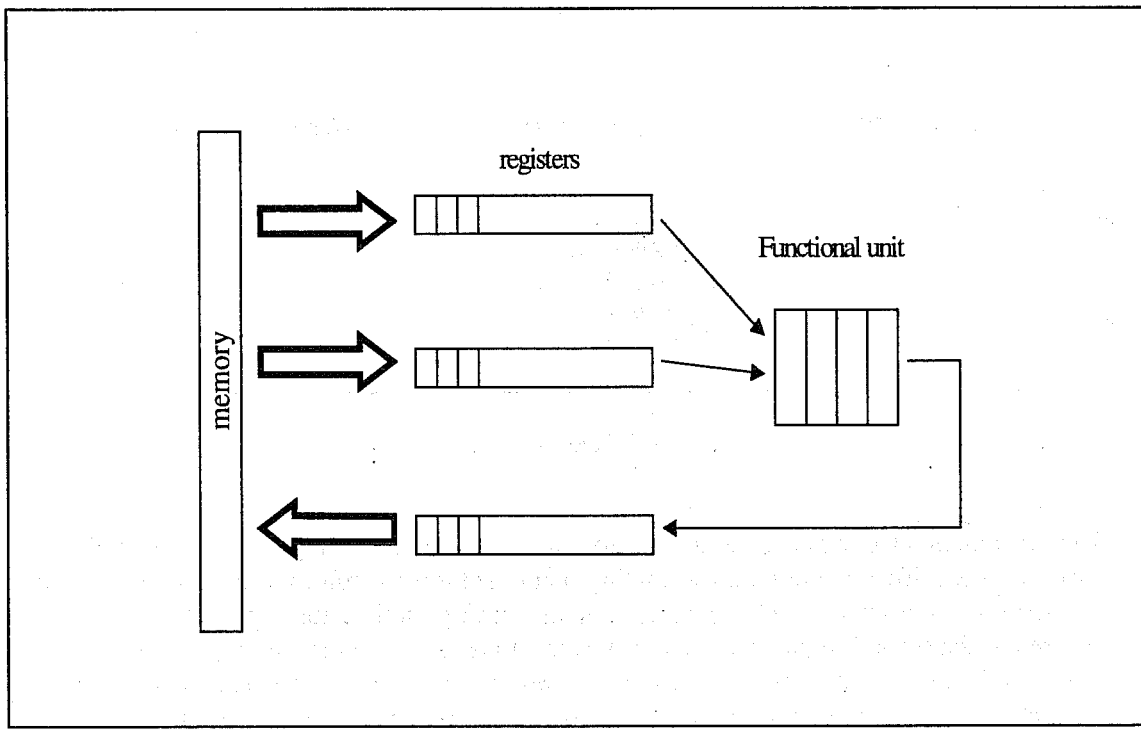

Figure 3: vector memory access

2.2 Scalar architecture

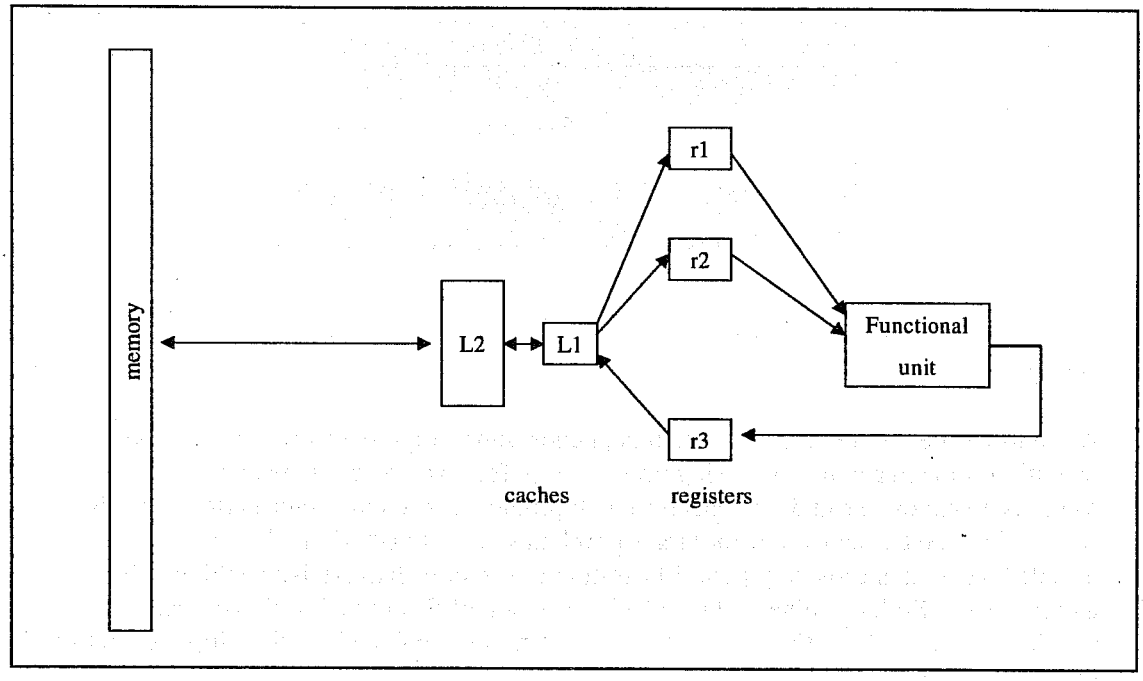

Figure 4: scalar architecture

In contrast to vector architectures, a scalar processor computes at most only one result per instruction. There is therefore no need for 'vectorizable code', but achieved execution rates are normally lower.
Memory access is assisted by provision of caches which are fast but small areas of memory. There may be up to 3 levels, all much closer to the processor than main memory in terms of access time.

Cache lines (typically 64 or 128 bytes) define the unit of loading/storing from memory. The management of cache is (almost) totally controlled by hardware and is known as the cache replacement strategy. Since there is such a performance difference between accessing main memory and accessing cached data, there is a major optimising opportunity to try to reuse data while it is in cache.

2.3 Application performance on scalar and vector architectures

Both vector and scalar architectures are in widespread use at production weather sites:

- FUJITSU VPP700 (vector) - ECMWF and Meteo-France
- CRAY T3E (scalar) - UKMO, DWD, SMHI, DNMI, FMI
- NEC SX/4 (vector) - DMI, LACE, AES
- SGI Origin (scalar) - KNMI, IMS
- CRAY C90 (vector) - INM

Typical peak performances of scalar machines are well below those of vector machines. Achievable computational rates are also substantially less.

	scalar	*vector*
Peak Mflops	400 - 1200	1000 - 2000
Achievable fraction	5 – 15 %	30 - 50 %
Achievable Mflops	20 - 180	300 - 1000

However, the vector rates depend on suitably vectorized codes. Otherwise performances are very poor.

A shallow water kernel demonstrates this phenomenon. Figure 5 shows floating point operations plotted against increasing problem size for a vector machine and a scalar machine. Vector start-up overhead dominants for small problem sizes, but vector performance becomes substantially better than scalar for larger problem sizes. If vectorizing is deliberately disabled, performance is very poor, illustrating a common characteristic exhibited by vector architectures. Both machines achieve a high fraction of theoretical peak for a range of problem sizes, thanks to the excellent overlap of floating point add and multiply operations in this application.

