18
A hierarchical classification of overheads in parallel programs

J. M. Bull
University of Manchester
Centre for Novel Computing, Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK.
Telephone: 0161-2756144. Fax: 0161-2756204.
email: markb@cs.man.ac.uk

Abstract

Overhead analysis is a powerful conceptual tool in understanding the performance behaviour of parallel programs. Identifying the source of overheads in the program requires the classification of overheads into different types. We present hierarchical classification schemes for both temporal and spatial overheads, which facilitate the analysis of parallel programs on a wide variety of parallel architectures by providing a flexible framework for describing performance behaviour. The issue of measuring the various classes of overhead is also addressed, with a view to improving the utility of performance analysis tools.

Keywords

Parallel programs, temporal overheads, spatial overheads, classification

1 INTRODUCTION

Performance tuning of programs is a key activity undertaken by users of parallel computers. An important part of this process consists of reducing the overheads associated with parallelism (as opposed to tuning the single processor performance of the parallel code). These overheads are of two types–**temporal** and **spatial**. Reducing temporal overheads assists the programmer in obtaining an acceptable execution time for the program in question, while reducing spatial overheads allows the program to make acceptable demands on memory. What is 'acceptable' will, of course, depend on the intended use of the program. At any stage in the process of performance tuning a program, it is useful for the programmer to know the source of the observed overheads. The purpose of this paper is to present classification schemes for both temporal and spatial overheads that encompass all sources. In contrast to previously proposed schemes, (see, for example, Burkhart and Millen (1989), Crovella and LeBlanc (1994), Eigenmann (1993), Tsuei and Vernon (1990) and Vrsalovic *et al.* (1988)) ours is hierarchical. The motivation for this is that the importance of any given class of overhead can vary considerably, depending on the program and architecture being considered. Thus any single-level classification will either omit some

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

important detail, or include unnecessary complexity for some program/architecture combinations. A hierarchical classification can be extended in areas of interest, or truncated in classes that are negligible, for the particular problem being considered. Thus it hoped we can provide a more flexible framework in which the programmer can reason about the performance of a program.

As discussed in Crovella and LeBlanc (1994), any classification scheme should have the following properties–

- **Completeness.** Any source of overhead should be classifiable within the scheme.
- **Orthogonality.** No source of overhead should appear in two different categories, unless one of the categories is a subset of the other.
- **Meaningfulness.** The classification should be meaningful, useful, and widely applicable in the context for which it is designed.

Crovella and LeBlanc acknowledge that their classification scheme is incomplete–we aim to produce a complete scheme. Other schemes such as those in Anderson and Lazowska (1990) and Martenosi *et al.* (1992) are also focussed on certain types of overhead. Unlike the first two properties, which can be objectively tested, meaningfulness is a subjective criterion. There is no one overhead classification scheme which can be said to be correct. In order to be able to assess the meaningfulness of our classification scheme, we must clearly state the context we have in mind:

> **Our primary purpose is to assist the programmer in choosing suitable code modifications to enhance the performance of a program on a given architecture.**

In particular the scheme is not designed to assist the computer architect in modifying the hardware so that a given program will run faster. Thus the categories that we choose will reflect abstract programming concepts at the highest levels, with hardware features only appearing near the leaves of the hierarchy, if at all. By designing our hierarchy in this way, architectural differences can be incorporated by relatively minor changes to the scheme, making our scheme applicable to a wide range of parallel machines. It is therefore intended that ours should be a program-oriented scheme, rather than a hardware-oriented scheme such as those described in Burkhart and Millen (1989), Tsuei and Vernon (1990), and to some extent in Crovella and LeBlanc (1994) where hardware resource contention is given as a primary category of overhead. In the process of performance tuning, a programmer can be greatly assisted by the use of performance analysis tools. We believe that a hierarchical classification scheme could be usefully employed by such a tool as a means of clearly explaining to the programmer the reasons for the temporal and spatial behaviour of the program.

