

DCompose: A Tool for Measuring Data Decomposition on Distributed Memory Multiprocessors

D. Abramson †
G. Cameron §

† School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
davida@cit.gu.edu.au

§ Department of Computer Systems Engineering
Royal Melbourne Institute of Technology
P.O. Box 2476V
Melbourne 3001
Australia.

ABSTRACT

In converting sequential programs for execution on distributed memory parallel processors, the programmer must determine the optimal data decomposition for the data structures. This task is an extremely complex optimisation problem and thus is usually performed manually. This chapter describes an X based visualisation tool called DCompose, which allows a programmer to measure the efficiency of different data decompositions. The system allows the user to visualise addressing patterns in the sequential program and then project them onto a parallel machine. It allows the user to define various work allocation and data placement strategies in the code. The instrumentation mechanism allows experimentation with different cache sizes and structures. It also allows users to experiment with different software based cache coherence strategies, by illustrating the point in the source code where cache incoherence occurs. A major feature of the system is that the visualisation can be performed on a sequential program without the additional complications of parallel program porting and debugging. Some examples are provided which illustrate the process.

1. INTRODUCTION

Data placement is one of the most important factors that influence the performance of programs running on distributed memory multiprocessors. Unfortunately, it is also one of the most difficult areas to optimise, especially when the ultimate performance measure, the time to

completion, is affected by so many different parameters. Further, writing programs for distributed memory machines can be a complex and time consuming task; this task could be simplified if the optimal data placement was known prior to the mapping of the algorithm onto the machine.

Data placement is an issue which should be addressed when programs are designed and when algorithms are chosen. However, there is a vast body of existing software written for sequential machines which is currently being ported to parallel platforms. To date, these have been moved to shared memory machines with relative ease, because the programmer can concentrate on the workload issues and largely ignore the communication requirements. However, moving these programs onto distributed memory machines requires a programmer (possibly someone other than the author) to understand and modify the communication which is inherent in the computation. This may involve a large amount of work for scientific programs which can span hundreds of thousands of lines of code.

This chapter describes a tool (called DCompose) which can be used to help a programmer in determining an optimal data placement for the data structures in a program by measuring the amount of communication which occurs. It is not a procedure for computing the optimal data placement, but rather an instrumentation system which measures the efficiency of a particular decomposition. Thus, it can be used profitably in combination with a manual or automatic placement scheme. DCompose displays the amount of processor-memory traffic that results from a particular program execution, in a manner which allows the programmer to understand the underlying communication. It allows the user to track the source of any inter processor traffic to the data structures and the code which manipulates them. DCompose is best used in combination with other tools such as profilers and call tree analysers. Later in the chapter we illustrate the use of DCompose and its screen layout through some examples.

DCompose is based on the premise that the inter processor communication which results from the parallelisation of a sequential program can be determined from address traces which are gathered from the execution of the sequential code. Whilst there are cases where the algorithm is changed in the parallelisation, and thus this relationship cannot be assumed, there are many cases in which the code parallelisation consists mainly of loop unravelling. This latter form of concurrency means that the central algorithm is unaltered, and thus sequential traces, which are annotated by the unrolling algorithm, can be used to model the address traces which will occur on a parallel machine.

DCompose also allows the programmer to determine experimentally when caches become incoherent, thus permitting the use of software based cache invalidation mechanisms. It has been designed to assist with mapping programs onto distributed memory machines which either use explicit message passing or provide a global shared memory address space. It is also useful for optimisation of programs on *cache-only* multiprocessors (Kendall Square Research) because it can be used to detect the need for incoherence operations, and thus reduce their frequency.

The chapter starts by reviewing different techniques for building multiprocessors and the effect this has on data placement. It then briefly describes the way data decomposition is controlled in FORTRAN D. It then discusses the mechanism used to gather the address traces, and shows the functionality of the visualisation tool. Two examples written in standard FORTRAN, and one in C are provided. The FORTRAN examples are a large Global Climate Model (GCM) and a photo chemical pollution model. The C example is a two dimensional Fast Fourier Transform.

2. MEMORY MODELS IN MULTIPROCESSORS

MIMD multiprocessors consist of a number of independent processing units and memories, connected by some high speed interconnection network. There are two main classes of MIMD machine; shared memory and distributed memory. In shared memory machines all processors can access (using primitive memory reference instructions) all of the memory units. Memory module selection is implemented in hardware using a randomisation algorithm which distributes the load across all memories. Typically, the lower order bits of the address are sufficiently random to select a module, however, more elaborate schemes have been proposed (Budnik and Kuck, 1971). In distributed memory machines, a memory can only be accessed by one processor, which is usually tightly coupled through local interconnect. Figure 1 shows generic shared memory and distributed memory systems. In some hybrid machines (for example the BBN butterfly (Crowther, et al, 1985), and Intel Paragon) the memory is physically distributed, but can be accessed by all processors as part of a shared address space. However, the disparity between local and remote access times is usually so large, that these machine behave like distributed memory ones from a performance viewpoint. In distributed memory machines the memory module is either selected implicitly (because it is the local memory of a processor) or some of the higher order address bits are used to partition the address space of the machine.

Shared memory multiprocessors rely heavily on high speed caches to hold both shared and local data. These caches hide the latency and low speed of the shared memory subsystem as well as the interconnection network. They also raise the issue of cache incoherence, which occurs when more than one processor cache contains a copy of the same variable. Distributed memory machines do not rely on caches other than for providing the normal fast access to local memory, and thus require no cache coherence hardware.

Writing a parallel program for a shared memory machine usually involves the following steps:

- 1 Determining the source and nature of parallelism in the application (loop level, remote procedure call, data parallel or explicit process creation)
- 2 Determining the strategy for allocating work to processors (work load allocation)
- 3 Deciding which variables must be globally accessible
- 4 Determining and implementing the synchronisation requirements (using shared locks, semaphores and barriers)
- 5 Optimising data locality to allow for efficient cache operation.

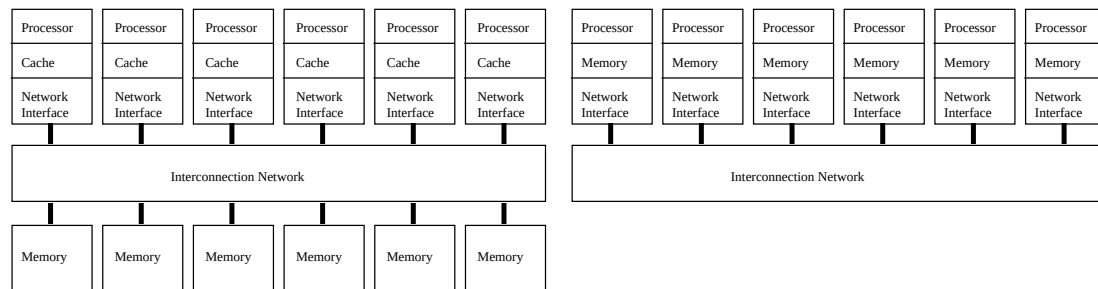


Figure 1 - Generic Shared memory (left) and distributed memory multiprocessors

Writing the same program for a distributed memory message passing machine involves a similar set of operations, however, rather than allocating variables to shared memory, the shared data structures must be transmitted between processors at the correct times. Thus, steps (3) and (4) are replaced by:

- 3 Deciding which data must be shared and when and where it needs to be transmitted and received.

Step (4) is no longer explicitly required because synchronisation is normally a by-product of having already establishing the message passing points. A new step is required as follows:

- 6 Determine the optimal data placement which minimises communication whilst allowing for good workload allocation.

This step requires knowledge of the algorithm's communication patterns as well as the volume of data which must be passed. In small programs, it is fairly easy to design optimal mappings for the data. However, in a large program this may be quite a large task. Further, a placement which is optimal for one part of the program may not be optimal for another, and thus the data must be moved and re-aligned in the memories. For example, in a two dimensional FFT if the data is transposed between one dimensional FFTs, then each of the single dimensional FFTs can be executed in parallel with no communication between them. However, without the transpose, the second set of single FFTs must continually access the data across the wrong index, causing large amounts of interprocess communication. This example is illustrated in section 5.1.

The partitioning operation is illustrated in Figure 2, in which sections of the array have been placed into different memories of the multiprocessor. This is denoted by the different shading of the array sections and corresponding memories. This operation does not arise in programming shared memory machines, however, it requires special attention and programming constructs for distributed memory systems. Whilst it is desirable for compilers to analyse programs in sufficient detail that they can automatically partition data structures, the state of the art is far short of performing this for complex programs. Thus, user level control is currently required for all but simple cases.

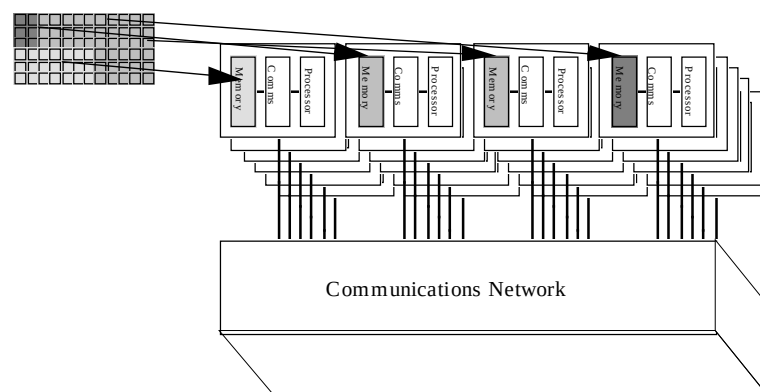


Figure 2 - Placement of data into the memories of a Distributed Memory Machine

In the next section we will discuss the constructs that have been added to FORTRAN in order to facilitate user level control of data placement.