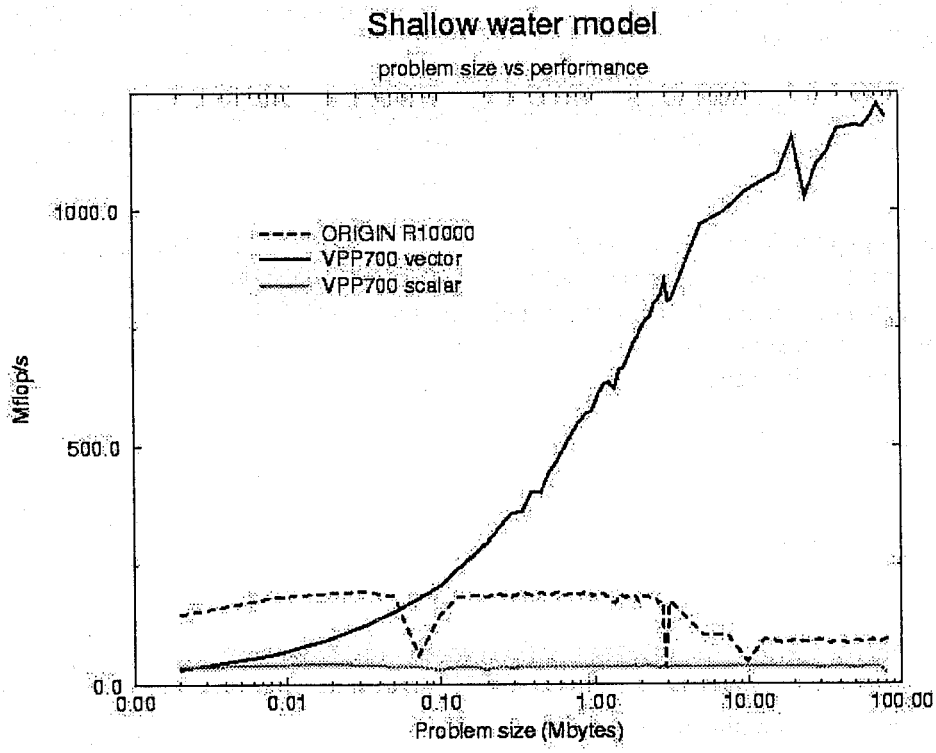

Figure 5: vector and scalar performances for shallow water kernel

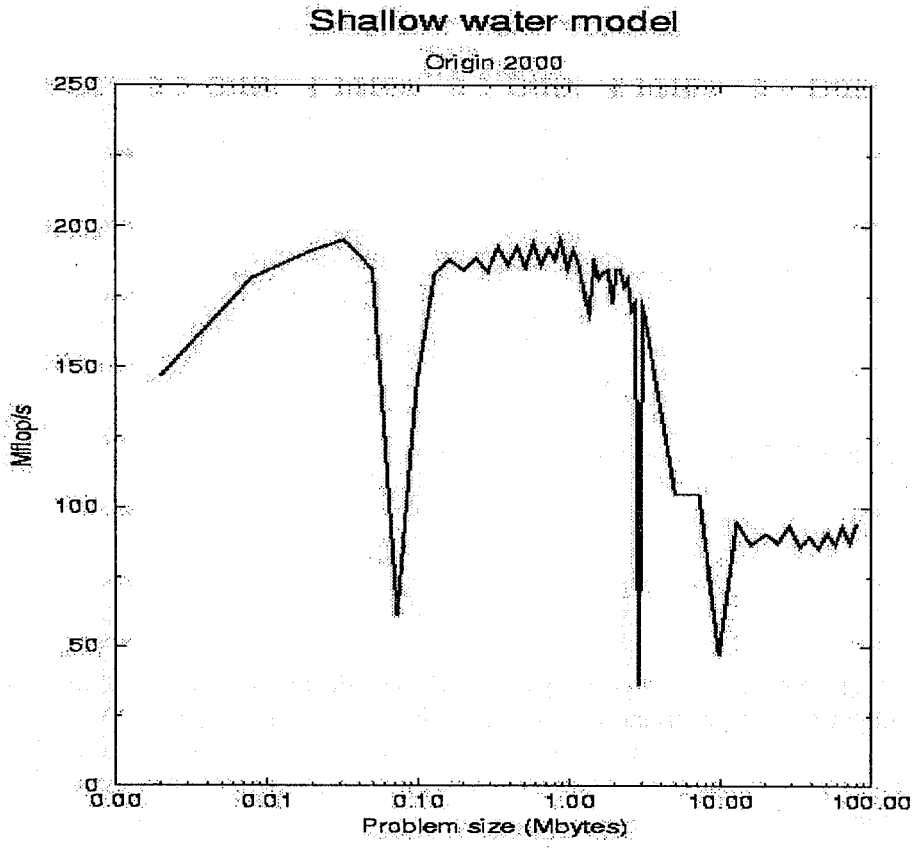

Figure 6: influence of cache size on scalar performance of shallow water kernel

Performance on a scalar processor (figure 6) becomes bad when the problem size increases, since the data no longer fits into cache and calculations therefore run at the speed of main memory access.

The poor performances at 0.8 MB and 3.2 MB are explained by looking at the way in which data is loaded into cache from memory. Several different Fortran array addresses can map to the same cache line and thus overwrite each other. Figure 7 illustrates this and indicates that it is also possible in some processors to bypass the cache completely by generating data transfers directly between processor and memory. This is a compiler optimising technique whereby data can be preloaded into cache before it is needed for computation.

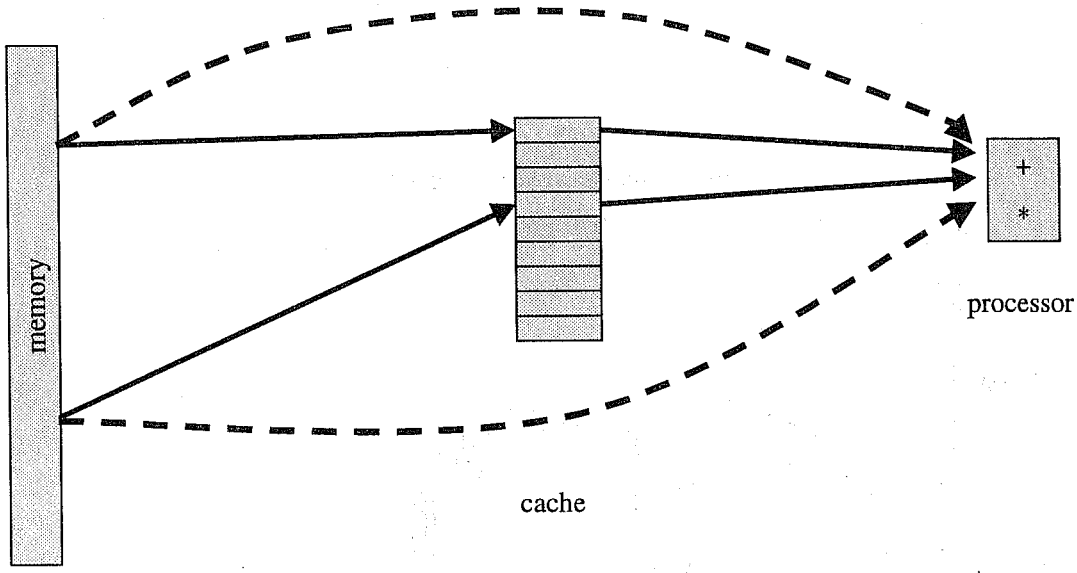

Figure 7: Data transfers between memory and processor, via cache

When accessing elements of Fortran arrays in a DO loop, the order of striding through memory is important for performance. Stride 1 is best, as in column oriented references to Fortran arrays (see figure 8). Row references require non-unity striding which means that both vector memory references and cache line memory references are inefficient, since only a subset of the loaded data is used (one word out of a cache line or one word out of a vector memory pipe reference).

Indirect addressing is even worse, because:

(a) the index list has to be loaded first
(b) Memory references may be randomly distributed, leading to (typically) one eighth of memory bandwidth.