Another property we could have added to our list is measurability. We might like our classes of overhead to be easily measurable on a given architecture. However, this property tends to conflict with all three properties listed above. In most real systems, not all overhead sources are easily measurable, measurement can result in counting some overheads twice, and what is easily measurable may not be meaningful. It is clear that measurability is the major criterion which drives the definition and classification of overheads for the Cedar system presented in Eigenmann (1993), and that this both restricts the meaningfulness and applicability of the classification. We will take the opposite view, in

the sense that we will avoid measurability as a criterion for defining classes of overheads. Nevertheless measurability is an important issue and we will return to it later.

2 TEMPORAL OVERHEADS

2.1 Definition

Before we can proceed to a classification scheme, we need to define what we mean by temporal overheads. Intuitively we would like the definition of temporal overheads to reflect the difference between the observed execution time of a program and the best execution time we could possibly expect if the entire program were parallelisable, and parallelism were free. Unfortunately, obtaining a tight lower bound on the execution time of a program is normally very difficult, since increasing the number of processors nearly always implies increasing the amount of fast memory (registers, vector registers or cache) available, resulting in the possibility of so-called superlinear speedup (see Helmbold and McDowell (1989)). The complexity of behaviour of fast memory (particularly cache) means that accurate simulation is the only way to obtain tight bounds on execution time. Thus we must accept that this ideal situation is in practice unattainable, and it makes sense to compare the observed execution time with a simple estimate of the 'best possible' execution time, bearing in mind that it will not in general be a lower bound. The simplest estimate we can give is to divide the execution time of a sequential version of the code by the number of processors. We define the temporal overhead on p processors O_p^T as

$$O_p^T = T_p - \frac{T_s}{p}, \tag{1}$$

where T_p is the observed execution time of the parallel code on p processors, and T_s is the execution time of the sequential code. Note that we will consider different classes of overhead as accounting separately for a certain amount of execution time and hence the classes will contribute additively to O_p^T. This means that they do **not** contribute multiplicatively to speedup or efficiency. This property also assists in verifying the completeness and orthogonality of a set of overhead measurements for a particular program, since the sum of the overheads over all classes should equal the total overhead directly measured by the above formulae. For the purposes of this study, we will assume that each processor runs a single thread of control.

Obviously there is the question of which sequential code is used to obtain T_s. This choice can be left to the programmer, as it will depend on what versions are readily available. On one hand, the programmer might wish to use a highly optimised implementation of a good sequential algorithm, in which case the overhead will include any algorithmic changes introduced to facilitate parallelism. On the other hand it may be more convenient to use a one processor version of the parallel code, but the programmer must be aware of the overheads that are being ignored by so doing. Frequently it will be some intermediate version between these extremes which is used.

It is tempting to divide overheads into those incurred by going from the best sequential version to a one processor parallel version and those incurred by going from the one

processor parallel version to the many processor parallel version. This presents some difficulties, however, since some useful classes of overhead such as synchronisation, scheduling and data access may be split between these two classes, whereas they are each better understood as a single class. In addition, there are cases where the parallel algorithm makes no sense for a single processor.

There are cases where T_s is unobtainable because a single processor does not have access to enough memory to run the problem, or else the execution time of the sequential version is unacceptably long. In these cases it may be necessary to extrapolate T_s from problem sizes which can be run, or else to use the approaches described in Crovella and LeBlanc (1994) and Hockney (1995), doing the entire overhead analysis on smaller problem sizes, and using modelling techniques to extrapolate the results to larger problem sizes. Neither approach is very satisfactory, however, as many overhead sources are difficult to model using simple functions of program parameters.

2.2 Classification Scheme

The main structure of the hierarchical classification scheme is shown in Figure 1. At the highest level, we identify four classes of temporal overhead–

T1 Information movement. Overheads associated with the movement of information in the system.
T2 Critical path. Overheads resulting from the critical computation path through the parallel program being longer than the ideal.
T3 Control of parallelism. Overheads resulting from additional code which does not contribute directly to the solution of the problem, but serves to manage the parallelism.
T4 Additional computation. Overheads resulting from changes to the sequential code in order to increase parallelism requiring more computation to be executed.