3. CONTROLLING DATA PLACEMENT - FORTRAN D

There are now a number of modifications to FORTRAN which allow the user to control data placement. In this section we will briefly survey the techniques available in FORTRAN D, a language which has been developed under a consortium between Syracuse University and Rice University under the Centre for Research on Parallel Computation (Fox, et al, 1992). Similar constructs are available in High Performance FORTRAN (HPF) and industry versions of FORTRAN-90 such the Thinking Machines FORTRAN for the Connection Machine (CMF) and the Maspar compiler (MPF). Whilst all of these languages support data parallelism as the programming paradigm, their data decomposition constructs are independent of the techniques which are used to expose the parallelism. There are three main statement classes; those which declare data shapes or decompositions, those which align different shapes and those which actually describe the distribution of data onto the real machine.

3.1 Declaring Data Shapes

FORTRAN D uses a new statement called DECOMPOSITION to declare a name, dimensionality and size of structures for each problem mapping. To quote the FORTRAN D user's manual, "A decomposition is simply an abstract problem or index domain. No storage is allocated for a decomposition". The following examples declare two new shapes called A and B. A is a single dimensioned decomposition of size N and B is two dimensioned of size N x N.

```
DECOMPOSITION A(N)
DECOMPOSITION B(N,N)
```

The Decomposition statement is similar to the type declarations found in other programming languages except that it only applies to the problem of data placement.

3.2 Aligning Data Shapes

The Align statement is used to map array to a particular data decomposition, and thus to align data structures to each other regardless of the way that they eventually get mapped onto the underlying memory modules. The Align statement uses *place holders* for array indices, which can be used to specify both simple and very complex alignments.

There are many options available with array alignment, and the interested reader is referred to the user's manual for details. For example, complex permutations and collapse operations are possible using static alignments. The following example in Figure 3 shows a few alignment statements and the resulting outcomes. Here, the array X1 has been shifted by one element with respect to the alignment, and X2 has been shifted by one element in the other direction. Similarly, X3 has been shifted along one axis, even though it has two dimensions, and X4 has been shifted in two directions.

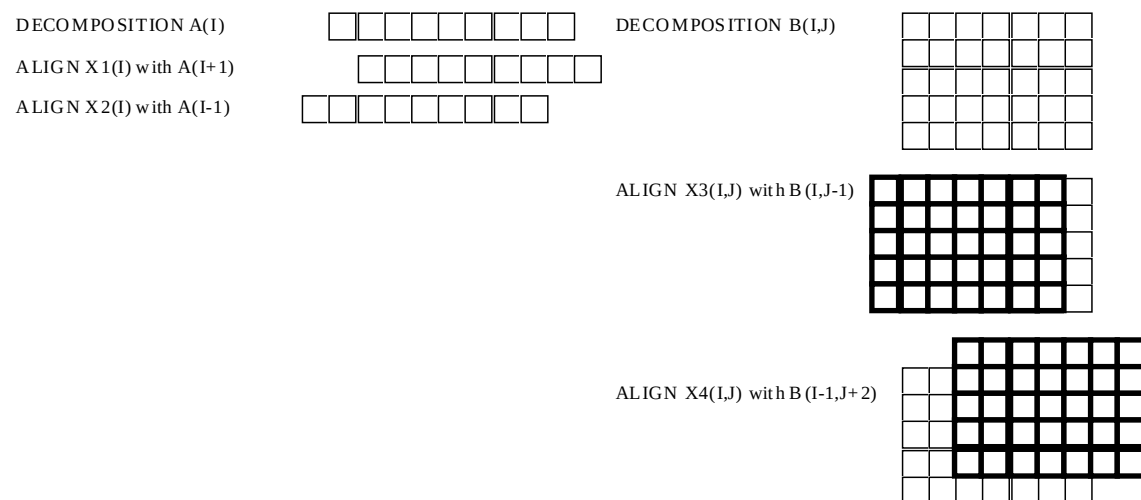


Figure 3 - Examples of the ALIGN statement in FORTRAN D

3.3 Distributing the Data over the Machine

The DECOMPOSITION and ALIGN statements indicate how data is to be viewed, but do not specify how it is to be mapped onto the memory modules. This is controlled by the DISTRIBUTE statement. It is possible to distribute the data structure with respect to any or all of its indexes over the memory modules of the machine. Three attributes can be assigned to each index; they are denoted by the words BLOCK, CYCLIC and BLOCK_CYCLIC. Figure 4 shows a few examples, however, there are many more in the FORTRAN D user's manual. Here, the decomposition A is allocated to processors named p1 through p4 using each of the possible distributions. BLOCK allocates contiguous regions of memory space within one module to *chunks* of the array. CYCLIC spreads the array out across all memory by allocating contiguous *chunks* across the modules. BLOCK_CYCLIC performs a combination of the two.

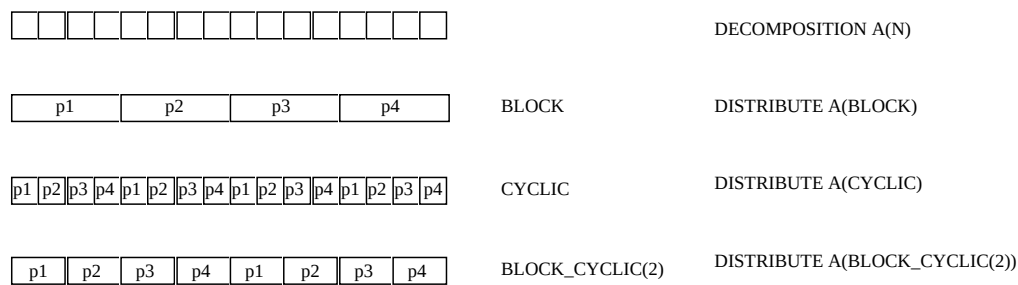


Figure 4 - Examples of DISTRIBUTE statement

4. PERFORMANCE MONITORING TOOLS

In the previous two sections we showed why distributed memory mutliprocessors require data to be partitioned into memory modules, and how this is controlled using modern parallel languages like FORTRAN D. In this section we will discuss the need for software tools which assist the user in finding optimal partitions without the need to run the program on an existing parallel system.

4.1 A Tool for Estimating Communication Requirements

Almost all of the existing parallel machine manufacturers provide performance monitoring software which assists the user in optimising the execution of a given program. For example, products are available for machines from Thinking Machines (Thinking Machines Corporation, 1993), NCube (NCube Corporation, 1992), Maspar (Maspar Computer Corporation) and Cray (Cray Research, 1993). The most common form of these provides performance profiles, which indicate the amount of time spent in particular sections of code. A number of tools provide information about the amount of interprocessor traffic that results from access to data structures that are not stored locally, however, they suffer from two major problems:

- The program must be rewritten in a parallel programming language, such as HPF, and then run on the parallel platform on which performance statistics are gathered
- Many of the available tools only provide global statistics which cannot be traced back to individual algorithmic constructs in the code.

The first of these problems occurs when old programs are converted for execution on parallel systems. The original programs are often large, poorly structured for parallel execution, and written in sequential programming languages. Whilst they must usually be rewritten, or substantially modified, for execution on parallel hardware, it is desirable

to determine efficient data partitioning strategies before the code is rewritten. In this way, a number of different partitioning scenarios can be studied on a version of the program which already works. For example, the program can be executed under different test conditions and with input data. None of the existing tools are capable of this level of performance modelling.

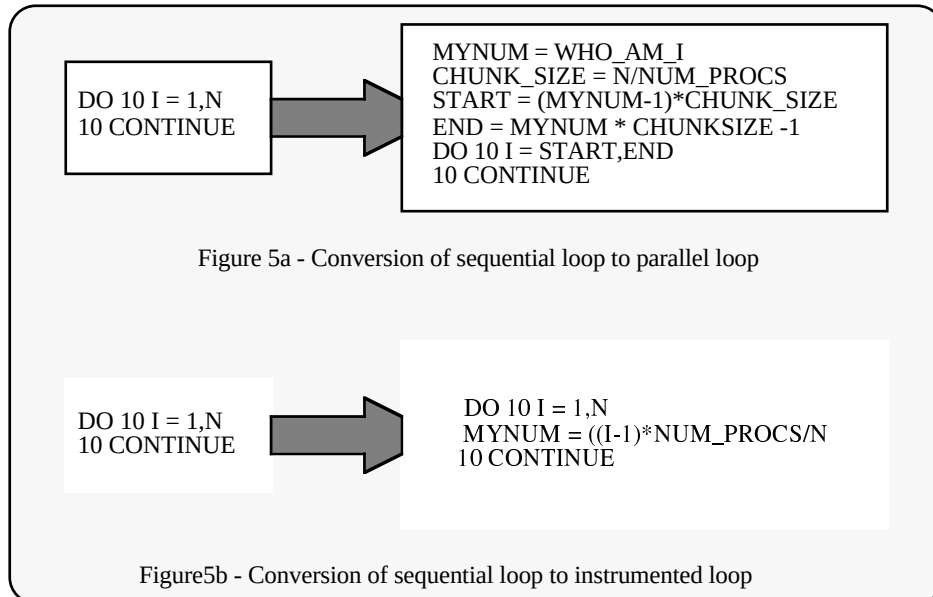
In order for this approach to work it is necessary to execute a program sequentially with some information about the potential parallel behaviour. The next section discusses this requirement and proposes a solution.