This is well illustrated by the performance of sparse matrix computations where significant processing speed is lost simply because of the indirection. The pattern of indirect addressing can generate even more overhead. However, sparse matrix operations are still an economic choice to make. One should simply not have very high performance expectations.

```
REAL A(8,4)

DO J2 = 1 , 4
 DO J1 = 1 , 8
  .......A( J1 , J2 )
 ENDDO
ENDDO
```

```
REAL A(8,4)

DO J1 = 1 , 8
 DO J2 = 1 , 4
  .......A( J1 , J2 )
 ENDDO
ENDDO
```

1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31
8	15	24	32

```
REAL A(8,4)
INTEGER LIST(8)
DO J2 = 1 , 4
 DO J1 = 1 , 8
  .......A ( LIST( J1 ) , J2 )
 ENDDO
ENDDO
```

Sparse MATRIX x VECTOR (2048 x 64)

percent of peak VPP700 speed

Dense	81%
stride 1	23%
stride 2	17%
random	11%

Figure 8: row and column Fortran array access to memory, and the effect of indirect addressing

Even with highly vectorized routines, considerable performance differences are seen, as illustrated by the ten most expensive routines in IFS running on VPP700 (figure 9). Concurrency of ADD and MULTIPLY is most important (matrix product is the best case). The Memory Access Index (MAI) is high when use of memory is light (high computational intensity) or is stride 1. Conversely, the MAI is low for heavy memory use and/or strided or indirect memory access.

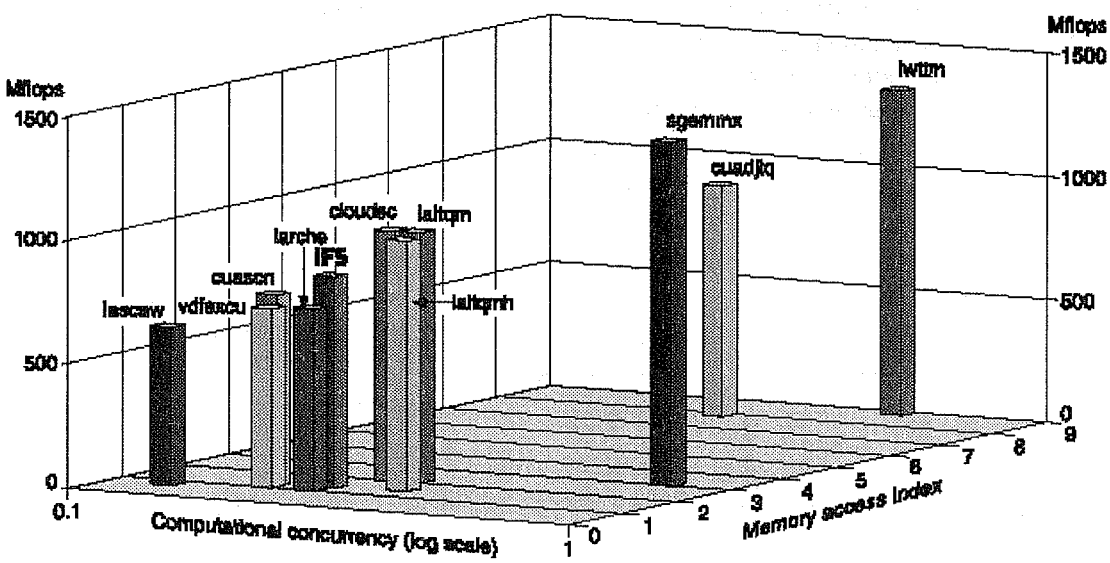

Figure 9: performance of some IFS routines as a function of computational concurrency and memory access patterns

Summary profiles of IFS (figure 10) show increase of Legendre Transform (LT) cost with increasing resolution due to the cubic scaling of this part of the spectral algorithm. This is one of the often-quoted disadvantages of spectral models.

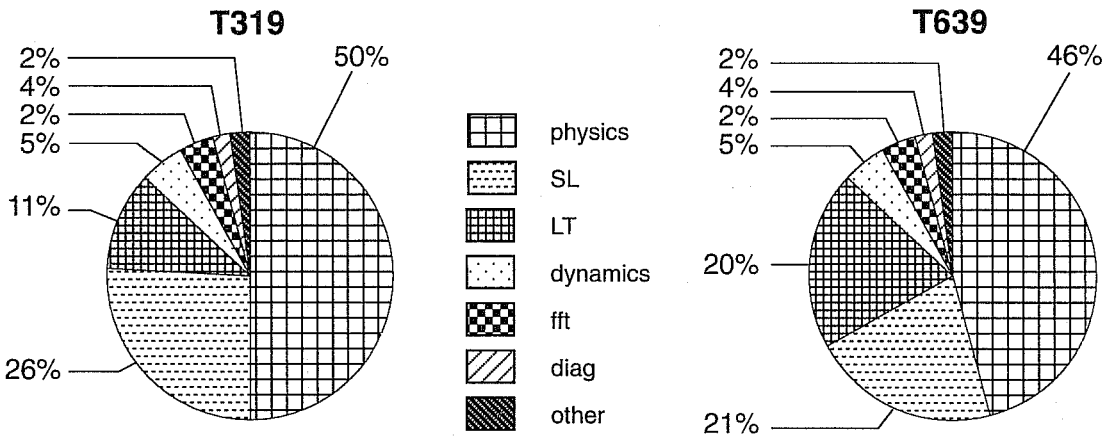

Figure 10: IFS profiles showing relative cost of Legendre transforms for two spectral resolutions

3. MEMORY ORGANIZATION

3.1 Memory size

High performance memory (CRAY C90, NEC SX/4 SSRAM) is expensive, and therefore limited in quantity. This forces models to use I/O schemes, in order not to be constrained by memory storage space. Secondary memory (SSD) on these machines is much larger and slower, and is usually configured to act as a repository for files. (Figure 11). SSD has become much less common now, as most vendors have moved to larger, slower main memory, thus eliminating I/O schemes. Provision of more than one processor leads to the next big architectural influence on algorithms.

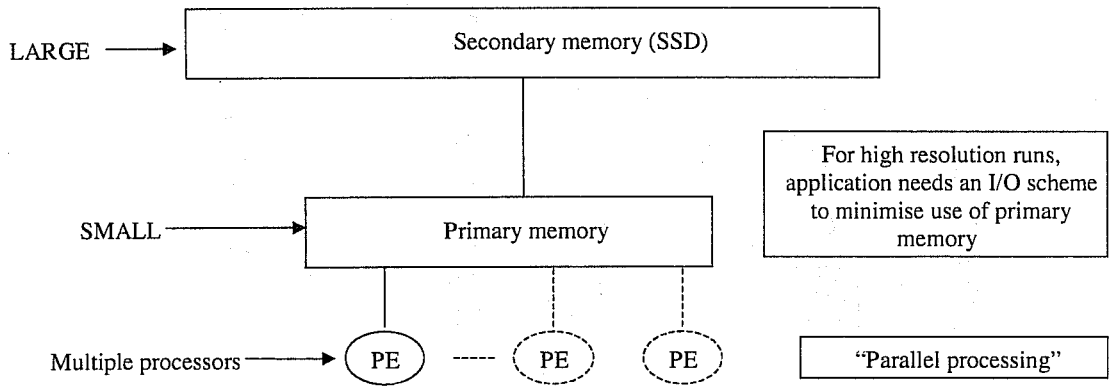

Figure 11: primary and secondary memories

3.2 Multiple Processors

The simplest architectural development from the application point of view is for processors to be connected to the same memory (CRAY X-MP, Y-MP, C-90, J-90, T-90, and NEC SX2/3/4). However, this becomes harder and more expensive to do as more processors are added and the largest configuration in regular use today is 32 (NEC SX/4 and CRAY T-90). At the other end of the scale, it is becoming commoner to connect multiple microprocessors together to share memory, usually in groups of 2,4 or 8 (SGI Origin, IBM SP) and often referred to as Shared Memory Processors or SMPs.

The total storage space can now consists of many separate memories which are interconnected by means of some kind of network (figure 12). The time to access data on a processor's own memory is usually significantly less than the time to access data from another memory, leading to the term Non Uniform Memory Access or NUMA.