Information movement

Information movement can be split into two classes, depending on whether the information being moved is data associated with the program, or information about the state of the computation–

T1.1 Data accesses. Additional time spent making data accesses. Note that any time spent making data accesses which is overlapped with computation is not considered as an overhead.
T1.2 Synchronisation. Time spent in performing synchronisation operations.

The data access overhead can be negative, owing to the increase in the amount of fast memory available as the number of processors is increased. Data access overheads can themselves be split into classes, according the different possible paths between levels of the memory hierarchy, e.g. local memory to local cache, local memory to registers (bypassing cache), remote memory to local memory, disk to local memory. The number and nature of these classes will depend on the architecture concerned. Note that any changes to the code to increase parallelism may result in a different amount of time being spent in data accesses, thus contributing to this class. In many parallel programs on distributed

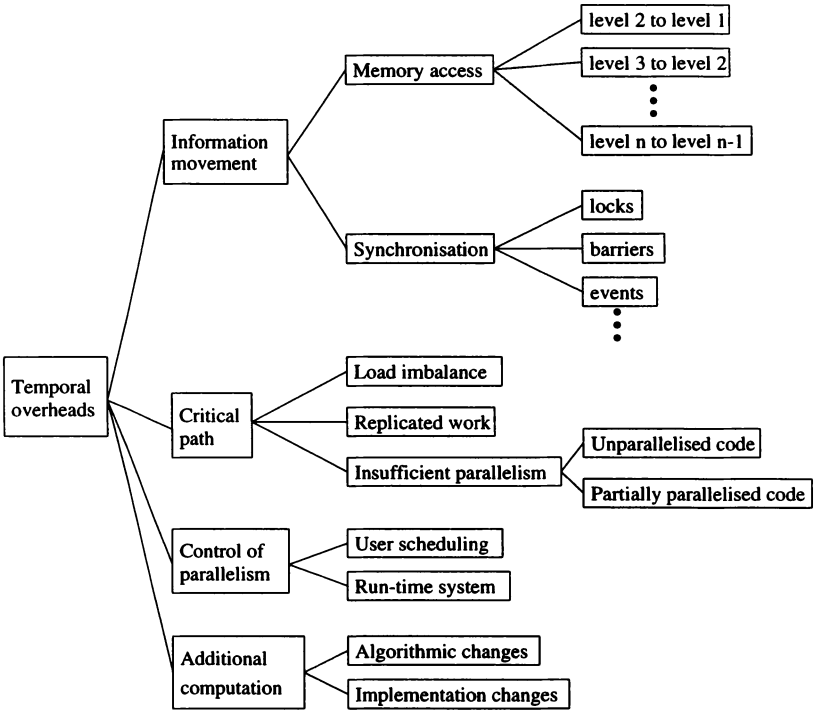

Figure 1 Classification of temporal overheads.

memory machines the greatest contribution to data access overheads will be from the remote memory to local memory path. It may prove useful to further subdivide this class– for example to distinguish between latency and bandwidth effects in the interconnection network, in message passing architectures to distinguish between read and writes, or in virtual shared memory architectures to distinguish between capacity misses and coherency misses. In contrast, data access overheads tend to be less significant on true shared memory architectures.

Synchronisation overheads may be subdivided according to the type of synchronisation construct being employed. The most commonly used are barriers, locks and events, though many other types do exist. It should be stressed that in this classification, synchronisation overheads **do not** include time spent waiting at synchronisation points. This time is accounted for by the class *T2* (critical path).

Critical path

Critical path overheads arise from imperfect parallelisation of the program. The usual effect of this is that processors will be idle, waiting at a synchronisation point for some period of time. Another possibility is that many processors may be executing the same code, since this may be less expensive that executing it on one processor, and then incur-

ring data access overheads in distributing the results. We divide the critical path overheads into three classes–

T2.1 Load imbalance. Time spent waiting at a synchronisation point, because, although there are sufficient parallel tasks, they are asymmetric in the time taken to execute them.
T2.2 Replicated work. Processors are not idle, but are occupied in replicating computation which is executed on other processors.
T2.3 Insufficient parallelism. Processors are idle because there is an insufficient number of parallel tasks available for execution at that stage of the program.