4.2 Predicting Performance from Sequential Execution

DCompose uses a sequential execution of the program to determine the communication requirements when that program is placed on a distributed memory machine. The primary output of DCompose is an indication of the amount of traffic between processors and the distributed memory units, and information which allows the traffic to be traced back to the source code.

In order to predict the communication patterns on a multiprocessor the sequential program must be augmented with two sorts of information. The first is the data decomposition for the shared variables, and the second is the workload allocation strategy. The data decomposition could be specified using the FORTRAN D syntax discussed in the previous section, although we have used a different mechanism at present. The workload allocation strategy requires the parallel sections of the program to be augmented with code which assigns work to pseudo processors. For many scientific codes this is performed through loop slicing. An example is illustrated in Figure 5. Most implementations of loop slicing take a sequential loop and use the processor identity to determine the bounds of the new parallel loop, as shown in Figure 5a. Thus, in this example, the variables START and END are computed using the original loop size (N) and the number of processors (NUM_PROCS). Each processor computes its own local START and END. In the instrumented code the system must determine the processor identity from the loop iteration value. This can be evaluated by computing the inverse of the normal loop slicing code as shown in Figure 5b. The instrumentation harness maintains track of the current processor for use in address interpretation. This simple scheme means that addresses can be tied to the processor that generated them without the need for a complex multi-tasking operating systems. Of course, it is not possible to obtain statistics which rely on the relative timing of addresses generated by processors, since the parallel loop is still

executed sequentially. However, for the purposes of predicting the processor-memory communication requirements, the technique is very effective.



4.3 Gathering Address Traces

In order to determine the amount of traffic between the processors and the memory units, it is necessary to produce an address trace for the program execution. In this section we discuss the different methods available for generating address traces, and describe the one used by DCompose.

Address traces have been produced traditionally by five main techniques (Stunkel, Janssems, and Fuchs, 1991): simulation (So et al 1988; Kumar, and Davidson, 1978; Axerold, Dubois, and Elgroth 1984; Mitchell, and Flynn 1988; Goldschmidt and Davis 1990), interrupt based tracing (Wiecek 1982; Eggers and Katz 1988), instrumentation [Borg, Kessler, and Wall 1990; Strunkel and Fuchs 1989; Eggers et al, 1990; Larus, J. 1990; Francis, Mathieson and Pears 1991), firmware modification (Agarwal, Sites. and Horowitz 1986) and hardware monitoring (Horst, and Metz 1984), each of which has advantages and disadvantages. Usually an address trace file is produced for post processing. Simulation can occur either on the high level source code or by simulating the machine code instructions produced by the compiler. Such schemes have the disadvantage that they are extremely slow, making it difficult to gather large address traces from real applications. Simulation has the advantage that the address trace file can be bypassed if each address can

be processed as it is generated, which removes the need for enormous trace files.

Firmware interpretation involves modifying the microcode of a machine to allow all memory addresses to be captured before they are used. This scheme is much faster than software simulation, however, it usually requires that the addresses are written to a memory buffer, which is then written out to a file for later processing. The scheme also relies on access to the system microcode, which is not always possible. Hardware monitoring, provides the most accurate tracing mechanism because it does not effect the speed of the executing program. However, it requires access to the memory hardware which is not always available. This form of tracing also has the advantage that multiprocessor hardware can be instrumented, allowing different bus and cache structures to be evaluated. Interrupt based schemes can be as slow because the operating system must be entered after each instruction. All of these schemes allow both data and instruction references to be traced.

Instrumentation based schemes modify the program to be traced, either by automatic or manual processing of some form of the program. In general, this only allows the program which has been modified to be traced, and instruction address traces may be produced. The framework described in this chapter is an instrumentation based scheme, and is shown in Figure 6. The source code, in this case FORTRAN or C, is processed using the standard compiler, however, assembler text is generated instead of the usual machine code. The assembler text is then processed by a special tool, LDST, which identifies all load and store instructions. Since most RISCs only provide access to memory through the load and store instructions (and a few special instructions and variants of loads and stores), it is possible to capture to value of the address used in the load or store. LDST prefixes each load and store by code to evaluate the target address, and then plants code which calls a general-purpose C routine which is linked with the application. On the SUN Sparc, this involves converting LD and ST instructions into ADD instructions, which execute the same sequence to compute the effective virtual address, but simply store the result into a register rather than using it to address memory. The target address is passed as a parameter to the routine, and is thus available to general C code for interpretation. In the implementation described in this document, the address is used on-the-fly rather than being written to a trace file. Program execution is not slowed down as much as in a full software simulator because most instructions are executed directly by the RISC processor. Only the loads and stores

are slowed down by the tracing code, yielding a slow down of approximately 3 times¹. Figure 7 shows a sample piece of assembler used as input and output of LDST. In this example, the single LD instruction has been converted into 10 instructions. The first 3 simply free up some processor registers so that an external C routine can be called. This involves saving them on the main memory stack. The **add** instruction computes the effective address which was specified in the original load instruction. The CountLd routine is then called, passing the computed address (in o0) as a parameter. Following the call, the spare registers are restored and the load performed as intended. The stack adjustment is necessary because of the position in which the compiler leaves the stack pointer.

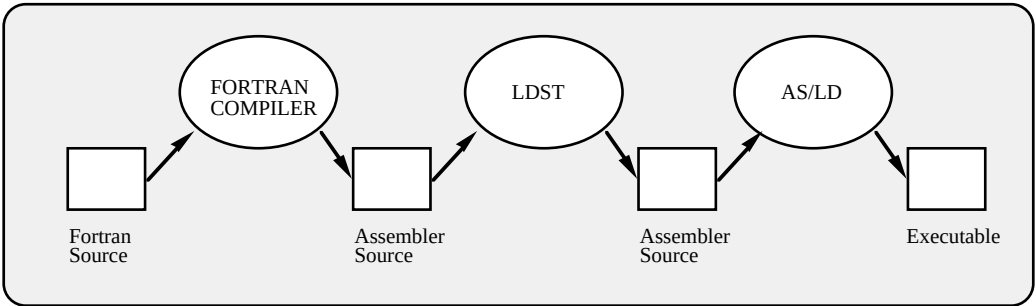


Figure 6 - Software Instrumentation Environment

Input	Output	Comment
	add -128, %sp, %sp	Get Free Memory on Stack
	st %o7, [%sp+124]	Free up o7 register
	st %o0, [%sp+120]	Free up o0 register
	add %l7,-4096,%o0	Compute Address
	call _CountLd, 1	Call load trace routine
	nop	
	ld [%sp+120], %o0	Restore o0
	ld [%sp+124], %o7	restore o7
	add 128, %sp, %sp	Free spare memory
ld [%l7+-4096],%l6	ld [%l7+-4096],%l6	Do the original load

Figure 7 - Output of instrumenter

¹ This time is for simple tracing in which some counters are incremented each time the routine is called.

4.4 Mapping addresses to memory modules

In order to determine the memory module number from the virtual address it is necessary to decode the address to recover the original array subscripts and to determine which array is being accessed. This decode computation requires the use of *div* and *mod* operators. Figure 8 shows this extraction where the row and column index limits are powers of 2. DCompose currently supports both methods of extraction.

To account for different data type sizes, an entry for each type is stored in a translation table. This table contains the maximum index sizes, as well as the data type size and the range of addresses for the structure. When an address is traced, the table is searched to find the data type information, which is then used during row and column extraction. At the beginning of the program an initialisation call is performed which sets the contents of the translation table.

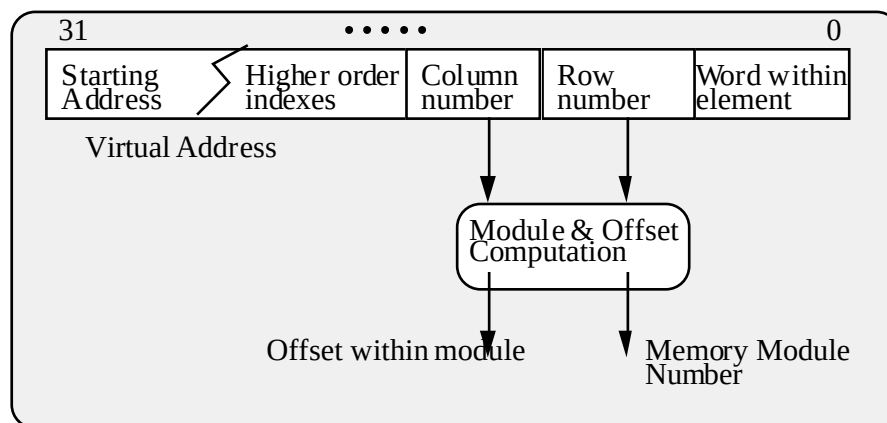


Figure 8 - Extraction of array subscripts

4.5 Visualising the address patterns

Figure 9 shows a sample of the user interface to DCompose. The amount of traffic is displayed using grey scale (and colour if available) squares, where the intensity denotes the relative amount of traffic. A new legend is computed for each display as is shown on the right hand side of the screen. Given the matrix representation, the local memory references (f) are found on the diagonal, and the non-local references (e) in the other squares. Given a particular network topology and processor number assignment it is also possible to spot other access patterns like row and column local access on a mesh. Four filters are available, namely selection of the code procedure (c) (or explicitly named section of code), selection of data structures (b), selection of line numbers (a) or type of access (read/write). Initially the traffic displayed is the sum of memory references across all code, data and lines of code. By placing specific

filters on the amount of data displayed it is possible to isolate critical traffic patterns, such as non-local references. If there are more processors than can be displayed in the region of the screen, then the user has the option of squashing the pattern into the available space, or showing a section. The former is useful for spotting global patterns, the latter for examining detail. We have trialed the system with up to 512 processors successfully. If only part of the total traffic pattern is displayed, then a panner (d) is available for moving the window.

