Expert Programmer versus Parallelising Compiler: A comparative study of two approaches for distributed shared memory

M. F. P. O'Boyle and J. M. Bull

Centre for Novel Computing, Department of Computer Science,
University of Manchester, Oxford Road, Manchester M13 9PL, UK

Abstract. This paper critically examines current parallel programming practice and optimising compiler development. The general strategies employed by compiler and programmer to optimise a Fortran program are described, and then illustrated for a specific case by applying them to a well known scientific program, TRED2, using the KSR-1 as the target architecture. Extensive measurement is applied to the resulting versions of the program, which are compared with a version produced by a commercial optimising compiler, KAP. The compiler strategy significantly outperforms KAP, and does not fall far short of the performance achieved by the programmer. Following the experimental section each approach is critiqued by the other. Perceived flaws, advantages and common ground are outlined, with an eye to improving both schemes.

Keywords: Distributed Shared Memory, Expert Programmer, Parallelising Compiler, Performance Analysis, Linear Algebra.

1 Introduction

Obtaining high performance from parallel computers is the goal of both programmers and optimising compilers. Despite this obvious overlap in concern there has been little investigation into how each tries to achieve this goal, and whether either has anything to learn from the other. In this paper we begin such an investigation by way of an experiment, where the experimental method, evaluation and results are clearly defined. We also describe and compare the underlying mechanisms used by programmer and compiler writer in their attempt to parallelise a given program.

Historically, scientific programmers and compiler writers have worked in relative isolation for good reason; one is primarily concerned with reflecting a particular scientific model in a particular program while the other is concerned with implementing a language on a machine or family of machines. Both are concerned with preservation of meaning—one from model to program, the other from language to machine, but there is a difference in focus. Compiler writers are concerned with a range of general program constructs while programmers are more concerned with a single program (at a time) and its particular structure. Furthermore, programmers view a program in the light of the physical reality it is modelling, while compiler writers see it as a particular instance of all possible programs.

In parallel scientific computing, the emphasis on performance brings the perspectives of programmer and compiler writer closer together. On one hand there is a need for programmers to understand the features of a particular parallel architecture and of the compiler that tries to efficiently map their program to it. Conversely, compilers need to understand programs at a higher level of abstraction than just a syntax tree if they are going to perform well. In a sense the program is a contract or dialogue between programmer and compiler, where the former tries to explain to the latter what computation is to be done in a clear enough manner to enable efficient compilation to take place. Sometimes the need for clarity is so great that explicit language extensions are needed to tell the compiler not just what has to be done, but also how to do it.

The assumption that the programmer is the author of the program is by no means always true. Often a 'dusty deck' is to be ported to a new machine, where the programmer may have to apply a process of reverse engineering to try to determine exactly what the program is trying to do before they re-implement it. This situation is due to a lack of integration between programmers and compiler writers. When the dusty deck was original written, it may well have been written in a high-level, easy-to-understand manner. However, because the compilers of the time could not deliver the required performance, the program was hand transformed into some obscure and contorted form so as to best utilise the machine. In effect, the programmer took on the role of the compiler for most of the optimisation. Unfortunately much of this transformation is architecture dependent and has to be undone before it can be mapped to a new machine. However, the contorted version is usually all that remains, so inevitably when a new architecture/compiler is used, it often gives miserable results when compared to the previous system, hindering progress in parallel computing.

In this situation the dialogue between programmer and compiler (i.e. the program) is no longer understandable to either in its present form and has to be rewritten before progress be made. Precisely how this effort should be divided is the area of debate. In this paper we try to address this question by examining the world views of both groups. We take a representative of each and asking them to describe their general methodology, and how they would implement a particular scientific program.

The chosen program, TRED2, is a 100 line Fortran program from the eigenvalue solver package Eispack. It reduces a symmetric matrix to symmetric tri-diagonal form using accumulating orthogonal similarity transformations. The program contains imperfectly nested loops, IF statements and GOTOs, and, as such, is typical of many programs found in scientific computing. It is long enough so as not to be trivial and extensively studied (such as matrix multiplication), but short enough to allow analysis of the resulting program behaviour in a reasonable time.

A description of the machine used and its programming environment is given in Section 2. In Section 3 both the programmer and compiler writer[1] set out their approach to parallelisation. Sections 4 and 5 describe the application of the two approaches to the implementation of TRED2 on the KSR-1. The performance of the resulting programs is reported in in Section 6 and compared with the output of an existing commercial parallelising compiler. The relation between the experiments can be seen in Figure 1. As the programmer and compiler writer are both authors of this paper, there may be a tendency to overly agree, so, in Section 7 each approach is critiqued by the other. The advantages and disadvantages of the two approaches are outlined, and common ground between them identified, with an emphasis on how each scheme might be improved. We seek a more co-operative way forward for programmer and compiler, and outline further research in this area.

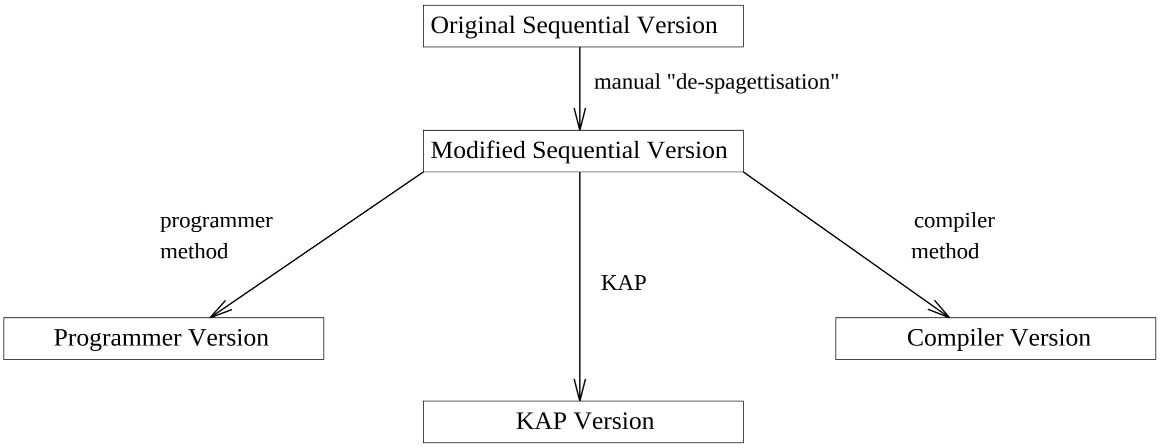

Fig. 1. Relationship between versions of TRED2

2 The Kendall Square Research KSR-1

2.1 Architecture

The target machine for the experiment is the 32-cell Kendall Square Research KSR-1 computer installed at the University of Manchester[2]. This is a distributed shared memory machine; it has a physically distributed memory but there is extensive hardware support for a single logical address space. Each

[1] The compiler strategy has recently been successfully implemented in a new compiler [2].
[2] Upgraded to a 64-cell system after this experiment was performed.

cell consists of a 20MHz processor with a peak 64-bit floating point performance of 40 Mflop/s and a 32 Mbyte main memory, which is also organised as cache memory. The cells are connected by a uni-directional slotted ring network with a bandwidth of 1 Gbyte/s. The machine has a Unix-compatible multi-user distributed operating system.

The memory system, called ALLCACHE, is a directory-based system which supports fully cache coherent shared virtual memory (SVM). Data movement is request driven; a memory read operation which cannot be satisfied by a processor's own memory generates a request which traverses the ring and returns a copy of the data item to the requesting processor; a memory write request which cannot be satisfied by a processor's own memory results in that processor obtaining exclusive ownership of the data item, and a message traverses the network invalidating all other copies of the item. The unit of data transfer in the system is a subpage which consists of 128 bytes (16 8-byte words). The operating system manages page migration and fault handling in units of 16 Kbyte.

The KSR-1 processor has a level 1 cache, known as the *subcache*. The subcache is 0.5 Mbyte in size, split equally between instructions and data. The data subcache is two-way set associative with a random replacement policy. The cache line of the data subcache is 64 bytes (half a subpage).

There is a 2 cycle pipeline from the subcache to registers. A request satisfied within the main cache of a cell results in the transfer of half a subpage to the subcache with a latency of 18 cycles (0.9 μs). A request satisfied remotely from the main cache of another cell results in the transfer of a whole subpage with a latency of around 150 clock cycles (7.5 μs). A request for data not currently cached in any cell's memory results in a traditional, high latency, page fault to disk.

In order for a thread to access data on a subpage, a copy of the page in which the subpage resides must be present in the cache of the processor on which the thread executes. If the page is not present, a page miss occurs and the operating system and ALLCACHE system combine to make a copy of the page present. If a new page causes an old page in the cache to be displaced, the old page is moved to the cache of another cell if possible. If no room can be found for the page in any cache, the page is displaced to disk. Moving a page to the cache of another cell is much cheaper than paging to disk.