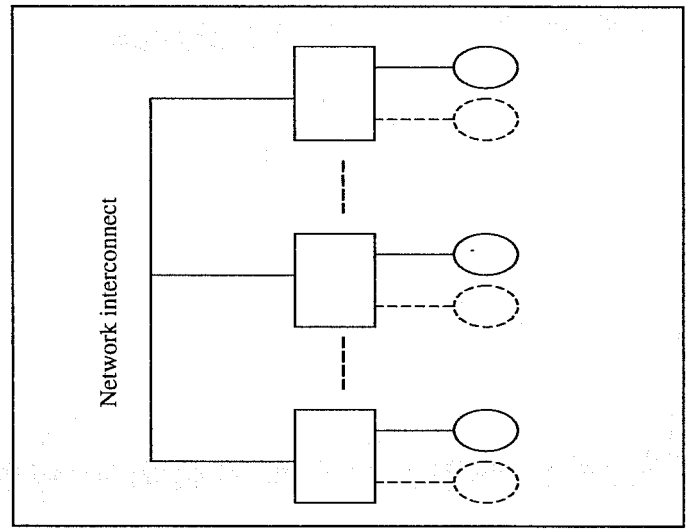

Figure 12: parallel processors with distributed memory

Programming parallel machines so that one application can usefully utilise multiple processors is a new challenge. Usually, the same code is run in each processor, although the hardware (Multiple Instruction Multiple Data or MIMD) does not dictate that. Useful parallelism is achieved by dividing the data in some way across the processors leading to a Single Program Multiple Data (SPMD) strategy. The programmer can describe this distribution in a number of ways. The simplest and therefore most attractive from the manpower point of view is a 'data parallel' extension to Fortran (HPF). So far, this paradigm has not been able to provide a general efficient mechanism except for very few highly suitable codes.

Invoking parallelism within an SMP has been possible for some time by means of a compiler capability (e.g. Cray Microtasking), aided by user provided compiler directives. These directives have now been standardised (OPENMP) and are becoming a useful way to parallelize within a node.

The only really successful general technique to become well established is 'message passing'. A standard has been defined (see MPI 1996) and, used in conjunction with F77 or F90, has proved to be a vehicle for creating portable parallel codes which attain reasonable parallel efficiency on a variety of platforms.

4. MESSAGE PASSING

The message passing programming model is straightforward:

- Data is viewed as being private to each processor.
- Data belonging to other processors must be transferred in a co-operative way, i.e. the 'owner' SENDS the data, and the requestor RECEIVES the data.

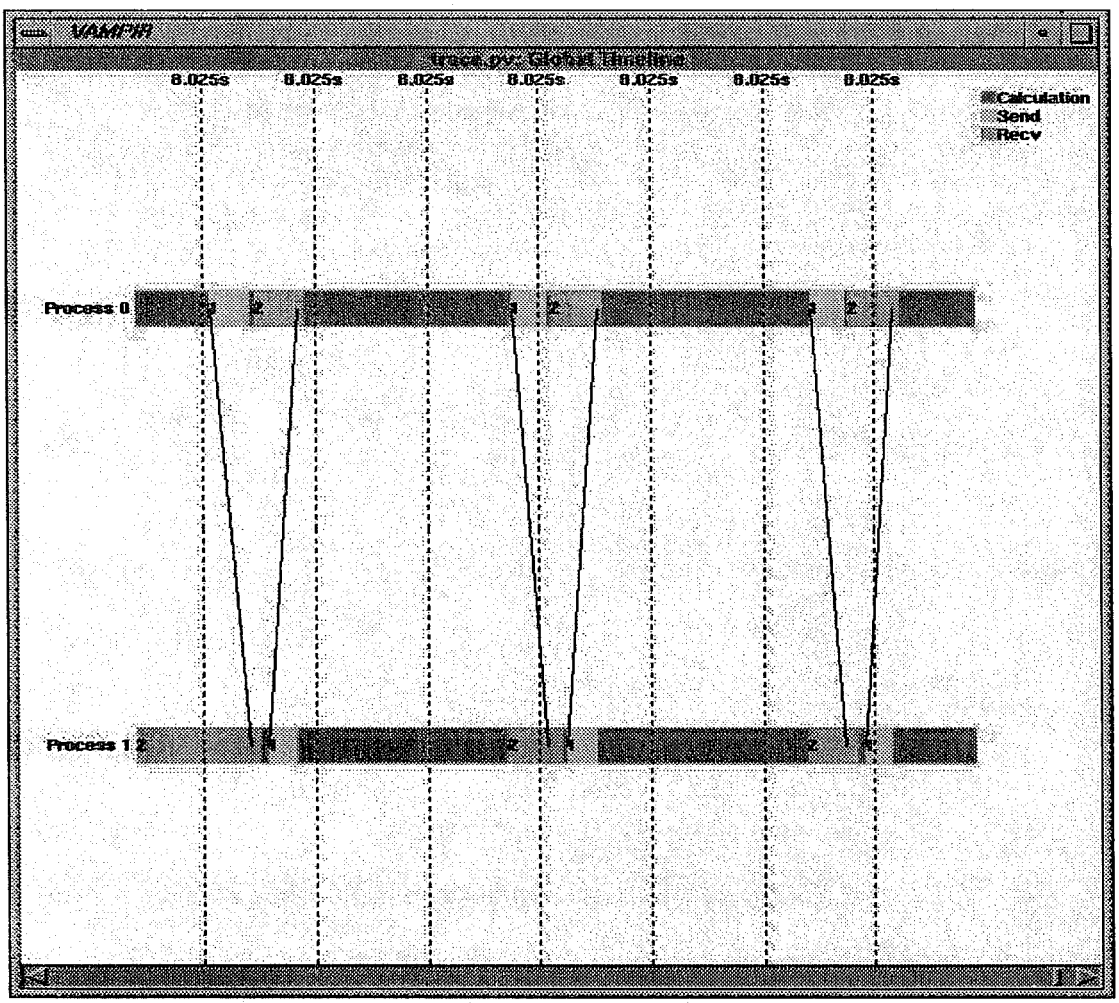

Figure 13: co-operative message passing showing SEND and RECV states

In figure 13, two processes (0 and 1) exchange messages. The horizontal axis is the execution time of the application. Lines connecting the processes indicate the transmission of messages.

Performance (i.e. time to transfer) depends on the hardware interconnect speed and also on the size of the message to be sent. There is a hardware and software 'start-up' time required to send an empty message which is typically of the order of 10 to 20 micro-seconds (see figure 14). This leads to an optimising technique whereby data to be transferred is grouped or copied into a buffer to create a longer message.

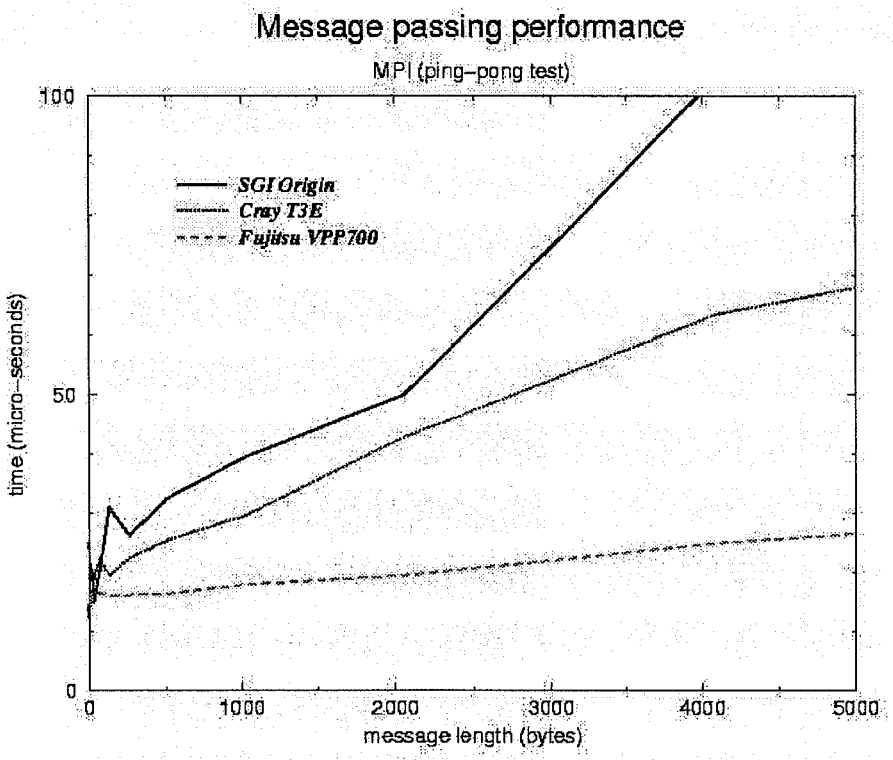

Figure 14: message passing costs as a function of message length for several parallel computers

There are facilities within MPI for the message to be held in a 'mailbox' in case the destination processor is not ready to receive. Note that any synchronisation between the participating processors is implicit. The sender can send without knowledge of the state of the receiver.