Figure 10 shows the effect of clicking the mouse over a particular traffic square. The user is presented with a list of sources for the traffic, together with the number of reads or writes.. In Figure 11 the user has selected a particular line and the source is then displayed.

In the next section we will examine three applications of DCompose.

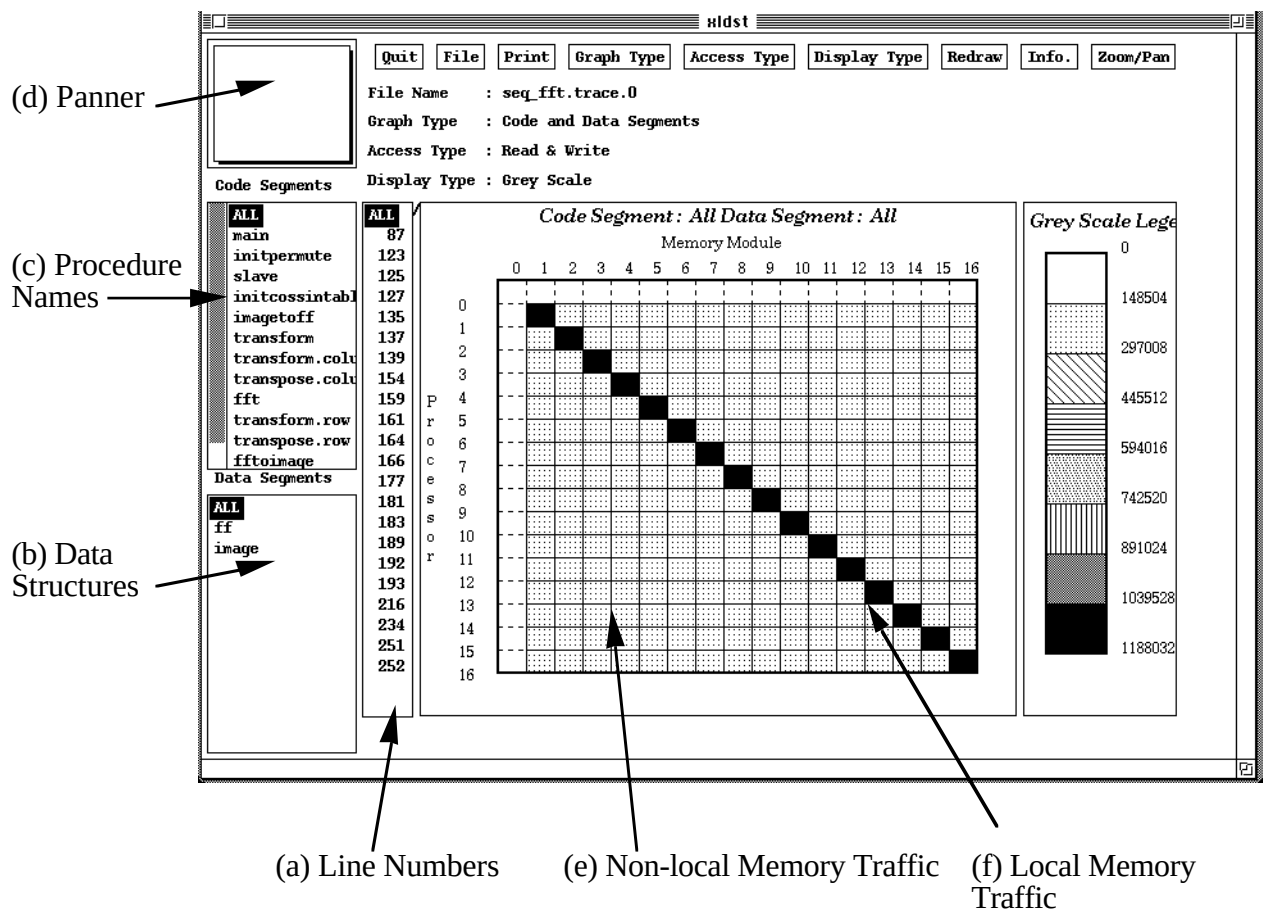


Figure 9 - Example of DCompose User

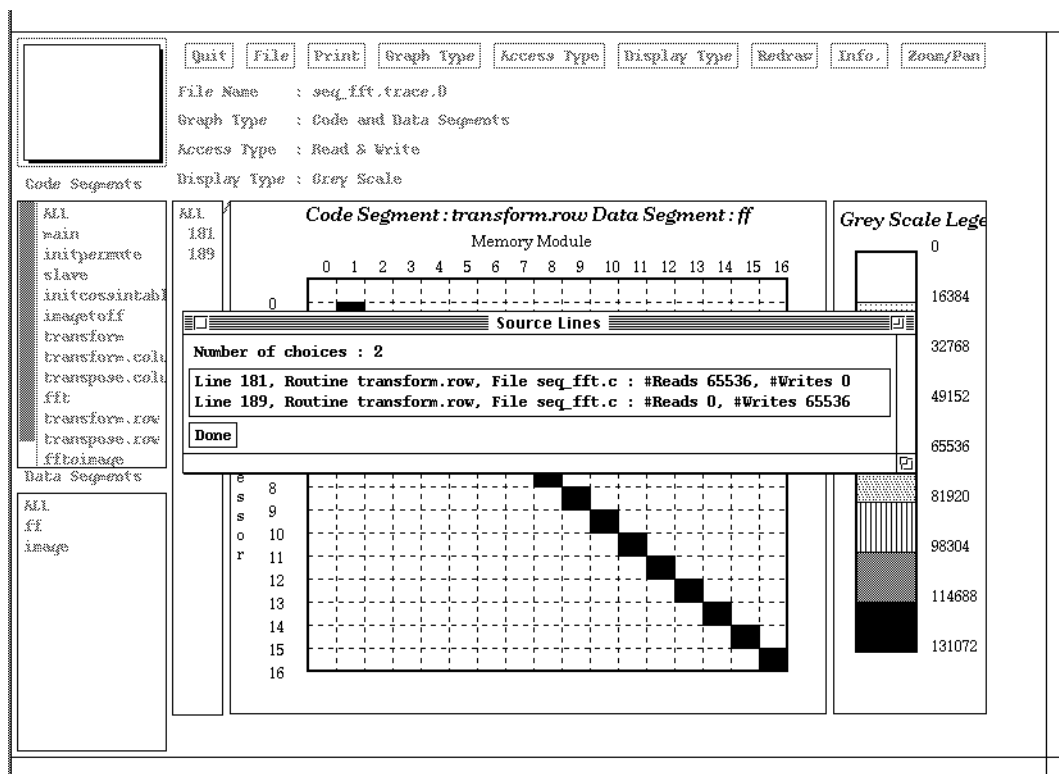


Figure 10 - Selecting a processor-memory pair

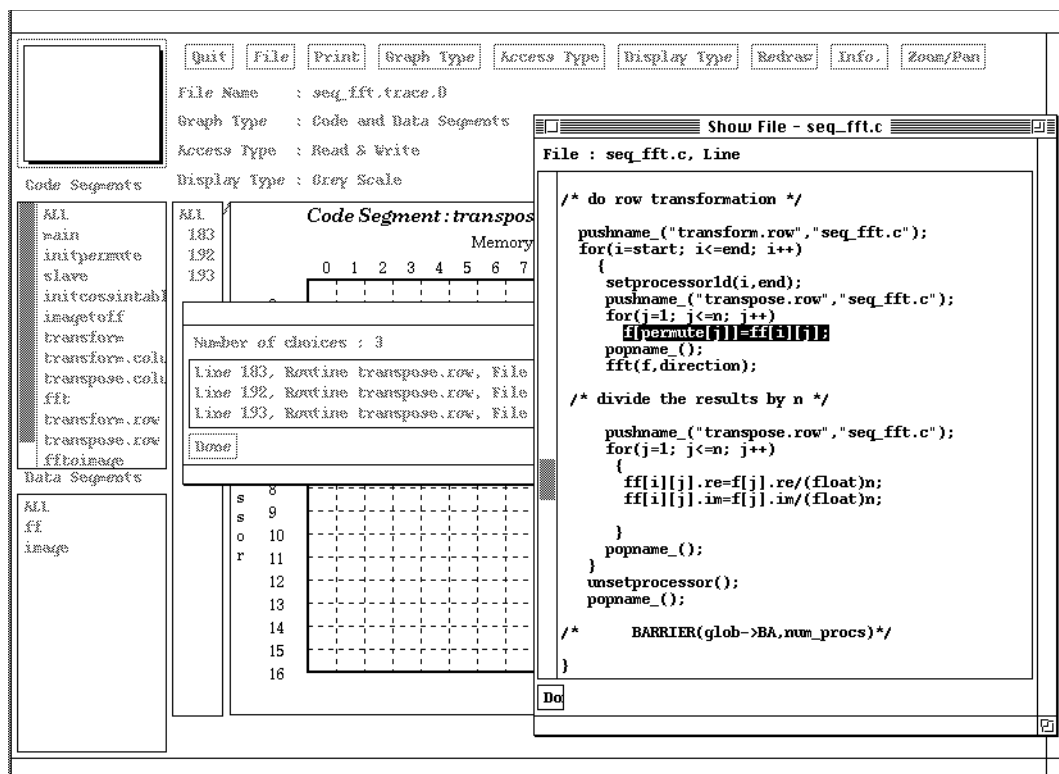


Figure 11 - Tracing addressing traffic back to the source

5. SOME EXAMPLES

In this section we illustrate the use of DCompose by showing the instrumentation of a small demonstration program (a real image processing example) and some large scientific programs (a global climate model and a photo-chemical pollution model).