2.2 Parallel Programming in Fortran

Parallel execution of programs is achieved by allowing a number of *threads* to participate in the execution. Each thread is a flow of control within a process. By default, the threads are scheduled by the operating system, and may time-share on a cell with other threads, or be re-scheduled from one cell to another during program execution. However the `allocate_cells` command can be used to reserve a number of cells for the execution of a program. Provided the number of threads required does not exceed the number of cells allocated, no time-sharing or movement of threads will occur.

Threads are managed in a Fortran program via a set of extensions to Fortran 77 consisting of compiler directives and library calls. For full details see [9]. The two most important directives are `parallel region` and `tile`. The `parallel region` directive encloses a section of code thus—

```
c*ksr* parallel region ([options])
         .
         .
        [section of code]
         .
         .
c*ksr* end parallel region
```

The enclosed code is executed by a number of threads, which is specified as one of the options to the directive. In addition it is possible to declare scalar variables as *private* variables: each thread will then have its own copy of these variables. All other variables are shared between threads. In order for threads to identify themselves, the integer function `ipr_mid()` is provided, which returns a thread ID in the range $0, \ldots, p-1$ when there are p threads executing the `parallel region`.

Synchronisation, by semaphores, between threads can be achieved at the most basic level by calls to `gspwt`(get subpage wait) and `rsp`(release subpage) which, respectively, attain and release atomic ownership status on a specified subpage. If a subpage is in atomic state, a thread will block on a `gspwt` call until atomic status is released by another thread using `rsp`. Library routines are available which use these constructs to implement mutual exclusion (*mutex*) locks, condition variables and barrier synchronisation. Barrier synchronisation also occurs at the beginning and end of a `parallel region`. This mechanism is used by the compiler method to achieve parallel execution.

The `parallel region`, together with the synchronisation mechanisms described above, is a very powerful parallel construct, but it requires careful management by the programmer and may necessitate significant code changes for problems requiring complicated scheduling of parallel work. Since loop-based parallelism is very common, a separate directive (`tile`) is supplied which applies only to perfect rectangular loop nests, including, of course, single loops. This reduces programmer effort, and the code changes required, for a class of common parallel constructs. The `tile` directive takes the following form:

```
c*ksr* tile (index_list, [options])
        .
        .
    [loop nest]
        .
        .
c*ksr* end tile
```

This divides the iteration space of the loop nest into a number of rectangular pieces (*tiles*). These tiles are then scheduled to be executed in parallel. The `index_list` allows the programmer to specify which iterators are to be tiled. The options allow specification of the number of threads to be used, and a choice of scheduling strategies. There are two strategies which are of interest in this experiment: `slice` and `mod`. The `slice` strategy divides the iteration space into p roughly equally sized tiles. The `mod` strategy divides the iteration space into more than p tiles (where possible), and schedules them on p threads in a modulo fashion. For either strategy the size of the tiles can be fixed by the programmer, or determined at run time. In the latter case the tile size will normally be chosen as a multiple of 16 to help avoid *false sharing* of subpages. False sharing is said to occur when two or more threads write to *different* words on the *same* subpage, causing unnecessary data movement. The options also allow scalar variables to be declared as *private* variables, or as *reduction* variables. In the latter case results are accumulated in local copies of the variable, and code is generated which reduces these to a single variable at the end of the tiled loop.

A further mechanism for avoiding false sharing is the `subpage` directive, which forces a scalar variable, or an element of an array, to be aligned on a subpage boundary. In order to minimise data movement, it is sometimes advantageous to force several different tiled loop nests which have common iterators to be tiled with the same strategy, so that any value of the iterator is always assigned to the same thread. This facility is provided by the `affinity region` directive.

3 Approach

3.1 Compiler

The strategy employed by the compilation approach is intended to be generic for all distributed memory architectures. In particular, the strategy was not determined by experiment on the KSR-1, but by considering common overheads such as synchronisation and communication. This will be in marked contrast to the method employed by a programmer.

A linear algebraic representation of the program is used which allows the easy formulation of transformations to optimise program behaviour. The compilation strategy can be broken into two parts: (i) transform the program to expose maximum parallelism, and (ii) map this parallel work so as to utilise machine parallelism and minimise overheads. To expose parallelism, it is assumed that a parallelisation tool such as Parafrase, Parascope or tiny will be used. For the purpose of this experiment a method based upon accurate data dependence analysis, loop distribution and scalar expansion was employed. The method is used in automatic vectorisation as described in [15].

Once parallelism is exposed, the next stage is to map it to the parallel machine. This can be done in a computation or data oriented manner i.e. it is possible to partition parallel loops or partition arrays, scheduling them across the processors. The approach taken here is data oriented; data layout is determined in an attempt to be globally optimum. The layout determines the work to be performed on each processor and the amount of communication.

The compiler strategy is based upon a method described in [13] and was applied manually to the program. The cost model consists of three cost factors: load imbalance, inter-processor communication and synchronisation. The compiler optimisation problem is to exploit sufficient parallelism while trying to minimise these costs. The most important feature is that there is a separation of concerns. The sequence of transformations the compiler undertakes reflects the perceived relative importance of hardware costs. Optimising for one cost followed by another may miss an optimum that would be found considering them both jointly, but it makes compiler decisions tractable and avoids backtracking. This compiler strategy will be modified in the light of this experiment. It is hoped that an important product of this experiment will be to show to what extent generic compilation techniques are useful and what modifications are needed to successively utilise shared virtual memory in general and the KSR-1 in particular.

Model of Computation

The model of computation used is single process multiple data (SPMD) where each thread executes the same program, but operates on different sections of the distributed array. One thread is scheduled to each processor. A local-write rule is used where only data that is local to a processor/thread is written to. This is very similar to the owner-compute rule but is slightly more relaxed in that it allows exploitation of reduction parallelism.

```
DO I = 1,N
   A = A +B(I)
ENDDO
```

Fig. 2. Simple reduction example

To illustrate this point, consider the loop in Figure 2. If an owner-compute rule were used on this program then the processor owning element `A` would access all of `B` and perform the calculation. With the local-write rule, each processor may calculate a partial sum which is then read by the processor owning `B`. Only local writes have taken place, but the parallelism associated with reduction has been utilised.

An essential characteristic of this approach is data affinity. Knowing where data resides allows the scheduling of work so as to minimise communication. The local-write rule is an example of this—work that writes to a particular piece of data is scheduled to the corresponding processor. Knowledge of the layout of data combined with an affinity approach can also help reduce inter-processor communication and the number of synchronisation points.

Finally a compiler strategy must be scalable if it is to be generally applicable. Any compiler introduced temporaries must fit in the appropriate part of the memory hierarchy for all data and problem sizes. Clearly, having a copy of all array data in every processor is not a scalable approach. The general strategy is as follows:

Pre-Process

A program may contain GOTOs and induction variables which make analysis difficult. At this stage control flow normalisation and induction variable recognition are performed so as to aid the parallelisation and subsequent stages. If an array access is a simple (linear) function of a DO loop then there exists a large body of techniques to analyse and transform the enclosing loop. Sometimes, an array access is a linear function of a loop but it is hidden by an aliasing variable. Recognising such aliasing of iterators is called induction variable recognition. and, in itself, does not improve program performance, but does enable subsequent transformations.

Parallelise

Only loop parallelism is investigated. Transformations for parallelism are based upon accurate dependence information and loop distribution. An enormous amount of research has been undertaken in this area [19],[11],[18],[20],[16], and there still remains much to do. No original methodology is claimed, the approach taken for the purposes of the experiment was based on traditional vectorising compiler techniques. After the program has been 'cleaned up' by removing GOTOs, loop distribution was aggressively used to expose parallelism. Distributing a loop containing two statements results in two new separate loops each with one statement. Loosely speaking, reducing the number of statements in a loop increases the probability that it may be executed in parallel. The main restriction on parallelisation is data dependence. While flow (or true) dependence cannot be removed, scalar expansion and statement re-ordering can be used to remove some output and anti-dependences. which are due to storage re-use rather than any flow of information. At this stage only uncovering of parallelism is addressed; there is no regard to how much overhead this may lead to, as this is the concern of later stages.

Align

In the compilation model used, data is eventually allocated to a processor. Alignment is concerned with the relative orientation of arrays with respect to a common index space and with maximising the number of references that are local. If two arrays are said to be aligned on an index, then no matter what the form of the subsequent data partition, corresponding elements of each array will be mapped to the same processor. For example consider the arrays `A(10,10)`, `B(10,10)`. If they are aligned, then elements `A(1,1)` and `B(1,1)` will mapped to the same processor as will elements `A(X,Y)`, `B(X,Y)` for any values of `X` and `Y` in the range 1 to 10. If a statement of the form `A(I,J) = B(I,J) +..` exists in the program then this alignment will imply that no non-local communication of `B` will take place. The processor that is writing to `A(I,J)` will have `A(I,J)` local to it due to the local-write implementation, and as `B(I,J)` will be mapped to the same processor as `A(I,J)`, it too will be local. Therefore, at compile time, we know there will be no communication necessary. If however, the statement were of the form `A(I,J) = B(J,I) +..` then communication would take place given the above alignment. If, however, `B` were stored *transposed* relative to `A` then again no communication would be needed. Determining the best relative storage to minimise communication throughout the program is the concern of this stage.