Insufficient parallelism has a variety of causes, giving rise to subdivisions of this class–

T2.3.1 Unparallelised code. Time spent waiting in sections of code where there is a single task, run on a single processor.
T2.3.2 Partially parallelised code. Time spent waiting in sections of code where there is more than one task, but not enough to keep all processors active.

Unparallelised code can occur because there exist data dependencies preventing its parallelisation, because it would more costly if it were parallelised, or simply because the programmer has not yet addressed the parallelisation of the code section. Partial parallelism can arise in a variety of parallel constructs, including fan-in/fan-out operations, critical sections, DOACROSS loops and wavefront loops. Note that distributed operating system code is a source of insufficient parallelism overhead, since access to shared resources such as page tables and I/O channels may be sequentialised.

Management of parallelism

This class of overheads arises because code which does not contribute to the generation of a solution, but is added to the sequential version to control and manage parallel tasks. The time spent in this activity may include executing instructions, data accesses and synchronisation, but it is not especially useful to sub-classify in this way. It is more meaningful for the programmer to divide it as follows–

T3.1 User scheduling. Time spent in user-written code that controls what parallel tasks are executed on which processor and when.
T3.2 Run-time system. Time spent in run-time system code.

Additional computation

In Section 2.1 we allowed the programmer to choose exactly how T_s is determined, and promised to include overheads resulting from changes to the sequential code in order to increase the available parallelism. So far we have accounted for additional data accesses arising from this source, but it remains to account for additional computation which may result. We sub-classify this overhead according to the level of specification at which the changes occur. As discussed in Gurd *et al.* (1993), levels of specification can be defined arbitrarily, so we are free to choose the most useful. We choose (without being rigorous) a specification level similar to the 'algorithm' level described in Gurd *et al.* (1993), which states what is to be computed in terms of simple arithmetic operations, basic mathematical

functions, indexed variables and simple set/logic concepts. Notice that such a description contains no information about how data is organised, nor any ordering of computation other than that required by the algorithm's data dependency structure.

We can classify additional computation according to whether it is caused by changes above or below this specification level–

T4.1 Algorithmic changes. Time spent in additional computation, resulting from changes at this specification level.

T4.2 Implementation changes. Time spent in additional computation, resulting from changes below this specification level.

Implementation changes include all the transformations that a restructuring compiler might make (loop interchange or fusion, for example), but also transformations that make use of some simple mathematical equivalences.

3 SPATIAL OVERHEADS

3.1 Definition

Before we can define spatial overheads, we must define more carefully what we mean by memory requirements. We choose the **maximum instantaneous memory usage**, since this the key quantity if there is a hard limit on the amount of memory available. It also has no temporal component, thus avoiding any overlap with temporal overheads. We would like spatial overheads to represent the difference between the memory requirements of a parallel code and those of its sequential counterpart. Our task is somewhat easier than for the case of temporal overheads, since the maximum instantaneous memory usage of the sequential program M_s is a reasonable lower bound on the maximum instantaneous memory usage of the parallel program M_p. We can therefore define the spatial overhead O_p^S as

$$O_p^S = M_p - M_s. \tag{2}$$

Once again there is the question of which version we use to measure M_s. If we wish to consider trade-offs between temporal and spatial overheads, then clearly we should use the same code used to measure T_s. Otherwise we allow the programmer to make their own choice, again bearing in mind that some overheads will not be accounted for if the one processor parallel version is used. Again, it is possible to split sequential to one-processor-parallel overheads into a separate class, but similar arguments to those used in the temporal case can be applied to justify avoiding this. In situations where M_s is not measurable, because the sequential code does not have access to enough memory, or will run for unacceptably long, it will be necessary to extrapolate M_s from smaller problem sizes. This is normally a much easier task than extrapolating T_s, as there is often a simple relationship between problem size and memory usage.