Most vendors provide their own (non-portable) message passing system which usually has a higher performance than MPI, since some of the software overheads are avoided. In the case of CRAY T3E, additional features are provided, such as the 'one-sided' transfers (SHMEM_GET, SHMEM_PUT). These allow a processor to transfer data into and out of the address space of another processor without its cooperation. Additional logic (such as a BARRIER) is needed to ensure that the required data is actually available at the time of the one-sided get. Similar functionality is being provided in a new extended definition of MPI known as MPI2 (see MPI2 1996).

Barriers generally apply across the full range of processors involved in the parallel execution. As each process arrives at the barrier, it is forced to wait until all participating processes have arrived. Although the cost of the barrier itself may be small, it is the potential for delays that leads to performance implications. Figure 15 shows wasted time in **black** as processors arrive at a barrier and are forced to wait for the slower processes to catch up before continuing with computation.

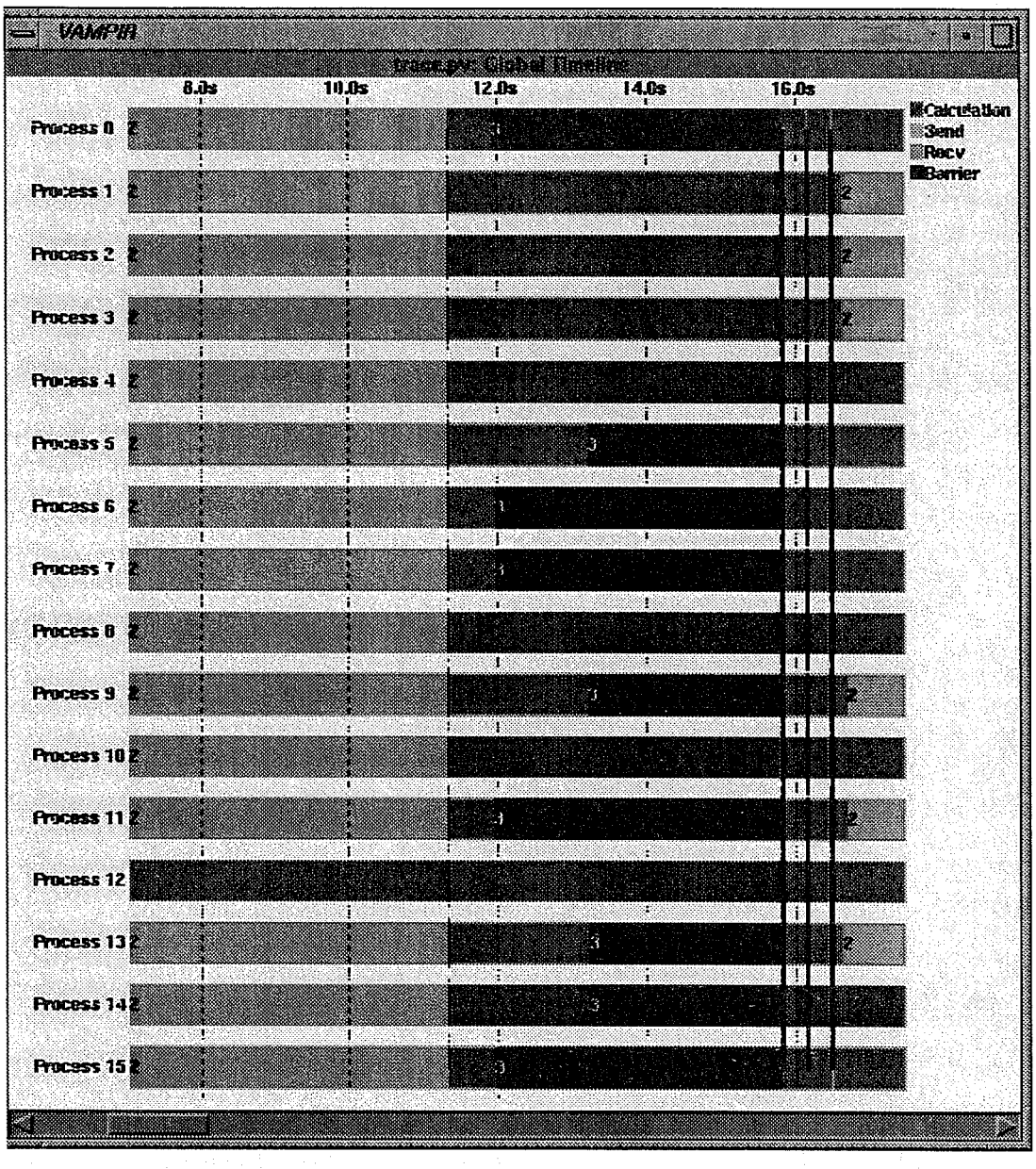

Figure 15: BARRIER inefficiency illustrating work imbalances

5. LOAD BALANCING

Since nearly all parallel applications use the SPMD model, the way in which the data is distributed across available processors is an important issue. A poor choice of data distribution can lead to a waste of computing resources as some processors finish their work ahead of others. Models that can run at a variety of resolutions and on a range of processor configurations may have to go to some lengths in order to guarantee the best possible static load balance for all cases. An example from IFS is shown in figure 16 where the distribution of grid points is such that, for an arbitrary resolution and machine partition, each processor is guaranteed to own an equal number of points (plus or minus 1).