Partition

To determine the overall data partition, parallelism and overheads are examined, each of which will suggest a particular data partition for a program section. In [13] four techniques are described which suggest a particular data partitioning to minimise a particular cost The relative significance of each cost will determine a particular partitioning. In general there is a trade-off—each cost will suggest a conflicting data layout. The availability of certain post partition transformations may also have an effect on partitioning and must be considered where appropriate. The basic approach is based on concentrating any cost analysis on the 'most significant' parts of the program and determining when a cost or transformation will have any significant impact. Given all this local information for 'significant region' a global decision is made.

Synchronisation

Synchronisation can be a significant overhead in parallel programs. It should be proportional to the amount of communication, but naive analysis can lead to the insertion of excess synchronisation points. This is often the case when barrier synchronisation is used to ensure that data dependence is honoured. It is possible to exploit the SPMD model of computation used in this strategy to reduce the amount of synchronisation. By detecting those dependences that are entirely local, synchronisation points can be removed.

Mapping

After the data layout has been determined, it is necessary to generate the parallel program based on this decision. As this is an SPMD implementation, there is only one fork which occurs at the beginning of the parallel region. To ensure a local write rule, only one thread ever writes to any particular array element. Essentially, the range of the iterators in a thread is limited by the bounds of the array elements written to. If there is any sequential work to perform, one thread will execute, while the others wait at a later synchronisation point. This approach prevents the high cost of forks and data movement, but is less dynamic. Whilst data cannot be *statically allocated* to processors on the KSR-1, threads can be. If only one thread writes to a piece of data that data will be allocated at the site of that thread and will remain there as no other thread will ever wish to have write ownership of it. Hence it is possible to exploit data locality based on alignment and layout on a COMA architecture.

Non-local Data Re-use

Once a non-local array element has been referenced it is desirable to store it locally if it is to be referenced again. Such a scheme exploits temporal locality by detecting data re-use in the program. Much research in compiler restructuring and architecture design has been directed at providing such a facility. Essentially by examining array references, it is possible to determine if data access is invariant of an iterator, or combination of iterators. For a fuller description of this method see [14]. In this paper, the data re-use method is extended by considering only those array accesses that are known to be non-local. Perfectly aligned data will be local and need not be considered.

Each of the above stages are applied in sequence to give a parallel FORTRAN program, as shown in Figure 3.

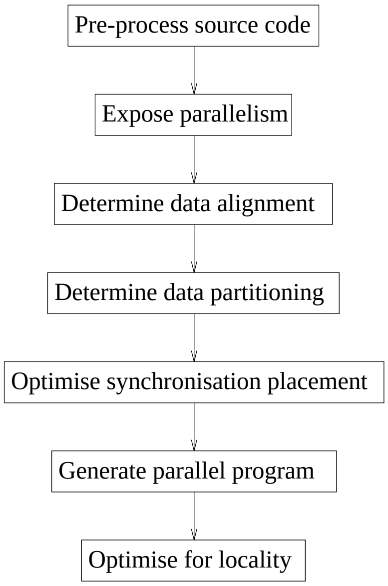

Fig. 3. Flow diagram for compiler approach

3.2 Programmer

The objective in this experiment, in common with much scientific parallel programming, is to minimise the parallel execution time for an existing piece of sequential code. This minimisation exercise is, in general, constrained by the amount of programmer effort available. In this study, an attempt is made to produce an efficient parallel implementation whilst keeping code changes, and therefore programmer effort, to a minimum. The methodology described here has been developed during approximately eighteen months of programming experience on the KSR-1. The results of applying this methodology to large scale applications are reported in [6] and [7]. Although in some respects, notably in certain cures for some performance bugs, the approach is machine specific, much of it will be applicable to other shared virtual memory architectures. It is, however, notably different from the approach that is required on traditional, message passing, distributed memory architectures.

The approach taken here is one of incremental parallelisation, and relies heavily on the use of profiling and program instrumentation. The programming model adopted is essentially a shared memory model, where by default all data is globally accessible. All code is executed by a single thread, (the *master* thread), unless the code is enclosed in a parallel construct, such as a `parallel region` or `tile` directive (see Section 2.2). The master thread controls the synchronisation necessary at the beginning and end of each parallel construct.

The approach can be thought of as a progressive reduction in parallel overheads. For a program running on p processors, the parallel overhead O_p is defined as the difference between the actual execution time and that predicted by a naive Amdahl's Law model (where all code is assumed to be parallel), that is

$$O_p = T_p - \frac{T_s}{p},$$

where T_s is the execution time for the sequential version, and T_p is the execution time for the parallel version on p processors. It is helpful to classify the overheads into a number of classes according to their source. There seems to be no general consensus as to how best to define this classification: see [5] and [4] for two examples of classification schemes. The classes used here are intended to be as program oriented as possible, and therefore largely hardware independent, because their purpose is to help identify the program constructs which are causing poor performance, rather than hardware bottlenecks. These classes are

- Unparallelised code.
- Load imbalance in parallelised sections.
- Memory access costs. These can be subdivided into
  - Remote memory accesses (movement of data between cells).
  - Local memory accesses (movement of data between levels of memory hierarchy within a cell).

  Note that memory access costs can either increase or decrease when the code is parallelised, so it is possible for negative overheads to occur.
- Synchronisation. This is the cost of barriers and locks, and of any idle time caused by contention for locks.
- Scheduling. This is the cost of any computation necessary to determine at run time which tasks are to be scheduled on which processor.

At any stage of the parallelisation process, the strategy is to measure the contribution to the overhead in each of the above classes for a given problem size, and to identify the sections of source code responsible for these contributions. The programmer then attempts to reduce the most significant overheads by changing the parallel code. This process is then repeated until the programmer runs out of time or new ideas. Sometimes a point may be reached where no further progress seems possible. The programmer may then decide to backtrack to a certain point in the process, and try again in a different direction.

Naturally the programmer must take care that any transformation of the program preserves the program semantics. This is normally achieved by checking the results generated by a sample input data set. This is not entirely foolproof, however, as although incorrect results indicate an incorrect transformation, the converse is not true. A much more difficult issue is whether a code transformation can affect the numerical stability of an algorithm.

Initially, of course, all the overhead will be due to unparallelised code. Thus the first few steps of the parallelisation process will consist of exposing and exploiting parallelism in appropriate sections of the sequential code. To aid this process, the programmer must first obtain an easily readable form of the code. The most important factor here is a simple control flow structure, with as many GOTO statements replaced by conditionals as possible. Non-standard language features such as the DO WHILE statement are helpful here. Clear visual layout (indentation of loops and conditionals) and removal of induction variables are less important, but may be of assistance. Much of this work (except the removal of induction variables) can be automated using commercially available tools. Next the programmer must identify the important parts of the code; that is, where most of the execution time is spent. In simple cases this can be done by inspection, but, in general, it is necessary to obtain a profile by code instrumentation. The instrumentation can be in the form of a profiling tool, such as the standard Unix utility GPROF, or by hand-inserted calls to a timer routines. Usually the latter is more satisfactory, if more laborious. With profiling information in hand, the programmer sets out to expose parallelism in the most expensive sections of the code. In cases where the code consists mainly of loops, an automatic parallelisation tool may be of some assistance. Current commercially available tools, however, can only apply a limited amount of restructuring, and for more complex loops, or where no loops are explicitly present, this process must be done by hand. Fortunately, the programmer often has a very powerful tool at their disposal—they can reason about the problem at a higher level of abstraction, in terms of the algorithm which the code implements. Parallelism may be obvious at this level even if it is deeply obscured at the code level, because at the algorithm level there is a more abstract, higher level, description of tasks and data objects. It is often easier to identify independent tasks at the algorithm level, rather than sets of independent arithmetic operations at the code level. If the algorithmic structure is not available to the programmer, thanks to inadequate or unavailable documentation, then provided something is known about the abstractions employed at the algorithm level, reverse engineering of the algorithm may be possible.

Having exposed parallelism in a particular section of code, the programmer may be presented with a number of choices as to how to exploit it. Some of these options may be immediately rejected if it is obvious that they will incur unacceptably high overheads. Having done this, the choice will normally be determined by the amount of effort involved in terms of changes to the code. Further analysis of overheads will decide on whether more effort on this section of the program might be beneficial.