3.2 Classification scheme

Figure 2 shows the hierarchical classification scheme for spatial overheads. The primary

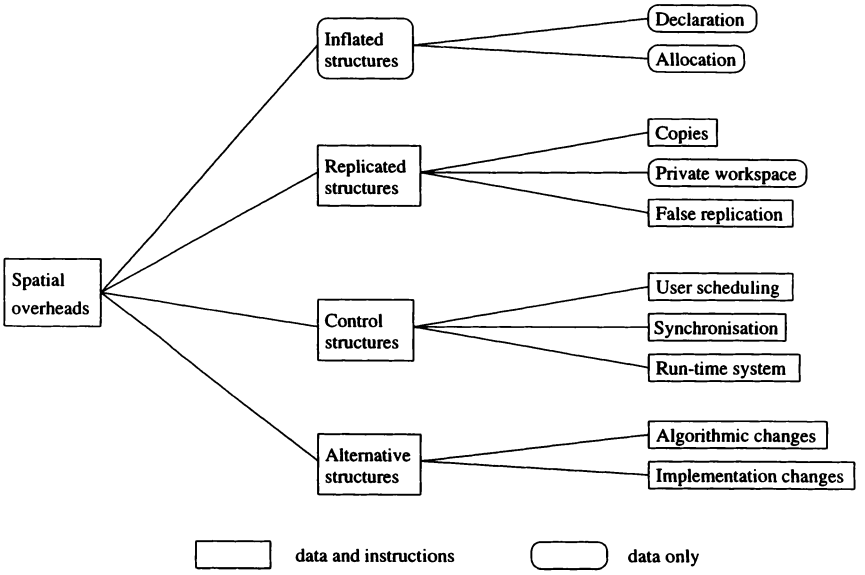

Figure 2 Classification of spatial overheads.

distinction in spatial overheads is between program data and instructions. However, since the subtree for instructions is a subset of the subtree for data, only the classification for data is shown, with those nodes which do not apply to instructions indicated.

At the next level we consider four main types of spatial overhead–

S1 *Inflated structures.* Additional memory required so that some dimension of a data structure is a convenient size.

S2 *Replicated structures.* Additional memory required because multiple versions of a data structure exist in the parallel code.

S3 *Control structures.* Additional memory required for data structures which are added to the program to enable the control and management of parallel tasks.

S4 *Alternative structures.* Additional memory resulting from changes to the sequential code in order to increase parallelism.

Inflated structures

Inflation of data structures only occurs in program data and not in instructions. Inflation takes a number of forms, but we can distinguish between two subclasses–

S1.1 *Declaration.* Inflation which occurs because the declared size of the data structure in the program has increased.

S1.2 Allocation. Inflation which occurs because memory allocation occurs in pages.

Declared inflation may occur because shared data structures are constrained to have certain sizes (for example, array dimensions must be powers of two on some systems). This effect is also seen on virtual shared memory machines, since a common technique to reduce false sharing is to pad the inner dimension of a data structure to the size of the coherency unit.

To understand allocated inflation, note that some space will be wasted on a single processor because the size of the data segment will be rounded up to a whole number of pages. On multiple processors, however, more memory may be wasted if each processor rounds up its segment to a whole number. This effect is compounded by any distinction between different types of data, as each type may have its own segment.

Replicated structures

Replication of data structures is often the most significant source of spatial overhead in parallel programs. We can classify replicated data by the way it is used in the program–

S2.1 Copies. Data space that is replicated and used to store copies of data stored elsewhere in memory.
S2.2 Private workspace. Data space that is replicated and used to store data values that are not replicated anywhere in the system.
S2.3 False replication. Declared data space that is replicated but never used.

Private workspace replication will occur on any architecture, and is the main source of replication on true shared memory machines. Copy replication is normally restricted to distributed memory, as its primary function is to avoid memory latency. Communication buffers are also a form of copy replication. Replication of instructions is limited to copy replication, and false replication. Note that false replication differs from allocated inflation in that in the former case the wasted memory corresponds to program variables whereas in the latter it does not. False replication occurs frequently in cache-only virtual shared memory systems, since space is allocated for a whole page even if the requesting processor only requires read access to a single word on that page. It may also occur in distributed memory systems when entire data structures are replicated because it is not known at compile time what subset of the structure will be accessed by a given processor.