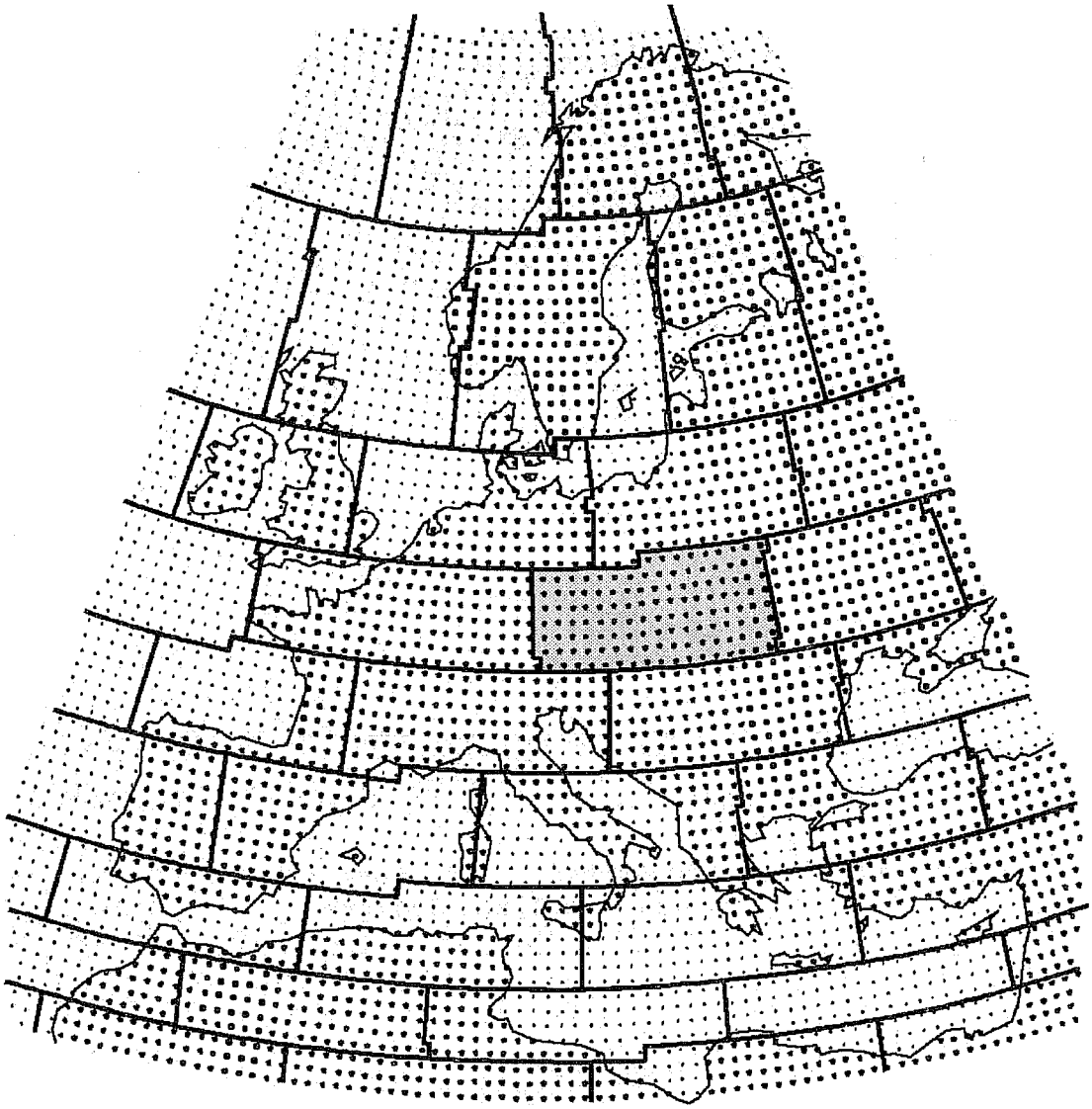

Figure 16: static load balancing in grid-point computations of IFS

Dynamic load imbalance may be generated when the computations take differing paths through the code depending on the input data. This commonly happens to a greater or lesser extent within the physical parametrization code of models. By its nature, it is often largely unpredictable and is much harder to deal with than static load imbalance. Instrumentation within IFS has been used to quantify its effect (see figure 17). The smaller the quantity of work assigned to a processor, the more likely it is to be a problem, so the magnitude of the imbalance depends on:

(1) model resolution
(2) number of processors being used
(3) the nature of the physics code

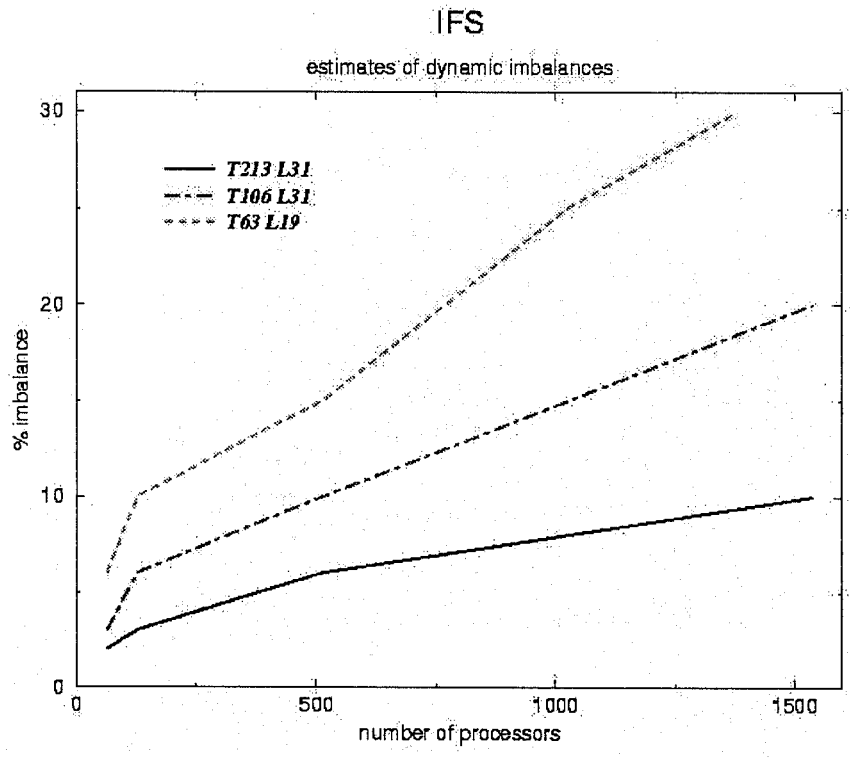

Figure 17: estimated dynamic load imbalances in IFS for several model resolutions

5.1 Dynamic Balancing Techniques

Shared memory architectures allow simple and effective dynamic load balancing schemes to be implemented. Since any processor can access any part of memory with equal efficiency, it is possible to 'hand out' units of work on demand (figure 18). The only control needed is the ability to perform an 'atomic update' on a shared variable, so that each requestor is given a unique item of work.

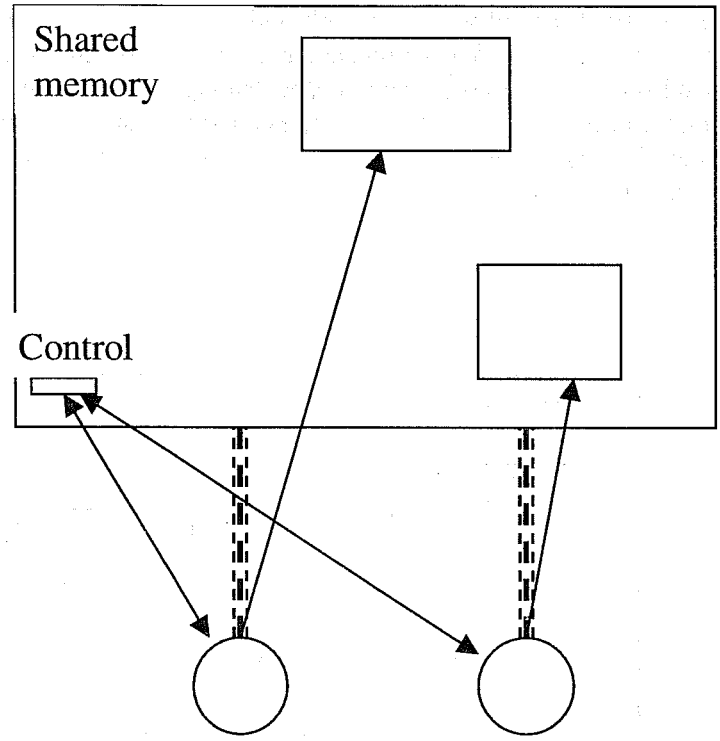

Figure 18: dynamic load balancing in a shared memory environment

In a distributed memory environment, the data associated with the work will usually reside on another processor and therefore must be moved. In order to keep within the two-sided philosophy of message passing, difficulties in the management of this dynamic mechanism exist. A relatively simple technique requires the creation of an 'administrator' code, which runs in a dedicated fashion in one processor. This code owns the work list and knows where the related data resides (see figure 19). The obvious disadvantage is the loss of one processor from the partition in use and is therefore likely to be an attractive solution only for models running on the more massively parallel platforms where hundreds of processors are typically involved in the computation. This load balancing technique has been applied successfully within the OI data assimilation codes of DWD and HIRLAM, running on T3E platforms.