Once unparallelised code is no longer the dominant source of overhead, the programmer can turn their attention to reducing overheads in the parallel sections of code. The part of the parallelisation strategy which is causing the most overhead is targeted for alteration. The programmer now has to find an alternative strategy that will reduce the total overheads. This is largely a matter of reasoning about the program (possibly at the algorithm level) using a rough cost model for overheads, combined with a certain amount of inspiration. While it is impossible to give a general 'recipe' for parallel programming, there are certain techniques, some generally applicable, others architecture specific, which often prove useful in reducing the various sources of overhead:

– **Load imbalance**
  Load imbalance may be either due to processors having different amounts of arithmetic operations to do, or to processors spending different amounts of time in memory access operations. In the former case, block interleaving of loop iterators is a common solution, provided that the number of computations is a suitably smooth function of the iterator. If it is not smooth, then it may be necessary to devise an explicit scheduling strategy. Most run time self-scheduling policies, the simplest of which is first-come-first-served, are rarely a success in scientific applications, since they tend to destroy data locality.
  There may be a number of causes for imbalance in memory accesses. If the pattern of imbalance changes from one instance of the loop to the next then it is often a symptom of false sharing; see later for possible cures. If the pattern is repeated from one instance of the loop to the next then it may be a genuine requirement of the algorithm, in which case the techniques used to balance computations may be applied, though it should be noted that there will be a tradeoff between remote access and load imbalance, and the optimum solution may not be load balanced.
  Another possibility is that the imbalance is a result of cache interference misses being much more severe for some values of the iterator than for others. It may be possible to reorganise data structures to avoid this.
– **Remote Memory Accesses**
  If significant numbers of remote memory accesses are occurring, then it will be necessary to determine whether they are genuinely required by the parallel algorithm, or whether they are a result of false sharing. In the KSR-1 memory system, false sharing can occur at both the subpage level and the page level. False sharing of subpages can be overcome by padding and aligning of arrays (or scalars) to ensure that no two threads access different words on the same subpage. It may also help to avoid tiling over the first index of arrays, if possible, or to use tile sizes which are a multiple of the subpage size. In order for a thread to access a subpage, the page on which the subpage lies must be resident on the requesting cell. False sharing at the page level can results if many threads access subpages that reside on the same 16K page. Since all the threads accessing a page must have space allocated for that page, capacity misses may cause pages to be displaced to other cells memories, with a resulting cost when they are next required. The solution to this problem is to reorganise the data structures or the parallelisation strategy such that, as far as possible, each thread accesses data which occupies a contiguous region of the virtual address space.
  If genuine communication is a significant source of overhead, then it will be necessary to reorder computations to improve locality. For example, if the parallelisation strategy consists of a spatial decomposition of a regular grid, with neighbour-wise communication, then choosing sub-domains with a minimum boundary length may reduce the number of remote data accesses. A frequent source of communication on SVM systems is sequential sections of code which may account for a small proportion of the run time in the sequential version, but cause a significant number of remote accesses in the parallel version because data is gathered by the master thread and subsequently scattered again.
– **Local Memory Accesses**
  Local memory access overheads are normally a result of the parallelisation strategy causing a loss of locality in the subcache. The solution is to reorder the computations so as to maximise the re-use of data items that are loaded into subcache. In scientific codes with large arrays, ensuring stride-one memory accesses through the arrays is frequently of significant benefit.
– **Synchronisation**
  If synchronisation costs dominate, then the programmer should check whether all synchronisation

points are indeed necessary to ensure that the program generates correct results. The KSR Fortran `tile` directive implies a barrier synchronisation at the beginning and end of the tiled loop nest. These barriers may not always be necessary, particularly if the tiled loop is contained in an outer loop. Loop fusion can be used to reduce the number of synchronisation points, as can the `parallel region` directive combined with calls to barrier routines.
If locks are used, then overheads can result both from the actual calls to lock routines, and from any contention for the locks. If contention is significant, then it pays to reduce the length of any critical sections as much as possible. If on the other hand the overhead arises largely from the calls themselves, it may be beneficial to reduce the number of locks, even if this means increasing the length of the critical section. Note that on the KSR locks are implemented via atomic status bits on subpages. Accessing these directly is significantly cheaper than using the mutex lock routines.
- **Scheduling**
Although run time scheduling is convenient, as it allows the number of threads to be varied without recompiling, it can be costly. On the KSR-1, scheduling can contribute at least as much to the cost of the `tile` directive as does barrier synchronisation. This can be minimised by supplying as much information as possible to the run-time system, or even by doing the scheduling in user code, using the `parallel region` directive.

Tradeoff situations are very common—reducing one source of overhead frequently results in the increase of another. Furthermore, the sources of overhead may not be localised in one section of the code—changes to one section may affect the overheads in another. In such situations it will be necessary to run experiments to determine which parallel strategy is the best. The run-analyse-modify process can be iterated, including possible backtracking, until either performance is satisfactory or other constraints force it to come to an end, as illustrated in Figure 4.

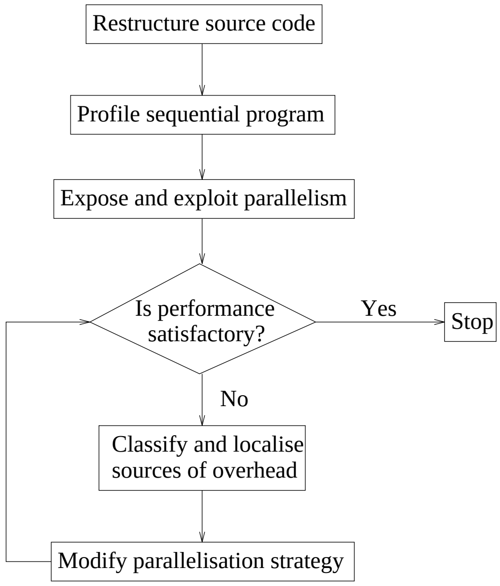

Fig. 4. Flow diagram for programmer approach

4 Compiler Method

4.1 Pre-Processing and Parallelisation

A considerable amount of preprocessing must take place before TRED2 can be sensibly parallelised. Firstly control-flow normalisation is attempted where the GOTOs are replaced by conditionals, leaving the structure of the code as intact as possible. Secondly, induction variable substitution takes place, which is then followed by forward substitution of scalar temporaries which hold array values. For example, consider the program in Figure 5a, where `I` and `L` are induction variables. All occurrences of `I` are replaced by `N+2-II` and all occurrences of `L` are replaced by `N+1-II` to give Figure 5b. A full listing of the code after this process has been applied is given in the Appendix.

Extracting parallelism by loop distribution is by no means trivial; whilst it is relatively easy to see, in columns Figures 5a and 5b, that the `K` loop will provide some parallelism and its two statements do not interfere, the `II` and `J` loops look sequential. This is, in fact, the conclusion of Gupta [8] when using the Parafrase environment. However, if certain transformations are applied, the program in Figure 5c can be obtained where the distributed `J` loop is also parallel.

To get to this form requires careful analysis of the order of access to each of the arrays. In Figure 5a statements 2 and 3, within the `K` loop, do not refer to each other and can thus be easily loop distributed to give Figure 5b. Each of the separate references to `Z` in the code fragment of Figure 5b refer to a different region of the array. `Z(J,I)` in the first line accesses the strictly upper triangular region of the `Z` array. On the next line the read of `Z(J,J)` is simply the diagonal, while the remaining read of `Z(K,J)` is to the strictly lower triangular region of the `Z` array. None of these accesses refer to the same region of the `Z` array and hence there is no order placed on the evaluation of the statements due to the data dependences arising from the access to `Z`.

There is no cross-iteration data dependence with respect to `G`, but there is a flow dependence from statement 1 to 2, 2 to 3 and 3 to 5, therefore the lexicographic order $1 \prec 2 \prec 3 \prec 5$ must be maintained. `G` must be scalar expanded before loop distribution can take place to allow each iteration of the distributed `J` loops to refer to the correct value of `G`.

The only remaining variable preventing distribution of the `J` loop is the variable `E`. There is cross-iteration anti-dependence from statement 2 to 4. The values written in statement 4 are read by statement 2 on the next iteration of the `J` loop. If loop distribution were performed then statement 4 would have to precede statement 2 (see [15]). The output dependence from 4 to 5 implies that statement 4 must precede 5 if `J` were distributed. This gives two partial orders on the statements. `G` requires $1 \prec 2 \prec 3 \prec 5$ and `E` requires $4 \prec 2 \prec 5$. Fortunately these orders are not conflicting giving a legal order (amongst others) of $1, 4, 2, 3, 5$. The final loop distributed version is shown in Figure 5c where it is easy to determine that each of the loops is parallel. Any one of these stages in itself is not very complex. However, the correct ordering of analysis and transformation is difficult and often defeats parallelising compilers.

4.2 Alignment

Alignment has direct global impact; the location of a value must be consistent throughout its potential multiple usages in a program. In this experiment, a restricted method is used whereby only index reordering is considered. Alignment is NP-complete [10], so to reduce the complexity of deciding the overall alignment, only the *most significant regions* of the program are examined. These are the regions where the most communication and computation is expected and correspond to the deepest loop nests and largest arrays. In the presence of conditionals static estimates based on control-flow information are used to determine the significant regions. A global alignment graph is created where an arc between two indices indicates that it is desirable that the indices be aligned. After applying a variant of the index domain alignment algorithm [10], the following global alignment, using a linear algebraic representation, was determined for the referenced arrays `E`, `Z`, `D` and `GG`.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{T(E)} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{T(Z)} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{T(D)} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{T(GG)} \tag{1}$$

This indicates that the one dimensional arrays `E` and `D` should both be aligned with the first index of `Z` while `GG` should be aligned with the second.