Control structures

Control structures can be sub-classified according to the type of control–

S3.1 User scheduling. Data and instructions used by user-written code that controls what parallel tasks are executed on which processor and when.
S3.2 Synchronisation. Data and instructions associated with synchronisation constructs.
S3.3 Run-time system. Data and instructions associated with the run-time system.

Control structures are not frequently a major source of overhead, but they can sometimes be quite large; arrays of lock variables and arrays that describe task to processor mapping, for example.

Alternative structures
This class arises for precisely the same reasons as class *T4*–we must account for changes in memory requirements resulting from changes to the sequential code in order to increase the available parallelism. These overheads can be sub-classified in the same way as class *T4*–

S4.1 Algorithmic changes. Additional data and instructions resulting from changes at the algorithm specification level.
S4.2 Implementation changes. Additional data and instructions resulting from changes below this specification level.

It is possible that this class of overheads could be negative, if the changes result in lower storage costs but possibly longer (sequential) execution times.

4 MEASUREMENT

4.1 Temporal overheads

The most obvious requirement for measurement of temporal overheads is a direct consequence of the definition we have used: it is necessary to analyse the performance of both the parallel and sequential version of the code, and, to enable overhead localisation, to be able to correlate the corresponding regions of the code in the two versions. This is relatively easy to do at the subprogram level, but at a lower level may require explicit marking of basic code blocks.

Accurate measurement of temporal overheads imposes a number of requirements on hardware. At the very least each processor must have a high-resolution clock which can be read very cheaply. To record memory access times, hardware support is required for logging and timing misses in all the levels of the memory hierarchy, since most memory operations are too fine-grained to be measured in software.

Since both critical path and information movement overheads are often characterised by the CPU idling (or spin-waiting), it can be difficult to distinguish between the two. In a message-passing environment, the cost of communication between processors requires co-operation between the sending and receiving processes. The receiving process can time how long is spent in a blocking receive, but without knowing the time at which the message is sent, it cannot distinguish as to whether this is time is waiting for the message to arrive (memory access overhead), or waiting for the sending processor to finish some computation (load imbalance overhead). It is therefore necessary to have the send time available, either at run-time if statistics are being accumulated, or from any trace which is used for post run-time analysis. Some method of synchronising clocks between processors is also required.

Measuring synchronisation overheads poses some similar difficulties. A processor requesting a lock can tell how long it takes to acquire a lock, but it cannot divide this time into time spent waiting for the lock to be released by another processor (partially parallelised code overhead) and the time to transfer the lock (lock synchronisation overhead).

Control of parallelism, additional computation and replicated work overheads can only

be identified by their program context. This is easy if the code responsible is packaged into subprograms (a run-time library for example), but much more difficult if the code is inlined. Again, some marking of code blocks may be required. The ability to count instructions (either in hardware or software) can also be of assistance here.

4.2 Spatial overheads

Measuring spatial overheads is significantly more difficult than measuring temporal overheads, since determining the amount of memory used by a program, especially in a virtual memory system, is in general harder than measuring the program's execution time. This may account for the general lack of memory statistics available from performance analysis tools.

Distinguishing between data and instructions, or between private and shared data is straightforward, since they occupy different segments of memory. Partial information about spatial overheads can be obtained by examining the size of statically declared data structures and tracking dynamic memory allocation–this will normally give an accurate picture of control structure overheads, for example. This approach will be in error, however, if structures are declared (or allocated) but not used in their entirety. Automatically tracking use of memory is not straightforward, but the programmer will often have available some knowledge of array access patterns which allows a good estimate of spatial overheads to be made. In contrast to temporal overheads, rough estimates may be adequate to allow the programmer to make the appropriate modifications to the code to sufficiently reduce spatial overheads. Usage of memory at the page level can be monitored by the operating system, by keeping track of page faults, but deciding which structures are being accessed either requires the use of symbol table information, or the use of different memory segments for different data structures. The problem is especially difficult in a virtual shared memory system, where multiple copies of pages can exist. Obtaining information about usage of memory at a lower level (e.g. how many words on a page are actually accessed) can only be achieved via address tracing which may be impractical for large programs. Distinguishing between copy overheads and false replication in a virtual shared memory system is an example of such a problem.