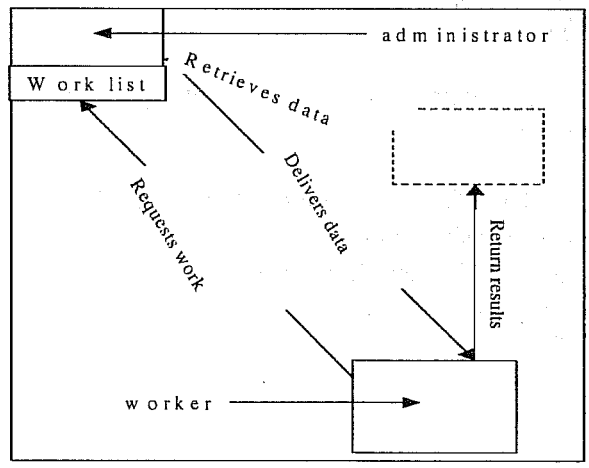

Figure 19: dynamic work redistribution using an 'administrator'

An alternative scheme in use in the convection scheme of the UKMO Unified Model depends on T3E SHMEM one-sided capabilities. The work statically assigned to each processor is subdivided into a number of units. When a processor requires a new unit of work, it will select from its own list until this is completed. It then 'snoops' through the work lists of other processors looking for unfinished work items, updates the work list pointer in an atomic fashion, retrieves the associated data, performs the work and then returns the computed results to the original 'owner' (figure 20).

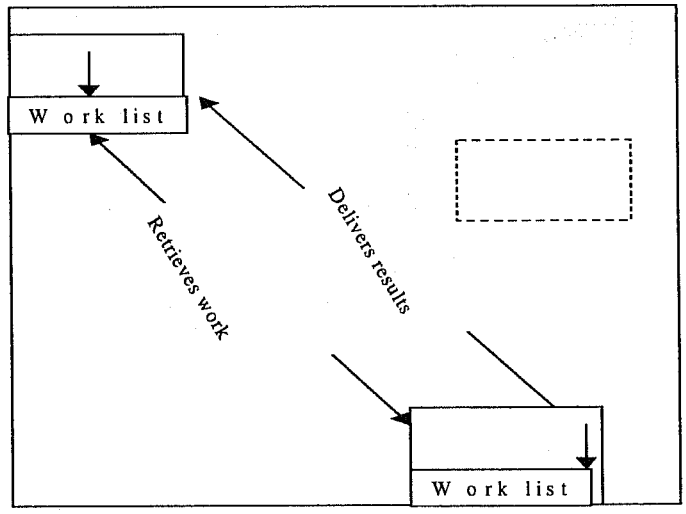

Figure 20: dynamic work redistribution by 'snooping'

5.2 Message Passing Strategies

The algorithms defined for a parallel model impose certain message passing requirements which cannot be avoided. However, there may be some room for optimisation, e.g. by buffering data in order to create longer messages. This is rather simple to do in a spectral model and may go some way towards explaining why spectral models parallelize very well despite their 'global' data dependencies.

Figure 21 compares the message lengths used in several well-known models. In order of increasing message lengths, the models are:

- unified model (UM) from UKMO (production resolution) (see Dickinson et al, 1995)
- GME (from DWD) (see Majewski, 1998)
- UM new dynamics (from UKMO) (see Davis, 1998)
- MC2 (from RPN) (see Desgagne et al, 1995)
- IFS (see Barros et al , 1995)

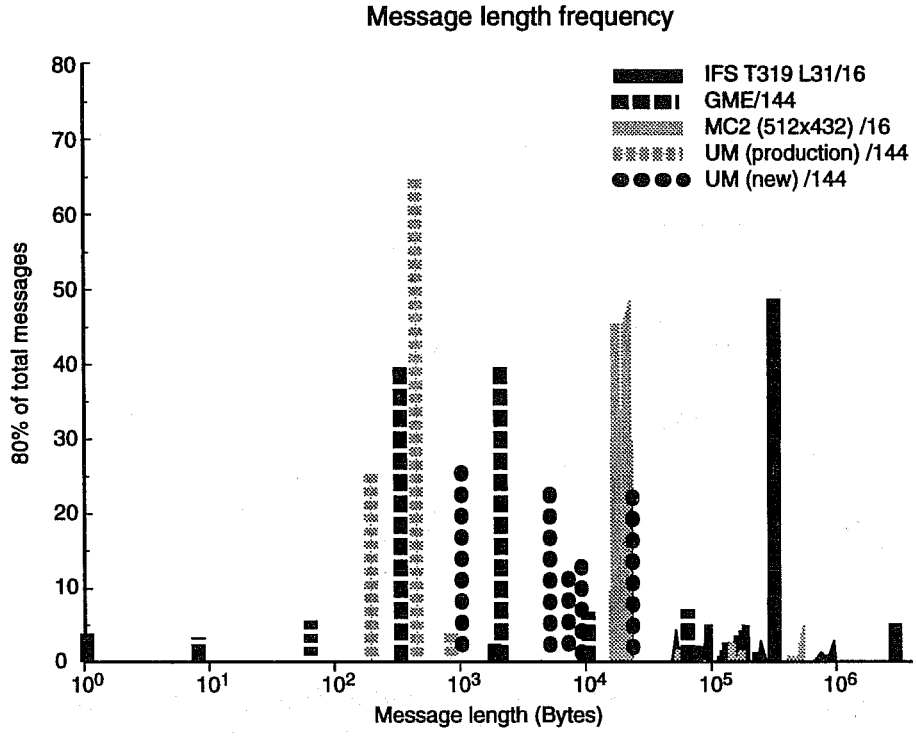

Figure 21: message length characteristics of several parallel models

Buffering of messages is used extensively in IFS and in the MC2 grid-point model, giving rise to significantly longer average message lengths for these models compared to others. However, on some platforms (e.g. T3E) the time taken to perform the local copy necessary to build up the longer buffered message can be as expensive as the cost of the remote transfer (SHMEM_PUT). Hence, many models optimised for T3E do not perform any buffering and tend to have rather short message lengths.

An additional cost overhead imposed by, for example staggered grids, is the more frequent use of barriers to provide a safe interchange of data between the grids.

Examination of message passing statistics allows models to be compared for:

(1) the locality of their message passing, where a local message is defined as one sent to a nearest neighbour processor
(2) the computational intensity (calculations per remote memory reference)

These are quantified in figure 22. Clearly, the UM is very local, especially in the one-dimensional case where processors are assigned only to latitude bands. In this case, the Fourier filtering employed over polar rows does not generate any need for additional (non-local) transfers. When run with a square distribution (4 x 4), this polar filtering communication becomes apparent, and the messages are not quite so local.

Model	*Platform*	*Pes*	*Local Messages*	*Remote Messages*	*Flops / words sent*
GME	T3E	144			356
HIRLAM	T3E	1024	55%	45%	158
IFS	VPP700	16	21%	79%	705
MC2	VPP700	16	50%	50%	140
UM (climate)	VPP700	1x16	92%	8%	173
UM (climate)	VPP700	4x4	85%	15%	324

Figure 22: locality of message passing and computational intensities for several parallel models

The other models (HIRLAM, MC2) are much less local than might be expected for grid point models due to the global nature of the semi-implicit schemes utilised. IFS statistics confirm the global nature of spectral models. There is a small degree of locality introduced by the semi-Lagrangian scheme. This increases the quantity of local messages to 21%

The computational intensities (CI), defined by floating-point operations per word communicated, vary quite a lot according to the data distribution. The square UM case has a much smaller boundary to interchange with its neighbours leading to a substantially smaller quantity of remote traffic, even though the Fourier filtering generates additional traffic. The high CI for IFS may be simply due to the expensive physics.

6. PARALLEL PERFORMANCE

Parallel efficiency is commonly used to demonstrate the scalability of parallel applications. While this can be a dangerous measure on its own, good parallel efficiency is certainly a necessary attribute. The HIRLAM (see Kauranne, 1995) performance over 1024 T3E processors is an example of what can be achieved, using dynamic load balancing and SHMEM interfaces for maximum efficiency (figure 23). The IFS scalability on a VPP has been demonstrated up to over 100 processors and up to 512 processors of T3E (figure 24).