4.3 Global Data Partitioning

Partitioning analysis is applied to all the significant statements, where each statement will suggest a particular data partition. The most popular partition(s) are stored in a set. *Para* is the set of the most popular partitions if the parallelism analysis is applied, *Align* is the corresponding set based on alignment, *Load* that on load balance and *Region* that on the region of access. The region of access analysis tries to determine the best data partition based on the assumption that data re-use transformations will take place after partitioning.

```
 DO II = 2,N
   .
   .
   I = N+2-II
   L = I-1
   .
   .
   DO J = 1,L
1:   Z(J,I) = D(J)
2:   G = E(J)+Z(J,J)*D(J)
     DO K = J+1,L
3:     G=G+Z(K,J)*D(K)
4:     E(K) = E(K)+Z(K,J)*D(J)
     ENDDO
5:   E(J) = G
   ENDDO
   .
 ENDDO
```

(a) Original source code.

```
 DO II = 2,N
   .
   .
   DO J = 1,N+1-II
1:   Z(J,N+2-II) = D(J)
2:   GG(J) = E(J)+Z(J,J)*D(J)
     DO K = J+1,N+1-II
3:     GG(J)=GG(J)+Z(K,J)*D(K)
     ENDDO
     DO K = J+1,N+1-II
4:     E(K) = E(K)+Z(K,J)*D(J)
     ENDDO
5:   E(J) = GG(J)
   ENDDO
6: G= GG(N+1-II)
   .
 ENDDO
```

(b) After inner loop distribution.

```
 DO II = 2,N
   .
   .
   DO J = 1,N+1-I
1:   Z(J,N+2-II) = D(J)
   ENDDO
   DO J = 1,N+1-II
     DO K = J+1,N+1-II
4:     E(K) = E(K)+Z(K,J)*D(J)
     ENDDO
   ENDDO
   DO J = 1,N+1-II
2:   GG(J) = E(J)+Z(J,J)*D(J)
   ENDDO
   DO J = 1,N+1-II
     DO K = J+1,N+1-II
3:     GG(J)=GG(J)+Z(K,J)*D(K)
     ENDDO
   ENDDO
   DO J = 1,N+1-II
5:   E(J) = GG(J)
   ENDDO
6:  G= GG(N+1-II)
   .
 ENDDO
```

(c) Fully loop distributed.

Fig. 5. Parallelisation

Stat.	*Para*	*Load*	*Align*	*Region*
a	1,2,1∧2	∅	1	1
b	1,2,1∧2	∅	2	2
c	1,2,1∧2	∅	1	1∧2
d	1,2,1∧2	∅	2	2
e	1,2,1∧2	∅	1,2,1 ∧2	1∧2
Indices				
1	5	0	3	1
2	5	0	3	2
1 ∧ 2	5	0	1	2

Table 1. Partition decisions for each analysis

Table 1 shows the results of applying each form of analysis for the TRED2 code based on the significant regions a,b,c,d,e (see code in Appendix). Here 1 denotes partitioning by the first index of `Z`, 2 by the second index, 1∧2 by both indices and ∅ by neither. As far as parallelism is concerned, all partitions were equally acceptable. The load balance analysis found no suitable iterators, while alignment preferred either the first index or the second index but not both. Finally the region of access preferred index 2 or indices 1 and 2. Applying the trade-off algorithm [13] to the TRED2 example gives the result that data should be partitioned along the second index. None of the analyses gave a unique choice, implying that certain sections have conflicting requirements. Other programs will have different behaviour. Because no one data partition is clearly superior throughout the whole program, it seems likely that the partitioning decision in this program is not crucial, and other factors such as later optimisations may dominate. Once the global partition has been decided it is necessary to generate the code that each thread will execute. The remainder of this section describes the code generation and subsequent post-partitioning transformations.

4.4 Synchronisation

Synchronisation between threads was achieved using a barrier synchronisation method. A call to a subroutine `Lock` is inserted at each synchronisation point, in which each thread atomically increments a shared counter. After this 'checking-in', each thread calls a subroutine `Spin`, where it spin-waits until a synchronisation variable is unset by the last thread exiting the subroutine `Lock`. At the time, hand coded routines were used, but it seems that the system barrier routines may actually be faster. If a naive synchronisation method were employed, whereby a barrier is placed at the end of every parallel loop nest, then the number of synchronisation points for the TRED2 code would be 36. By detecting those dependences that are entirely local, synchronisation points can be removed reducing the total to 14.

To illustrate this point consider the fragment in Figure 6. The only synchronisation point required is following the final ENDDO due to a flow dependence on `GG(J)` from statement 3 to statement 5. The flow dependences from statements 1 to 2, 2 to 3 and 3 to 4 can be removed as all writes are local, the reads are aligned and are therefore also local. Output dependence from statement 1 to 4 and 2 to 3 can be removed due to the SPMD implementation and finally the anti-dependence from 2 to 4 can be removed as the writes and reads are totally local.

```
         DO J = 1, N+1-II
           DO K = J+1 ,N+1-II
1            E(K) = E(K)+Z(K,J)*D(J)
           ENDDO
         ENDDO
         DO J = 1, N+1-II
2          GG(J) = E(J)+Z(J,J)*D(J)
         ENDDO
         DO J = 1, N+1-II
           DO K = J+1 ,N+1-II
3            GG(J) = GG(J)+Z(K,J)*D(K)
           ENDDO
         ENDDO
         DO J = 1, N+1-II
4          E(J) = GG(J)
         ENDDO
5        G = GG(N+1-II)
```

Fig. 6. Synchronisation Example

4.5 Mapping

After the data layout and synchronisation points have been determined, it is necessary to generate the parallel program based on this decision. The appropriate basic program construct provided by KSR Fortran is the `parallel region` construct. Using this construct it is possible to generate an SPMD implementation.

The basic format of the parallelised and partitioned code is shown in figure 7. This will create p parallel threads of activity, which are responsible for the execution of the whole program; all synchronisation must be explicitly handled within the code of each thread.

All of the global array data present in the sequential program is passed as arguments to the subroutine `SUBTRED2`. The data is declared at the beginning of the main program and again as a formal parameter within the subroutine. These arrays will be read and written to by the threads and this will form all the communication within the implementation. Local variables declared solely within the subroutine will

```
C*KSR*PARALLEL REGION(numthreads=P)
      call SUBTRED2(...)
C*KSR*END PARALLEL REGION
```

Fig. 7. Parallel Region

be private to each thread (i.e. each will have it's own copy) and will include iterators, loop bounds and compiler introduced temporaries. The range of the iterators in a thread is limited by the bounds of the arrays elements written so as to ensure local writes. For example, consider column 1 in Figure 8; `Z` is to be partitioned along its second index and hence `J` will be partitioned. `D` is aligned and will be similarly partitioned.

`DO J = 1,N+1-II`
`  DO K = J,N+1-II`
`    Z(K,J) = Z(K,J)-D(J)*E(K)-E(J)*D(K)`
`  ENDDO`
`ENDDO`
`DO J = 1,N+1-II`
`  D(J) = Z(N+1-II,J)`
`ENDDO`
	`ID = ipr_mid()+1`
	`ZLO = (ID-1)*(N/P)`
	`DLO = ZLO`
	`ZHI = ID*(N/P)`
	`DHI = ZHI`
	`DO J = ZLO,ZHI`
	`  DO K = J,N+1-II`
	`    Z(K,J) = Z(K,J)-D(J)*E(K)-E(J)*D(K)`
	`  ENDDO`
	`ENDDO`
	`DO J = DLO,min(N+1-II,DHI)`
	`    D(J) = Z(N+1-II,J)`
	`ENDDO`

Fig. 8. Mapping

After removing redundant constraints ($\texttt{J} \geq 1$, $\texttt{J} \leq \texttt{N+1-II}$) in the first loop nest, the program in column 2 is generated. Here x`LO` and x`HI` are the local lower and upper bounds of array data. The bounds on the data are calculated at the beginning of the program. Some loops may have to be executed serially even though the source program contains parallelism because of the SPMD model used. For instance at one point in the program, a column of the **z** array can be written in parallel. However, as the partitioning is by column, it will be scheduled to one processor and the available parallelism is not exploited.

All scalar variables in the source program are privatised. This implies that each processor, in parallel, must do the necessary work to calculate its value, rather than waiting for it to be calculated on one processor and its value accessed when needed by other processors. Although replication increases the size of program data this would occur anyway within the ALLCACHE memory system when the scalar is referenced. Furthermore a synchronisation point is avoided and inter-statement locality may be enhanced.