5 CONCLUSIONS AND FUTURE WORK

We have presented hierarchical classification schemes for both temporal and spatial overheads in parallel programs. Since the classification has been motivated by meaningfulness and usefulness for a programmer, we hope that the schemes will form a useful framework in which overheads can be analysed on a wide variety of parallel architectures.

Prototypes of the temporal scheme have already been used in the development of parallel programs (see Egan *et al.* (1994) and Falcó Korn *et al.* (1995) for examples), and it is intended to utilise the full schemes in future porting exercises, incorporating them into a general methodology for parallel program development. It is also intended that this type of framework be extended to multi-threaded architectures, which we specifically excluded here.

REFERENCES

Anderson, T.E. and Lazowska, E.D. (1990) Quartz: a tool for tuning parallel program performance, in *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 115–125.

Burkhart, H. and Millen, R. (1989) Performance-measurement tools in a multiprocessor environment. *IEEE Trans. on Computers*, **38(5)**, 725–737.

Crovella, M.E. and LeBlanc, T.J. (1994) Parallel performance prediction using lost cycles analysis, in *Proceedings of Supercomputing '94*, IEEE Computer Society.

Egan G.K., Riley, G.D. and Bull, J.M. (1994) Parallelisation of the SDEM distinct element stress analysis code on the KSR-1. *Proceedings of ACM International Conf. on Supercomputing*, 85–92.

Eigenmann, R. (1993) Toward a methodology of optimizing programs for high performance computers. *Proceedings of ACM International Conference on Supercomputing*, 27–36.

Falcó Korn C., Bull J.M., Riley G.D. and Stansby P.K., (1995) Parallelisation of a three-dimensional shallow water estuary model on the KSR-1. *Scientific Programming*, **4(3)**, 155–170.

Gurd, J.R., Cooper M.D., Hedayat G.A., Nisbet, A., O'Boyle, M.F.P., Snelling D.F. and Böhm, A.P.W. (1993) A framework for experimental analysis of parallel computing. *University of Manchester Department of Computer Science Technical Report UMCS-93-2-3*, University of Manchester, UK.

Helmbold, D.P. and McDowell, C.E. (1989) Modeling speedup(n) greater than n, in *Proceedings of 1989 Int. Conf. on Parallel Processing*, Pennsylvania State Univ. Press, III-219–III-225.

Hockney, R.W. (1995) Computational similarity. To appear in *Concurrency: Practice and Experience.*

Martenosi, M., Gupta, A. and Anderson, T. (1992) MemSpy: analyzing memory system bottlenecks in programs, in *Proceedings of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1–12.

Tsuei, T.-F. and Vernon, M.K. (1990) Diagnosing parallel program speedup limitations using resource contention models, in *Proceedings of 1990 Int. Conf. on Parallel Processing*, Pennsylvania State Univ. Press, I-185–I-189.

Vrsalovic, D., Siewiorek, D.P., Segal, Z.Z. and Gehringer, E.F. (1988) Performance prediction and calibration for a class of multiprocessor systems. *IEEE Trans. on Computers* **37**, 1353–1365.

BIOGRAPHY

Mark Bull received the M.A. degree in Mathematics from the University of Cambridge, and the M.Sc. degree in Numerical Analysis from the University of Manchester. He is currently a Research Associate in the Centre for Novel Computing within the Department of Computer Science at the University of Manchester. His main research interests are in the design and implementation of parallel numerical algorithms and in parallel programming techniques and methodology, particularly for virtual shared memory architectures. Previously he worked as a researcher in atmospheric boundary layer simulation at the UK Meteorological Office.