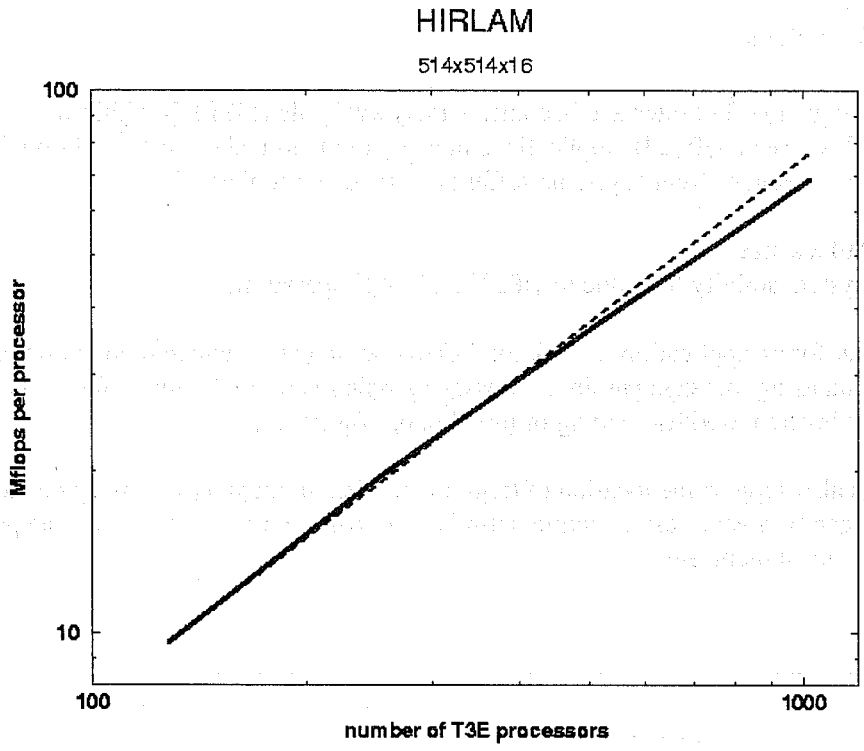

Figure 23: parallel efficiency of HIRLAM

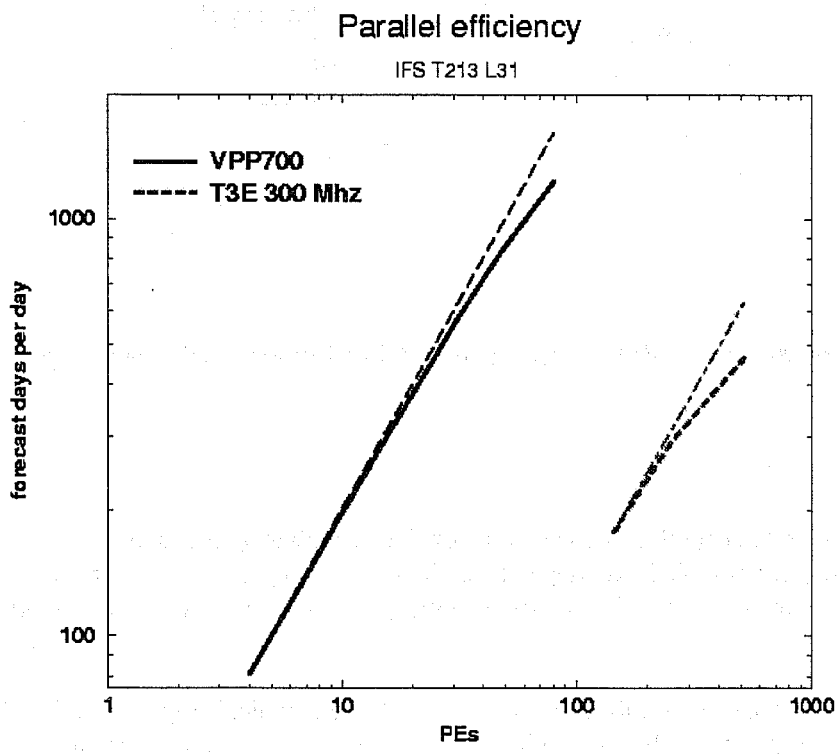

Figure 24: parallel efficiency of IFS on vector and scalar platforms

7. I/O Considerations

Until a portable parallel I/O interface becomes widely available (MPI I/O within the new MPI2 standard has been defined), applications must continue to make use of traditional single processor I/O techniques. Many systems suffer performance problems due to:

(1) small I/O block sizes
(2) frequent system activity (e.g. due to OPEN / CLOSE operations)

It is often better for an application to limit the I/O to one processor which then communicates the relevant data using message passing. Message passing is normally much faster than I/O. This technique is often used for reading of initial data (figure 25).

Many systems also support the location of (temporary) files in memory (memory resident files) and this can be a very cost-effective optimisation, particularly coupled with the present trend for larger local memories.

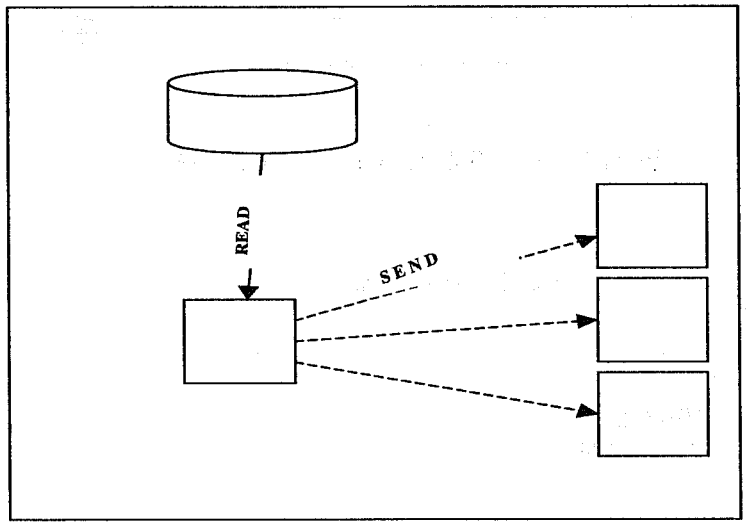

Figure 25: reading input data serially and distributing with message passing

8. Summary

There remains some architectural influence on the choice of high level algorithm, as demonstrated by the fear of global data dependency in spectral models. However, there seems to be plenty of evidence that the spectral technique can be parallelised as efficiently as grid-point models.

Clearly, the architecture has great influence on the underlying technical solutions and to some extent on the coding style, as evidenced in physics code where an unnatural code and data layout is utilised in order to achieve vector performance.

Use of vendor specific facilities for message passing allow dynamic load balancing schemes to be implemented which are very successful in overcoming inefficiencies. However, there is a danger that such techniques will not perform well when ported to other platforms.

Optimisation of I/O can be very significant for the time critical performance of operational suites. Today, the techniques use can be rather laborious to implement, but the arrival of MPI-IO should help.

Ways in which codes are optimised for different architectures mean that for the near future at least, a common code that performs equally well on a variety of platforms is still a dream.

Acknowledgements

Considerable assistance has been received and is gratefully acknowledged from:

D.Snelling (FECIT)
P.Towers (FSE)
D.Salmond and B.Carruthers (SGI/Cray)
A.Dickinson and P.Burton (UKMO)
S.Thomas (RPN)
D.Majewski (DWD)

References

MPI, 1996: MPI: The Complete Reference. The MIT Press.

MPI2, 1996: MPI-2: Extensions to the Message-Passing Interface.

A.Dickinson, P.Burton, J.Parker, R.Baxter, 1995: Implementation and Initial Results from a Parallel Version of the Meteorological Office Atmospheric Prediction Model. In: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in meteorology. World Scientific, 1995

D.Majewski. The GME global model. Proceedings of this seminar

T.Davis. The UM new dynamics. Proceedings of this seminar.

M.Desgagne, S.J.Thomas, R.Benoit, M.Valin, A.V.Malevsky. Shared and distributed memory implementations of the Canadian MC2 model. In: Proceedings of the Seventh ECMWF Workshop on the Use of Parallel Processors in meteorology. World Scientific, 1997

S.R.M.Barros, D.Dent, L.Isaksen, G.Robinson. The IFS Model – Overview and Parallel Strategies. In: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in meteorology. World Scientific, 1995

T.Kauranne, J.Oinonen, S.Saarinen, O.Serimaa, J.Hietaniemi. The Operational HIRLAM 2 Model on Parallel Computers. In: Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in meteorology. World Scientific, 1995