5.1 A Two Dimensional FFT

This code was originally written for a sequential work station, and was then parallelised for a shared memory multiprocessor. The program performs two two-dimensional fast Fourier transforms on an image; one to transform an image into a spectral representation in which some noise can be removed, and the one to return the spectral form to an image. The image is represented using a two dimensional array of pixel values as shown in Figures 12 (a), (b) and (c). The two-dimensional transform is performed first on the rows of the image, and then on the columns. Clearly this type of access pattern causes problems for data partitioning, because an optimal partitioning for one phase of the transform is not optimal for the other. This is illustrated in Figure 9, which shows the transform composed of both local and non-local accesses. Figure 13 shows the column transform, which generates a large amount of non-local accesses, whilst Figure 14 shows the row transform which is all local. Of course, in this application there is no data mapping which can satisfy both mapping. If a transpose of the array is placed between the two phases, then both will show local memory traffic.

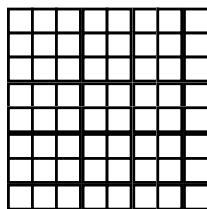


Figure 12 (a)
Image representation

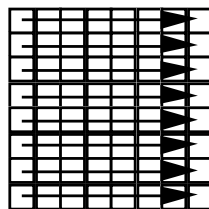


Figure 12 (b)
FFT across rows

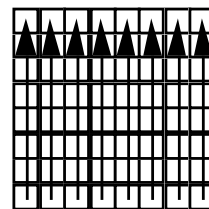


Figure 12 (c)
FFT along columns

Figure 12 - Method of processing image

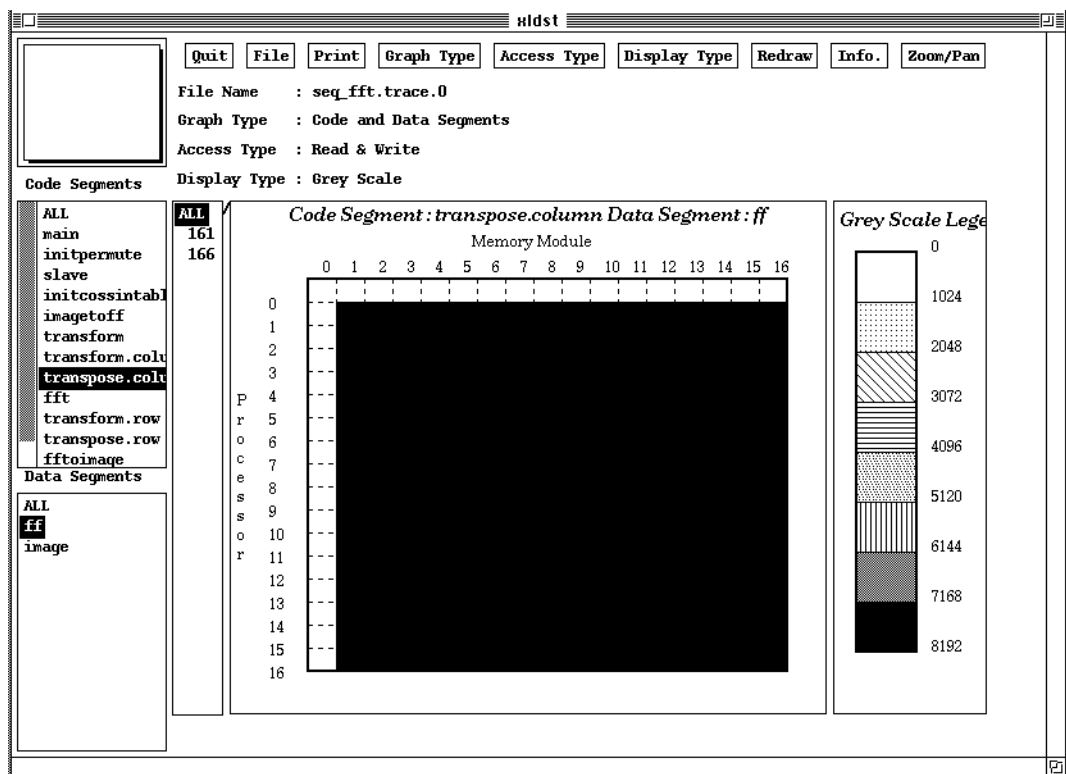


Figure 13 - Column access in FFT

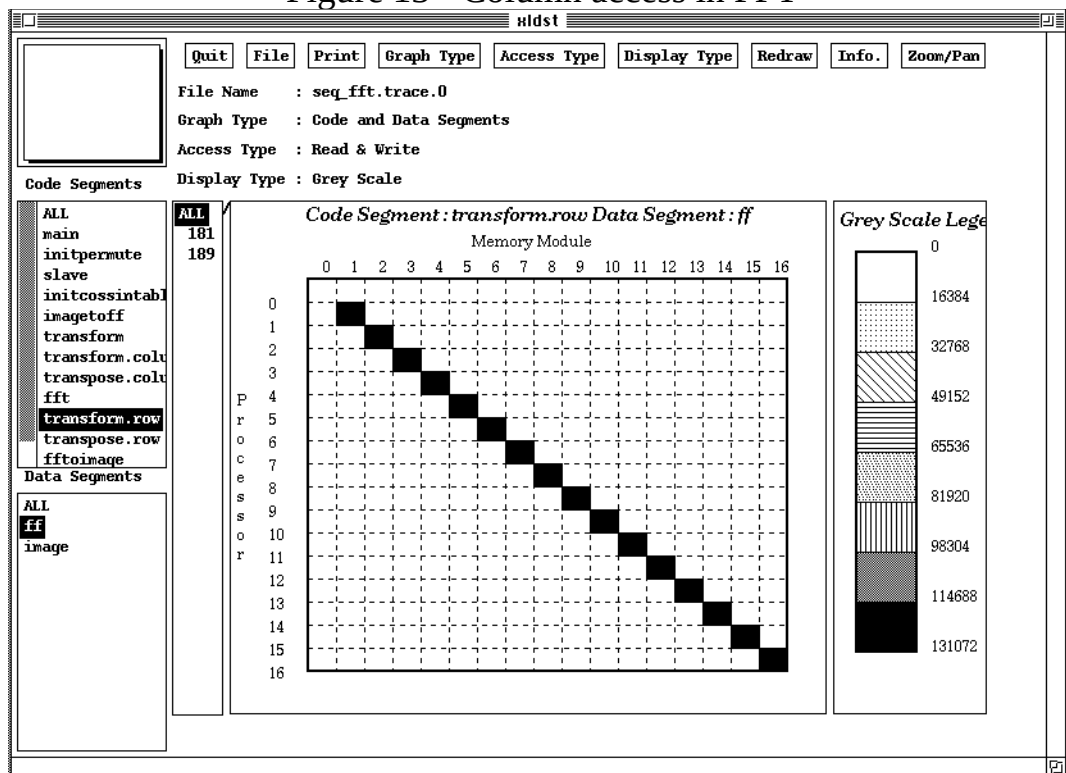


Figure 14 - Row accesses in FFT

5.2 1 A Global Climate Model

This study involved the parallelisation of a Global Climate Model (GCM) called CSIRO4. CSIRO4 was developed by the Commonwealth Scientific and Industrial Research Organisation (CSIRO) Division of Atmospheric Research (Rotstayn and Dix 1993) for studying the effects of climate change. CSIRO4 was originally written for sequential processors, but has been ported to a number of vector machines over the years, and has also undergone significant development work. GCMs attempt to model the behaviour of the atmosphere by considering it as a fluid undergoing a number of physical processes. Consequently, the model is composed of two main phases. First, a simplified form of the Navier-Stokes equations are solved to determine the state of the fluid. This is referred to as solving the *dynamics*. Second, the real world physical processes are modelled using approximations of their global behaviour. This is referred to as solving the *physics*. As shown in Figure 14, the atmosphere is composed of a number of grid cells, which are mapped onto a three dimensional data structure, indexed by longitude, latitude and height. At each grid cell a number of variables are represented, such as temperature, pressure, etc.

There are a variety of possibilities for mapping the three dimensional structure onto a parallel machine. The following FORTRAN declaration defines TEMP, which represents the temperature at any point in space.

```
REAL TEMP(LONG,LAT,HEIGHT)
```

Using a FORTRAN D decomposition the following BLOCK combinations are a few of the potential mappings for TEMP:

1. DISTRIBUTE TEMP(BLOCK,BLOCK,BLOCK)
2. DISTRIBUTE TEMP(BLOCK,BLOCK,*)
3. DISTRIBUTE TEMP(BLOCK,*,*)
4. DISTRIBUTE TEMP(*,BLOCK,*)

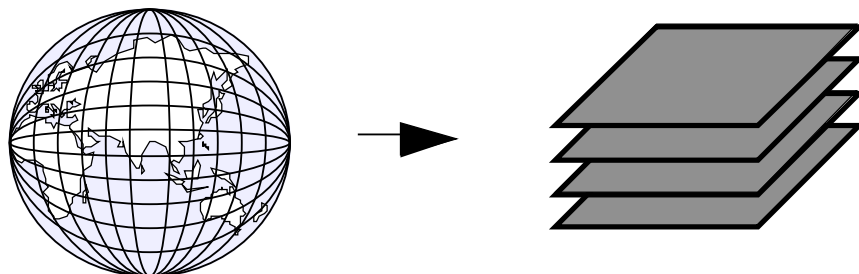


Figure 14 - Mapping of earth's atmosphere onto three dimensional array

The first of these divides the space into cubes and maps each one onto a memory module. Thus, any accesses from a processor within the cube are local, but others require remote accesses to other memory modules. The second mapping folds all layers of the atmosphere onto the same module, thus any accesses up and down columns of the atmosphere remain local. It turns out that this type of mapping suits the modelling of the physics which requires accesses to columns of data. The remaining two mappings distribute by longitude or latitude only. At some points of the dynamics distribution number 3 is suitable and at some points distribution number 4 is appropriate.