False Sharing

Although the compiler strategy is based upon an extremely simple distributed memory model, a concession to SVM was made by considering false sharing in two instances. All global data objects are sub-page aligned by the `subpage` directive. This means that each object starts on a sub-page boundary, and no two objects reside on the same sub-page. This helps remove false sharing by ensuring that partitioning of data occurs along sub-page boundaries. Reduction parallelism was exploited in a modified manner to that described in [12] so as to avoid false sharing. The source program shown in column1 of Figure 9 is translated into the form in column 2.

Each thread writes the partial sum of the locally available elements of `D` into its local element of the global `PSCALE` array. `PSCALE` is expanded to a $16 \times \texttt{P}$ array to prevent false sharing. 16 elements reside on a sub-page, and this expansion can be seen as padding the array such that each thread writes to a separate sub-page. Without this expansion, up to 16 processors would be trying to gain write ownership

<table>
<tr>
<td><pre>
DO K = DLO,DHI
  SCALE = SCALE + DABS(D(K))
ENDDO
</pre></td>
<td><pre>
DIMENSION PSCALE (16,P)

PSCALE(1,ID) = 0.0D0
DO K = DLO,min (DHI,N+1-II)
  PSCALE(1,ID) = PSCALE(1,ID) + DABS(D(K))
ENDDO
call Barrier()
DO I = 1,P
  SCALE = SCALE + PSCALE(1,I)
ENDDO
</pre></td>
</tr>
</table>

Fig. 9. Partitioned Reduction

of one sub-page at any one time. Once each partial sum is calculated, every processor accesses every other processors partial sums and accumulates them into a local private copy of the scalar `scale`. Thus the reduction parallelism available has been exploited without incurring false sharing at the expense of one barrier.

4.6 Data Re-use

Once a non-local array element has been referenced, it is desirable to store it locally if it is to be referenced again. Such a scheme exploits temporal locality by detecting data re-use in the program. For a fuller description of this method see [14]. In this paper, the data re-use method is extended by considering only those array accesses that are known to be non-local. Perfectly aligned data will be local and need not be considered. For example, consider the program fragment shown in column1 of Figure 10.

<table>
<tr>
<td><pre>
DO J = ZLO,ZHI
  DO K = J,N+1-II
    Z(K,J) = Z(K,J)-D(J)*E(K)-E(J)*D(K)
  ENDDO
ENDDO
</pre></td>
<td><pre>
DO K = ZLO,N+1-II
  TD = D(K)
  TE = E(K)
  DO J = ZLO, min (ZHI,K)
      Z(K,J) = Z(K,J)-D(J)*TE-E(J)*TD
  ENDDO
ENDDO
</pre></td>
</tr>
</table>

Fig. 10. Data Re-use Transformation

References `D(J)` and `E(J)` are local as they are aligned with the second index of `Z`. The non-local references `D(K)` and `E(K)` are invariant of iterator `J`. On applying the data re-use transform, the program shown in column 2 is derived. This reduces the number of non-local array accesses by $2 \times (\mathtt{ZHI} - \mathtt{ZLO} + 1)$. The applicability of the region of access analysis depends on the ability to exploit data re-use. This post-partition transformation is therefore integral to the partitioning algorithm described earlier.

Spatial Locality The main shortcoming of the compiler approach is due to inaccuracies in the assumed architecture model. Divergence has already taken place when considering false-sharing in compiling for reduction parallelism. While the KSR-1 is a distributed memory machine with a single address space it also has the additional features of cache-only, shared virtual memory.

The compiler model assumes that the only communication overhead is between processors, and that it is a simple function of the amount of remote data accesses. Both these assumptions are not correct. The main impact on the compiler strategy is that striding through memory in the correct direction is important. In other words, spatial locality is significant, but is currently completely ignored in the data re-use section, where only temporal locality is considered. Furthermore, the main overhead is not due to increased inter-processor communication, but rather because of large intra-processor communication. Not taking stride into account means that there will be a sub-cache miss when accessing each element of

Z in four of the five significant regions. The main memory, or cache, is large enough so that, even when striding in the wrong direction, the cache lines (or sub-pages) will remain local until re-used. However, this is not true of the sub-cache, which is much smaller.

Ideally inter-processor and intra-processor communication should be optimised using a hierarchical memory model as described in [14]. However, since at present this often induces excessive loop overhead, a more simplistic method was used, where loops are re-ordered such that the innermost iterator strides through memory with a step of one. Loop interchange can be applied on all the statements with stride problems and is a post-partition transformation i.e. it does not affect any of the previous sections. In particular it has no effect on data partitioning and mapping. This techniques was used, with and without interleaving, to give two new programs. Over all the data sizes interleaving proved expensive. Interleaving should have improved performance, but it seems that the overhead due to the mod operator removed any benefit due to better load balancing. A more efficient implementation of interleaving may give better results, but due to time constraints, this was not further investigated in this paper.

5 Programmer Method

Starting point: Control flow normalised code with variable substitution, produced as described in Section 4.1. Working from a control-flow normalised version is important as it is much easier to build a mental picture of the control structure of the code. The variable substitution is less important, but it allows an easier evaluation of memory access patterns.

Step 1: Identify the 'important' sections of the code.

TRED2 has no subprograms, so this step reduces to identifying the loop nests which take the longest time to execute. This is achieved by adding timing routines round every loop in the code. There are two loops at the outermost level. Both contain significant amounts of work, but a cursory examination shows that data dependencies exist between iterations of these loops. Both are sufficiently complex that restructuring the code to expose parallelism would be a formidable task. If this is indeed possible, it could only take place without unreasonable effort by restructuring at the algorithmic level, as the complexity of analysis and transformation required at the code level would defeat all but the most dedicated programmer. At the next level of loop nest, three loop nests account for over 99% of the execution time on a single processor for problem size $N = 400$. These are shown in Figure 11.

The inner (K) loops in Fragments 2 and 3 (but not that in Fragment 1) could be parallelised as they stand. However, the timing results and a naive operation count suggest that unless N is very large (many thousands), the granularity of these loops is too small for the KSR-1 to achieve any speedup, because a tile directive is implemented via barrier synchronisations at both the beginning and the end of the loop. (Here the programmer is appealing to experience—they know that the overhead incurred in tiling a loop is $\sim$1ms. If a loop executes in less than $\sim$1ms it is certainly not worth parallelising.) In some circumstances the repeated synchronisation is not necessary, and this is a drawback of the tile directive. In these cases, however, the synchronisation would be necessary to ensure correct results. Thus, to achieve any parallel performance, the programmer must concentrate on parallelising the J loops.

Step 2: Try to expose parallelism with an automatic tool.

The automatic parallelisation tool KAP is applied to the code and its treatment of each of the three fragments observed. KAP successfully parallelises the J loop of Fragment 3 and (after trying a variety of combinations of options) the K loop of Fragment 2, but nothing in Fragment 1.

Step 3: Exploit parallelism in Fragment 3.

Following KAP, a tile directive is added to the J loop of the third fragment as it stands, using the default strategy (slice). The scalars G and K are declared as private variables to ensure that each thread has its own copy of them. This reduces the execution time for this loop by a factor of $\sim$6.5 on 8 cells for this loop (N=400). The default choice of tile size, however, is always a multiple of 16, and since the upper loop bound is increasing by one each time the loop is executed, this can result in some poor load

Fragment 1

```
DO J = 1, N+1-II
  Z(J,N+2-II) =D(J)
  G =E(J) +Z(J,J)*D(J)
  IF (II+J .LE. N) THEN
    DO K = J+1 ,N+1-II
      G =G +Z(K,J)*D(K)
      E(K) =E(K)+Z(K,J)*D(J)
    ENDDO
  ENDIF
  E(J) =G
ENDDO
```

Fragment 2

```
DO J = 1,N+1-II
  DO K = J,N+1-II
    Z(K,J) =Z(K,J)-D(J)*E(K)-E(J)*D(K)
  ENDDO
  D(J) =Z(N+1-II,J)
  Z(N+2-II,J) =0.0D0
ENDDO
```

Fragment 3

```
DO J = 1,I-1
  G =0.0D0
  DO K = 1, I-1
    G =G +Z(K,I)*Z(K,J)
  ENDDO
  DO K = 1,I-1
    Z(K,J) =Z(K,J) -G*D(K)
  ENDDO
ENDDO
```

Fig. 11. Important fragments in TRED2

balancing. Thus the tile size was set to be $\lceil \frac{i-1}{N} \rceil$, giving a reduction in execution time of $\sim$7.7 times on 8 cells for this loop.

Step 4: Expose parallelism in Fragment 2.

Attention is now concentrated on Fragment 2, as this appears easier to deal with than Fragment 1. The reason KAP fails to parallelise the `J` loop is a data dependence on array `D`. However, since each iteration of the `J` loop writes to `D(J)` but subsequent iterations only read `D(J+1)`...`D(N-1+II)`, the loop can be split as shown in Figure 12.

The double nested loop, which contains most of work in the fragment, is now clearly parallelisable. Based on the known cost of tiling a loop, the other loop contains too few arithmetic operations to be worth parallelising, and so remains sequential.

Step 5: Exploit parallelism in Fragment 2.