Figure 15 shows the execution of the model when mapped onto a 4x4 mesh of processors using a number of different DISTRIBUTIONS for the various variables of the model. A few different patterns can be observed. First, all processors perform a certain amount of local data access. Second, all processors access memory module 1. This is because the global variables and constants are not distributed, but are stored in one memory module (number 1). In order to remove these accesses it is necessary to distribute copies of the data on each time step of the program. The other two dominant forms of communication are row and columns major. In some phases the program must access variables at the same longitude, and this causes each processor to access the memory modules stored in the same row of the mesh. Likewise, during some phases a processor must access all variables at the same latitude, and this is demonstrated by column accesses in the mesh. The picture presented in Figure 15 was the result of a large amount of design and experimentations with different mappings. Since the CSIRO4 model is in the order of 10,000 lines of code, it was an advantage to be able to perform the different mappings on the original sequential FORTRAN source running on a SUN Sparc Station.

In Figure 16 we show some more detailed pictures of the CSIRO4 model on the same mesh of processors. In this picture some of the physics processes have been selected, demonstrating that they only generate local memory references and accesses to the global constants in memory module 1.

Figure 17 shows the effect of performing Fourier transforms across the latitudes of the model. This is required when the data is transformed from the grid space representation discussed here to another representation in spectral space. The details of this operation are available elsewhere (Rotstajn and Dix 1993).

In summary, DCompose allowed us to experiment with different data decompositions without any major modifications to the sequential code. These were performed without access to any particular parallel machine.

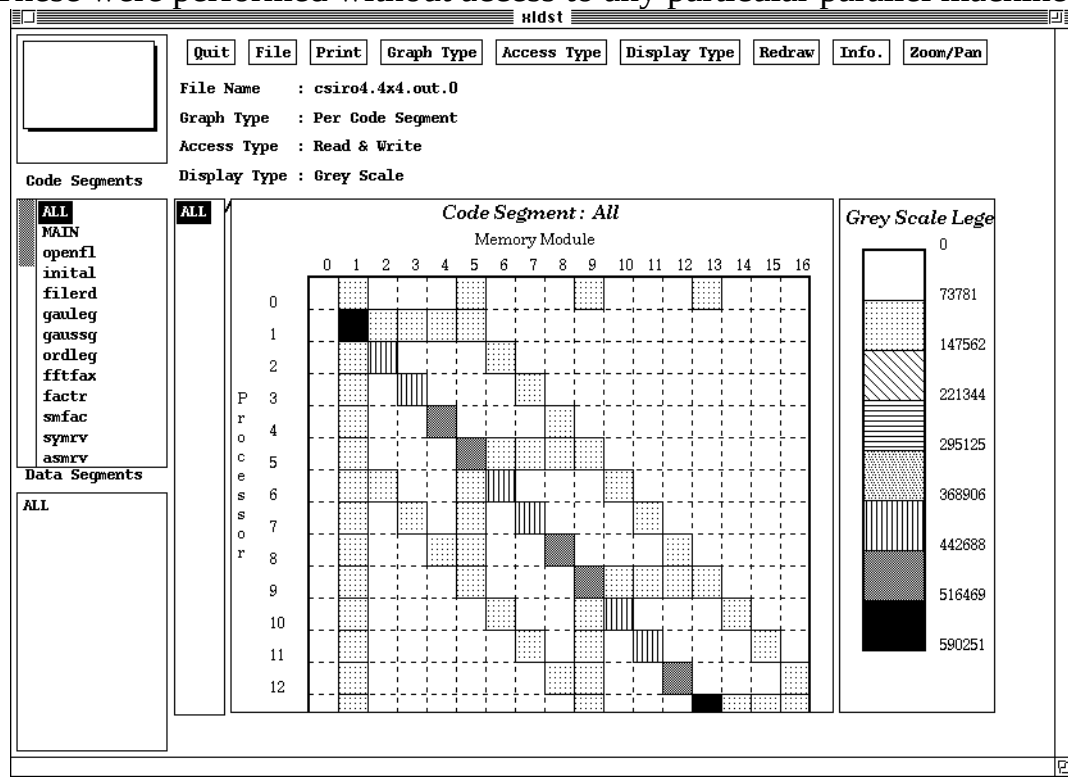


Figure 15 - Placement on 4x4 mesh

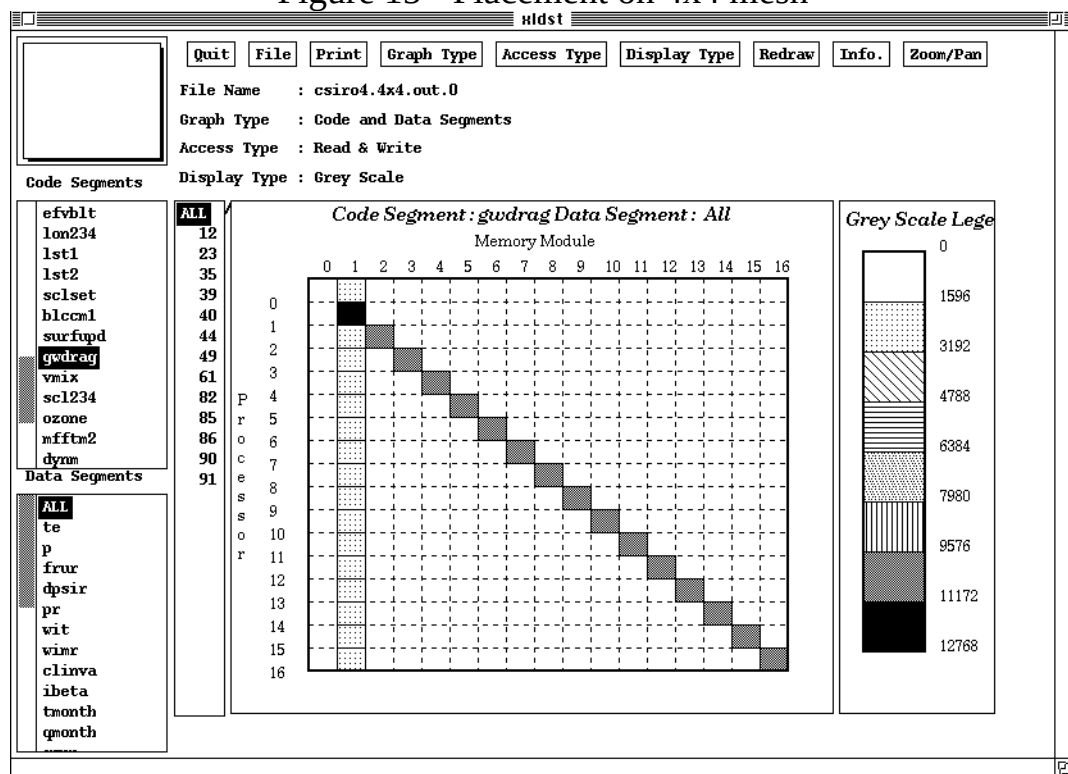


Figure 16 - Access caused by physics processes

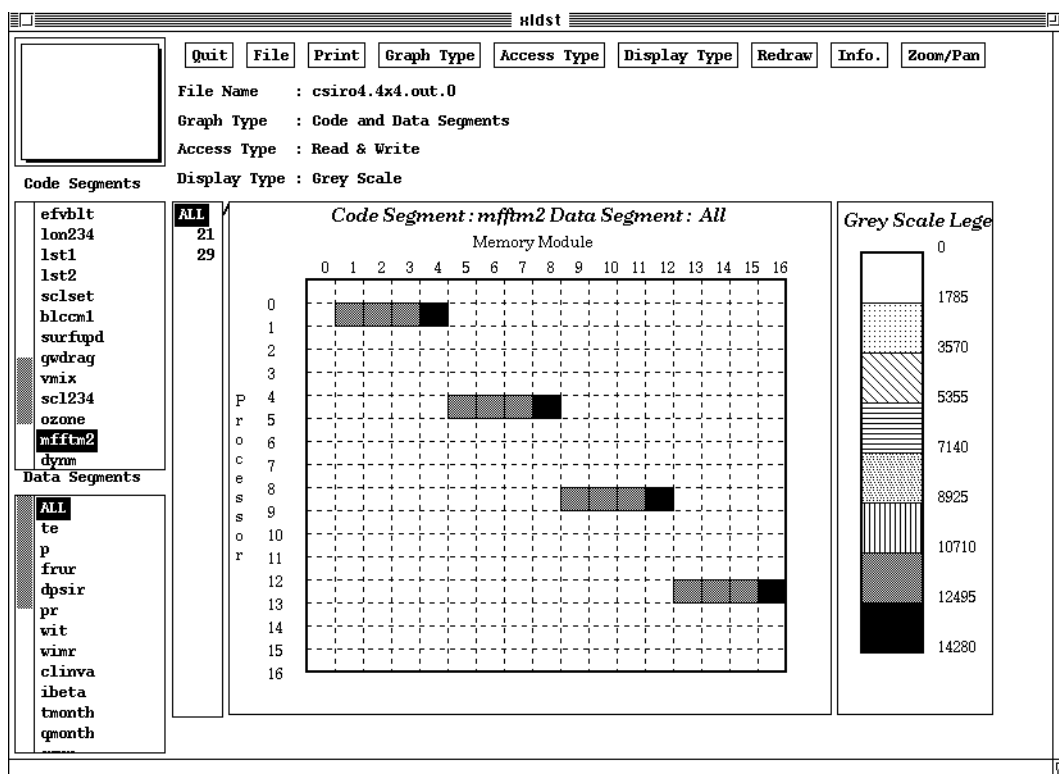


Figure 17 - Accesses caused by Fourier transforms

5.3 A Photo-chemical Pollution Model

This study concerns the parallelisation of a program (AIRSHED) which models the chemistry of photochemical pollution. It was originally designed at Caltech by McRae (McRae, Goodwin and Seinfeld 1982) and has been used for a number of pollution studies around the world. AIRSHED models both the chemistry that occurs when various precursor species are mixed in the atmosphere and exposed to ultra violet radiation, as well as the transport that occurs due to local meteorology. AIRSHED has been ported to a number of sequential, vector and shared memory machines with relative ease. DCompose is being used to determine the optimal decomposition of the data structures for execution on distributed memory processors. Figure 18 shows the basic computational structure. Like CSIRO4, the atmosphere is divided into cubes which are mapped onto a three dimensional array. The concentration of each chemical species is stored in another dimension, thus the concentrations can be saved in a four dimensional array CONC as follows:

CONC(Z,X,Y,SPECNO)

AIRSHED contains two main phases. The first solves the chemical interactions which occur in any major time step. These are modelled by a

set of very stiff ordinary differential equations, and each grid cell is computationally independent of any other. This is illustrated in Figure 18 by the chemical equations in the expanded cell. The second phase performs transport of species around the grid in the X, Y and Z directions. The wind vectors are pre-computed using a separate prognostic wind field model (Goodwin, McRae and Seinfeld 1979). Each time step a new set of wind vectors are read and used to disperse the chemical species. The modelling process continues for many simulated hours in order to compute pollutant concentrations.

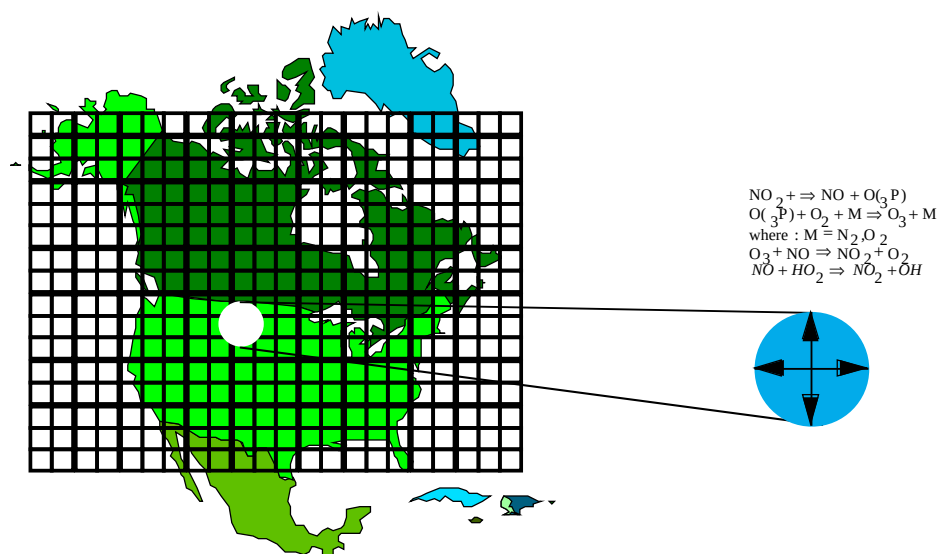


Figure 18 - Mapping of atmospheric chemistry model

There are a number of ways of parallelising this model. Since the chemistry is independent at each grid cell, these form a natural form of concurrency. Also, the dispersion of the chemical species is done in three separate passes; one for each dimension in 3d space. Consequently, all of the latitudes of space can be dispersed concurrently, followed by all of the longitudes, followed by all of the columns. This structure produces an even workload distribution across the processors and good speedup is observed. However, there is no data partitioning which suits both the chemistry and all of the dispersions passes. For example, the best way to partition the data for the chemistry is by using a BLOCK decomposition on one of the 2d spatial indexes, and then parallelising the loop over one of these indices. This is illustrated in Figure 19 in which the X index is used to block the concentration array, and it results in totally local data accesses.

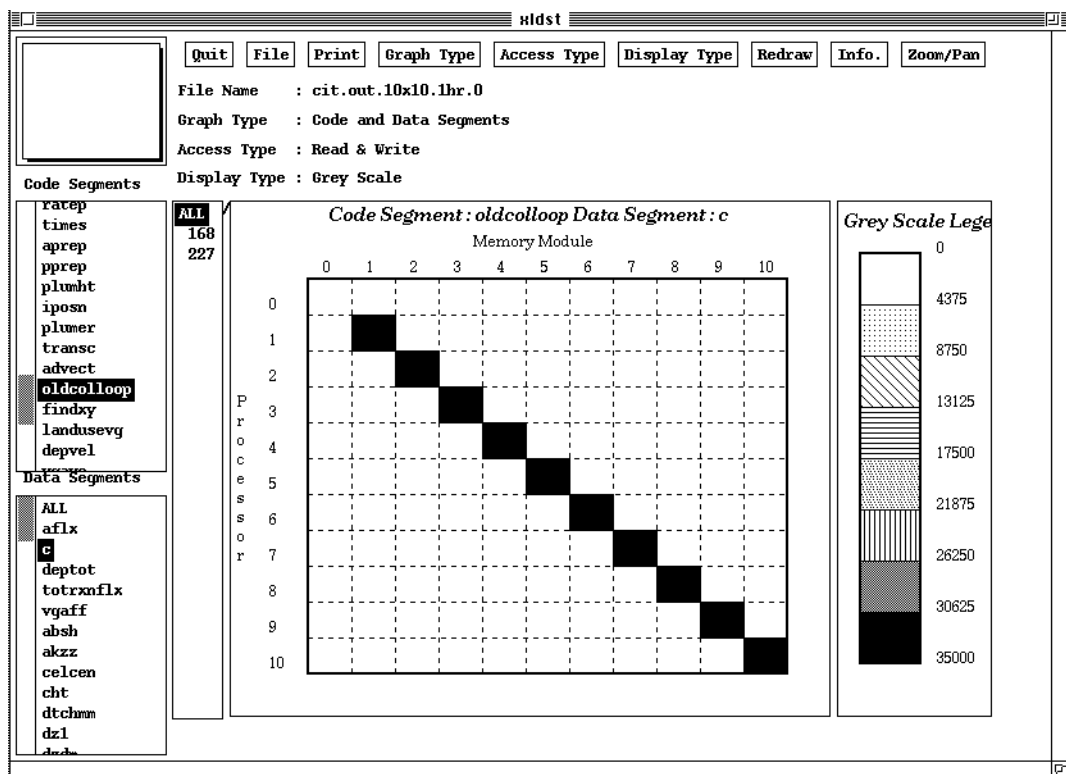


Figure 19 - Accesses to concentration vector in chemistry

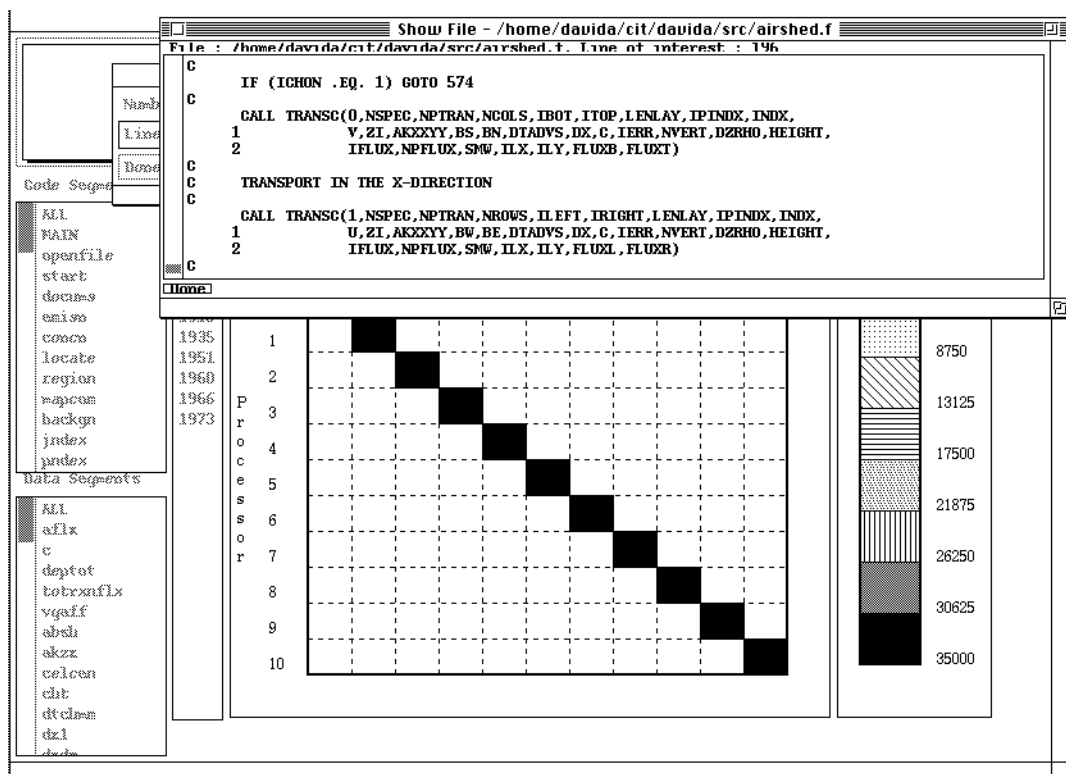


Figure 20 - Accesses to concentration vector during x transport

The same decomposition allows the transport code in the X direction to also exhibit local behaviour, as shown in Figure 20, however, the transport in the Y direction requires a transpose of the data in order to keep accesses local. Figure 21 shows this effect in which all processes access all memory modules.