The new double loop has a triangular iteration space, and therefore slice strategy would result in load imbalance. Mod strategy (which performs a block interleaved partitioning of the loop) is used to improve the load balance. The tile size is chosen by experiment: although a tile size of 1 would give the best

```
DO J = 1,N+1-II
  DO K = J,N+1-II
    Z(K,J) =Z(K,J)-D(J)*E(K)-E(J)*D(K)
  ENDDO
ENDDO
DO J = 1,N+1-II
  D(J) =Z(N+1-II,J)
  Z(N+2-II,J) =0.0D0
ENDDO
```

Fig. 12. Fragment 2: parallel version

load balance, there is an overhead associated with tile generation which is proportional to the number of tiles. Thus the programmer searches for the value which minimises the execution time for this loop. For N=400, values in the range 1 to 20 are considered, as it is soon apparent that the optimal value lies between these two. Unfortunately there are now some non-local memory accesses to the array Z, because the partitioning strategies for Fragment 2 (mod) and Fragment 3 (slice) conflict. Since the dominant overhead is now the unparallelised code of Fragment 1, the programmer refrains from any further optimisation of Fragment 2 at this stage. All three fragments reference the two-dimensional array Z, which is by far the largest data object in the program, so the programmer will want to choose tiling strategies for all three fragments which give sensible data access patterns for Z.

Step 6: Expose parallelism in Fragment 1.

Fragment 1 is the hardest of the three to parallelise; it is not at all easy by inspection to spot a transformation which exposes parallelism. However the problem can be simplified with the following observations—

- The statement

  ```
  Z(J,N+2-II) =D(J)
  ```

  writes to column N+2-II of Z, but the rest of the loop only writes to columns 1 to N+1-II. Also the array D is not written to in this fragment. Thus this statement can be distributed into a separate loop.
- The conditional

  ```
  IF (II+J .LE. N) THEN
  .
  .
  ENDIF
  ```

  is superfluous, because it is implied by the bounds of the loop it contains.

Thus the fragment can be rewritten as in Figure 13 and the programmer can concentrate on the second J loop. It is still not clear how to proceed from here, so a decision is made to view the problem at the algorithm level rather than at the code level. This is a linear algebra kernel code, so it may be possible for the programmer with sufficient experience in this area to work out what this loop is doing in terms of matrix/vector operations. A 'paper run' of the fragment with N=4 is carried out, and the pattern of accesses to E, D and Z leads the programmer to recognise that this loop has the form of adding a matrix-vector product to a vector. If the subarrays D(1:N+1-II) and E(1:N+1-II) are thought of as vectors d and e, then this loop has the form $e := e + Yd$ for some Y. Further inspection shows that if the square subarray Z(1:N+1-II,1:N+1-II) is thought of as a matrix Z, and the strictly lower triangular part of Z denoted by L and the diagonal part by D, then $Y = L + D + L^T$. It is now evident that this loop is parallelisable, since the product of each row of the matrix $L + D + L^T$ with d can be computed independently. With this in mind, the fragment is rewritten as in Figure 14, where the two J loops have been fused together again. The J loop can be parallelised. Note that use of G here as a temporary variable

```
DO J = 1, N+1-II
  Z(J,N+2-II) = D(J)
ENDDO
DO J = 1, N+1-II
    G = E(J)+Z(J,J)*D(J)
    DO K = J+1 ,N+1-II
      G = G+Z(K,J)*D(K)
      E(K) = E(K)+Z(K,J)*D(J)
    ENDDO
  E(J) = G
ENDDO
```

Fig. 13. Simplified Fragment 1

```
DO J = 1, N+1-II
  Z(J,N+2-II) =D(J)
  G =E(J)
  DO K=1,J
     G=G+Z(J,K)*D(K)
  ENDDO
  DO K=J+1,N+1-II
     G=G+Z(K,J)*D(K)
  ENDDO
  E(J) =G
ENDDO
```

Fig. 14. Fragment 1: parallel version

is not necessary to exploit parallelism. However it prevents false sharing of subpages of the array `E`. Note that this is the only one of the three fragments where false sharing is a potential problem, since it is the only one where writes to a one dimensional array occur. In the other two fragments all the writes in parallel loops are to `Z`, and since it is the second index which is partitioned, there is no possibility of significant false sharing.

Step 7: Exploit parallelism in Fragment 1.

A `tile` directive is added to the `J` loop. Although this loop is load balanced in terms of floating point operations, it is *not* balanced in terms of local/non-local memory accesses, because the `J`th iteration makes `J` accesses to a row of `Z`, which are mostly non-local, and `N+1-II-J` access to a column of `Z`, which are all local. Thus mod strategy is used to achieve block interleaving.

Step 8: Data access patterns.

Use of timers, hardware monitoring information and an event logging tool (GIST) shows that by this stage the overheads are no longer dominated by unparallelised code, but by load imbalance and remote memory accesses. Both of these can be reduced by considering data access patterns. The array `Z` is the only two-dimensional array, so it is only necessary to consider accesses to `Z` when optimising for data alignment. In Fragments 1 and 2 it is necessary to distribute `Z` by interleaved blocks of columns in order to achieve load balance. Thus the obvious choice is to use interleaved blocks of columns for all three parallelised loops. To ensure that the same columns of `Z` are always accessed by the same processor, an `affinity region` directive is added to the code, which forces the mod strategy to be used on all tiled loops inside it. The programmer also needs to choose a sensible block size. Experiment shows that a block size of 4 is optimal for N=400. Very small block sizes (1 or 2) give the best load balance, but produce substantial overheads associated with tile generation. For larger N the block size should be scaled with N so that the number of blocks in constant with problem size. Thus a heuristic block size of max(4, N/100) is chosen. This could possibly be refined, given additional programmer effort.

Step 9: Alternative parallelisation of Fragment 1.

A large proportion of the remaining overheads are associated with Fragment 1. At least part of the reason for this is that, having exposed parallelism, the loop

```
DO K=1,J
   G=G+Z(J,K)*D(K)
ENDDO
```

makes non-unit strides through array Z, resulting in an increased subcache miss rate. In order to preserve unit stride through Z, a complete rethink of the parallelisation strategy for this fragment is undertaken, and an alternative strategy is chosen. It may be noted that this fragment computes additive updates to the array E. It is therefore possible to accumulate the updates computed by each thread in a local copy of the array E, and then perform a reduction operation to add in all the updates to E itself. In other words, the original J loop can be made parallel by making private copies of E. This technique is frequently used in N-body problems for computing the sum of forces on particles due to interactions with other particles, as a means of exploiting reduction parallelism with little synchronisation required. The resulting code is shown in Figure 15, where we must ensure that the array ETEMP is initialised to zero at the start of

```
DO J = 1, L
  Z(J,L+1) =D(J)
  IAM=IPR_MID()
  G=Z(J,J)*D(J)
  DO K=J+1,L
     G=G+Z(K,J)*D(K)
     ETEMP(K,IAM)=ETEMP(K,IAM)+Z(K,J)*D(J)
  END DO
  ETEMP(J,IAM)=ETEMP(J,IAM)+G
ENDDO
DO IP=0,NTHREADS-1
   DO J=1,L
      E(J)=E(J)+ETEMP(J,IP)
      ETEMP(J,IP)=0.
   END DO
END DO
```

Fig. 15. Fragment 1: alternative parallel version

the program. The first J loop may then be tiled as before, keeping it within the `affinity region` to maintain data locality and good load balance. Note that KSR Fortran does not support private arrays, and so the programmer is forced to use array expansion and indexing by processor number. Thus for the first time the code is no longer a sequential code with annotations, as it now makes explicit reference to thread IDs and the number of threads being used.

The amount of work in the second loop is proportional to both the problem size N and the number of threads p. It is possible to parallelise this loop, but it will only be worth doing so if the amount of work it contains is sufficient. A simplistic analysis suggests that it should be parallelised if $N(p-1) > c$ for some constant c. To parallelise this the loop order is inverted and a temporary scalar variable is introduced to minimise false sharing, as shown in Figure 16.

Note that the J loop is placed outermost to avoid the unnecessary synchronisation which would result if it were innermost. This gives non-unit stride through ETEMP, but since each thread accesses only N

```
DO J=1,L
   G=0.0
   DO IP=0,NTHREADS-1
      G=G+ETEMP(J,IP)
      ETEMP(J,IP)=0.
   END DO
   E(J)=E(J)+G
END DO
```

Fig. 16. Fragment 1: parallelised reduction loop

elements of `ETEMP`, and N is much smaller than the number of words in the subcache, this should not result any loss of performance due to subcache misses.

Since this loop nest has a rectangular iteration space and makes no reference to `Z`, there is no need for it to be within the `affinity region`. Therefore to minimise false sharing and tiling overheads the `J` loop is tiled using slice strategy. The constant c is determined by experiment—a value of 1000 is found to be satisfactory. Note that this parallelisation strategy requires some extra memory (for the local copies of `E`). The amount required is only $O(Np)$, compared to $O(N^2)$ for `Z`, and is therefore not a significant added demand on memory.