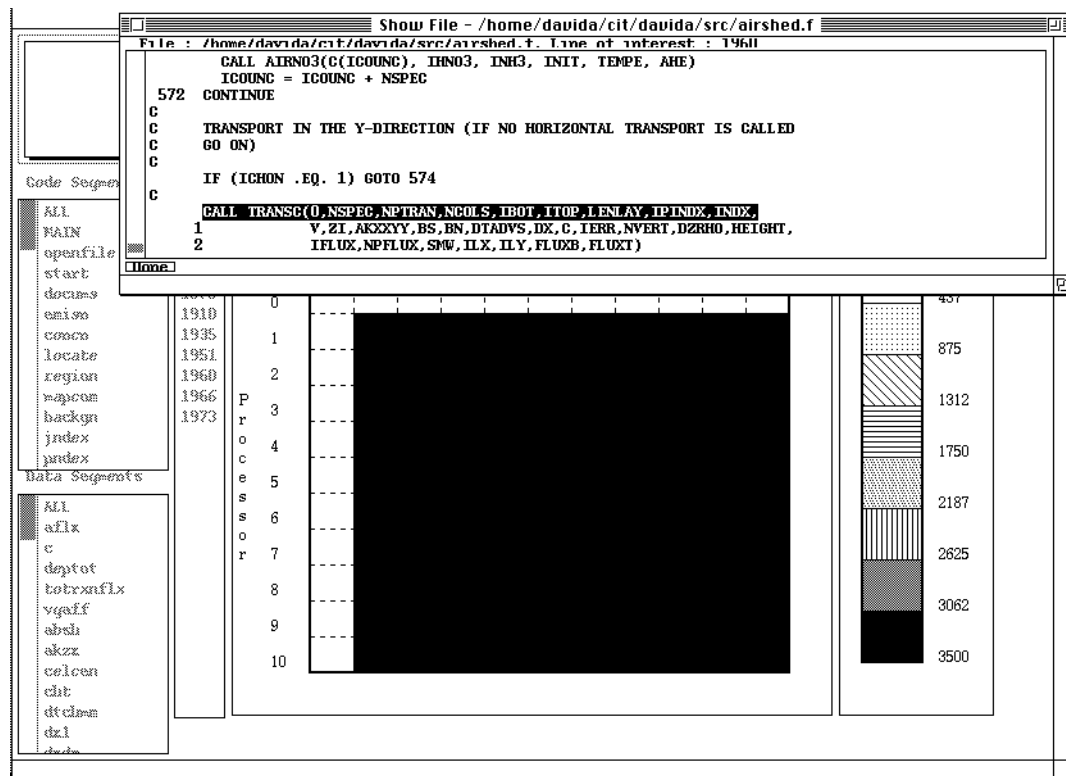


Figure 21 - Access to concentration vector during y transport

6 CONCLUSIONS AND FURTHER WORK

In this chapter we have discussed the problem of computing efficient data placements for large parallel programs running on distributed memory multiprocessors, and have described a tool called DCompose which allows a user to experimentally measure the amount of interprocessor traffic that results from a particular mapping. We illustrated DCompose using three real examples, two of them being large complex scientific codes written in FORTRAN.

One of the problems with DCompose is that it only gives information related to the communication within the parallel program. It does not predict the workload which results from a particular parallelisation strategy. Obviously, program performance is controlled by a suitable combination of a workload allocation strategy which spreads the work, and a data placement policy which minimises communication. An efficient program is one which obtains optimal settings for both

parameters. We are currently investigating some profiling techniques which will allow DCompose to report both statistics.

ACKNOWLEDGEMENTS

This work was conducted as part of a collaborative program between the CSIRO Divisions of Information Technology and Atmospheric Research, and the RMIT Department of Computer Systems Engineering. Thanks go to Martin Dix from the Division of Atmospheric Research and Martin Cope from the Victorian Environment Protection Authority for provision of the two models used in the experiments. The FFT code was originally written by Alan Harvey from the Department of Electrical Engineering at RMIT.

REFERENCES

Kendall Square Research, The KSR-1, Technical Summary.

Budnik, P.P. and Kuck, D.J. (1971) "The organization and use of parallel memories", IEEE Transactions on Computers, C-20, no 12, pp 1566-1569, 1971.

Crowther, W. et al (1985) "Performance measurements on a 128-node butterfly parallel processor", Proceedings of the 1985 International Conference on Parallel Processing, IEEE. Cat 85CH2140-2, pp 531-540.

Fox, G. et al, (1992) "The FORTRAN D Language Specification", Internal Report, NPAC, Syracuse University, NY 13244., January 1992

Thinking Machines Corporation, (1993) The Prism Reference Manual, 1993

NCube Corporation, (1992) A Family of High-Performance Massively Parallel Computers, nCube-2 Systems, Technical Overview., 1992

Maspar Computer Corporation, The MPPE Programming Environment

Cray Research, (1993) CRAY T3D Emulator User's Guide, Cray Research publication number SG-2500, 1993

Stunkel, C., Janssens, B. and Fuchs, W. K., (1991) "Collecting Address Traces from Parallel Computers", Proceedings of 24th Hawaii International Conference of System Sciences, Vol I, pp 373-383., 1991

- So, K., Darema, F., and George, Norton, V. A. and Pfister, G. E., (1988) "PSIMUL -- a system for parallel execution of parallel programs", Performance Evaluation of Supercomputers (J. Martin, ed.) pp 187-213, 1988.
- Kumar, B. and Davidson, E. S. (1978) "Performance evaluation of highly concurrent computers by deterministic simulation", Commun ACM, vol 21, pp 904-913, 1978.
- Axerold, T.S., Dubois, P. and Elgroth P. (1984) " A simulator for MIMD performance prediction - application to the S-1 MkIIa multiprocessor", Parallel Computing, vol 1, pp 237-274, 1984.
- Mitchell, C. L. and Flynn, M. J. (1988) "A workbench for computer architects", IEEE Design and Test, pp 19-29, Feb 1988.
- Goldschmidt, S. R. and Davis, H., (1990) "Tango Introduction and Tutorial", technical report CSL-TR-90-410, Computer Systems Laboratory, Stanford University, Jan 1990.
- Wiecek, C. A. (1982) "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution", Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems", pp 177-184, March 1982.
- Eggers, S. J. and Katz, R. H. (1988) "A characterization of sharing in parallel programs and its application to coherence protocol evaluation", Proceedings of 15th Annual International Symposium on Computer Architecture, Honolulu, HI, pp 373-383, May 1988.
- Borg, A., Kessler, R. E. and Wall, D. W. (1990) "Generation and Analysis of Very Long Address Traces", Proceedings of 17th Annual Symposium on Computer Architecture, Seattle, pp 270-279, May 1990.
- Strunkel, C. B. and Fuchs, W. K. (1989) "TRAPEDS: producing traces for multicomputers via execution driven simulation", Proceedings 1989 ACM SIGMETRICS Int. Conference on Measurement Modelling Computer Systems, Berkeley CA, pp 70-78, May 1989.
- Eggers, S. J., Keppel, D. R., Koldinger, E. J. and Levy, H. M. (1990) "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor", ACM SIGMETRICS Int. Conference on Measurement Modelling Computer Systems, pp 37-46, May 1990.

- Larus, J. R. (1990) "Abstract Execution: A Technique for Efficiently Tracing Programs", technical Report Computer Sciences Department, Univ of Wisconsin, Madison, WI, Feb 1990.
- Francis, R. S., Mathieson, I. D. and Pears, A. N. (1991) "Compiler-Integrated Multiprocessor Simulation", International Journal in Computer Simulation, Vol 1, Num 2, 1991, pp 169-188
- Agarwal, A., Sites, R. L. and Horowitz, M. (1986) "ATUM: A new technique for capturing address traces using microcode", Proceedings 13th Annual Symposium on Computer Architecture, Tokyo, pp 119-127, June 1986.
- Horst, R, and Metz, S. (1984) " New System Manages Hundreds of Transactions per Second", Electronics, pp 147-151, April 19, 1984.
- Rotstayn, L. and Dix, M., (1993) "Parallelization of a Spectral General Circulation Model", submitted for publication, CSIRO Division of Atmospheric research, Private Bag 1, Mordialloc, Vic 3095, Australia.
- McRae, G.J., Goodwin, W. R. and Seinfeld, J.H. (1982) "Development of a Second Generation Mathematical Model for Urban Air Pollution: I Model Formulation", Atmospheric Environment, 16, pp 679-696., 1982
- Goodwin, W.R., McRae, G.J. and Seinfeld, J.H. (1979) "An Objective Analysis Technique for Constructing Three-Dimensional Urban-Scale Wind Fields", Journal of Applied Meteorology, 19,98-108, 1979