Step 10: Alternative parallelisation of Fragment 3.

Since the `J` loop of Fragment 3 has a rectangular iteration space, the only reason for tiling it with mod strategy is to maintain data locality. However, since Fragment 3 is contained within a separate outer loop from Fragments 1 and 2, the resulting reduction in non-local memory accesses is not very significant. On the other hand, the use of mod strategy increases the overhead associated with tile generation, and causes a small amount of load imbalance, particularly when then length of the loop is small. Thus there is a trade-off situation, and experiments show that slice strategy, as described in Step 3, is slightly more efficient for this fragment, and so is re-adopted.

Step 11: Additional parallelism in Fragment 2.

In Step 4 the `J` loop of Fragment 2 was split to give two loops, of which the first was tiled, but the second remained sequential. The decision not to tile the second loop was based on the amount of computation it contains. However, when the code is run using more than 1 thread, the cost of this loop increases, as it then produces a significant number of non-local memory accesses. If this loop is tiled (using mod strategy within the `affinity region`) this does not eliminate non-local accesses, but they are now distributed between threads, thus resulting in improved performance.

6 Experimental results

In this section we present the results of running various versions of TRED2 on the KSR-1. Results are given for three versions:

1. **Compiler** Version developed by the 'compiler' in Section 4.
2. **Programmer** Version developed by the programmer in Section 5.
3. **KAP** Version obtained by preprocessing the original source code with KAP.

All versions were compiled with the `-r8` flag to ensure that double precision real variables are handled as 64-bit floating point numbers. (Without this flag, they are handled as 128-bit numbers, and floating point arithmetic is done in software). The KAP version was preprocessed with the flags `-popt=2 -autotile=all` (see [9]). The resulting programs were then run using the `allocate_cells` command to ensure one thread per cell execution and to avoid any time slicing with other processes in the machine. The initialisation of the arrays and the printing of results are excluded from the timings. Table 2 shows the execution times in seconds for the three versions, for problem sizes $N = 128$, 256, 512, 1024 and 2048 respectively, with the exception that the KAP version was not run for the largest problem size, owing to the excessive time that would be required to do so.

		No. of processors				
N	Version	1	2	4	8	16
	Compiler	5.1	2.9	1.7	1.2	2.1
128	Programmer	4.2	2.6	1.6	1.2	1.1
	KAP	3.6	2.7	2.2	1.9	1.9
	Compiler	43.2	22.7	12.3	7.2	5.2
256	Programmer	33.5	17.7	9.8	6.1	4.4
	KAP	30.4	21.9	17.3	15.0	13.8
	Compiler	343	179	93.6	50.4	30.3
512	Programmer	264	137	71.7	39.0	23.3
	KAP	245	173	136	116	107
	Compiler	2772	1408	728	379	211
1024	Programmer	2077	1111	562	306	171
	KAP	2006	1407	1099	949	874
	Compiler	29430	15600	9461	6225	3782
2048	Programmer	22870	11420	5033	2523	1417

Table 2. Elapsed times in seconds for TRED2

Figures 17–19 present the same results for $N = 128$, 512 and 2048, in terms of temporal performance in solutions per second, which for TRED2 is simply the reciprocal of the execution time. The naive ideal performance is computed by multiplying the performance of the original sequential code by the number of processors; this is analogous to linear speedup on a speedup diagram.

For $N = 128$ performance gains in all parallel versions are poor. With such a small value of N the granularity of the parallel computations is too fine for the KSR-1 to give significant performance improvement over the sequential version, and all the parallel versions are dominated by synchronisation overheads. There is little to choose between Programmer and Compiler for this problem size, except on 16 processors, where Compiler takes a significant drop in performance. This is due to the effects of false sharing which become important when $N/p < 16$, the number of words on a subpage. KAP is the fastest version on one processor, but it scales poorly, only just outperforming Compiler on 16 processors. Indeed, for all problem sizes up to 1024, KAP performs very poorly, never achieving more than about two times the performance of the sequential code.

With $N = 256$, 512 and 1024, Programmer outperforms Compiler by 20-30% regardless of the number of processors. The scaling behaviour of these two versions is very similar for these problem sizes, and steadily improves as the problem size increases.

For $N = 2048$, the memory requirement exceeds the memory available on a single cell. Thus for small numbers of processors, high numbers of page misses are generated. On this problem size, Programmer scales very well, with some superlinearity observed, whereas Compiler is again 30% slower on one processor, but over 2.5 times slower on 16 processors.

6.1 Human Effort

The work described covers a long period and was concurrent with other projects and commitments. The programmer spent approximately eight working days optimising the code over a four month period. In particular, Step 6 (obtaining a parallel version of Fragment 1) took about a day.

The compiler version took considerably more time. The strategy was based on previous work but applying it to a general Fortran program required three months research while the actual implementation on the KSR-1 took another three months. Ensuring that the synchronisation points preserved program meaning (and did not deadlock) was the most time consuming phase; the implementation of interleaving was a close second. The strategy has recently been implemented in Sage++ [3] taking eight man months.

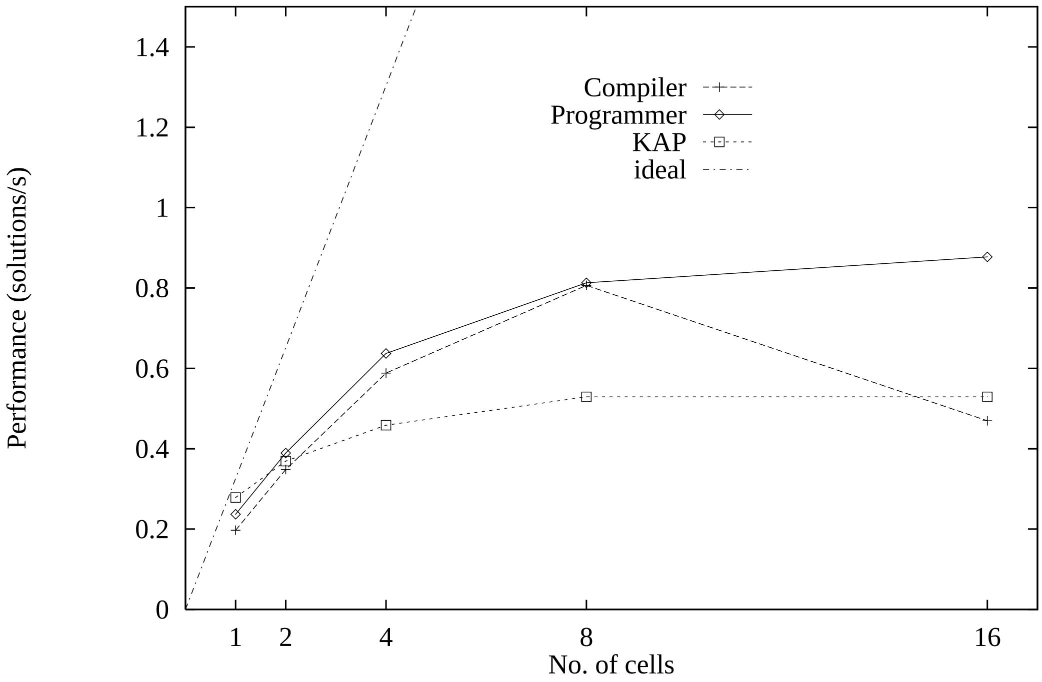

Fig. 17. Temporal performance of TRED2, $N = 128$

7 Perspectives

7.1 On the Compiler Method

Many of the differences between the two approaches arise from the compiler and programmer employing distinctly different programming models. The compiler's model is based on the partition of data, whereas the programmers model is essentially a shared memory approach, with emphasis on the partition of computation. Of course the programmer has to be aware that memory accesses are not uniform, but data layout is only considered when it is perceived that communication is contributing significantly to the overheads. On the other hand, the compiler approach is heavily data oriented, and the enforcement of the local write rule results in code which is strongly reminiscent of a message-passing paradigm, even though there are no explicit messages. One effect of this difference is that the programmer has to make relatively few changes to the source code, while the compiler code is almost unrecognisable from the sequential version, and is also difficult to read. With the continuing poor performance of 'black-box' style parallelising compilers, it is becoming clear that it is necessary for the programmer to be able to understand and influence compiler decisions. Because the compiler's output code is so hard to interpret, the compilers decisions would have to be communicated to the programmer via some interface other than simply the transformed code.

As is clear from the programmer's analysis of Fragment 3, the data-oriented approach is not always optimal. In this case the load imbalance in the affinity based schedule employed by the compiler outweighs the communication costs in the programmers version. Indeed, load balancing appears to be a weak area in the compiler strategy. While, in general, the programmer is able to design quite sophisticated schedules, the compiler is limited to interleaving, and even this seems messy and expensive to implement. The local-write rule employed by the compiler can be over-restrictive. Some parallel loops become sequentialised, and a sequential loop with data dependencies between iterations would require a lot of synchronisation. Notice that the programmer does not have to worry about making local copies of remote data, as this is provided automatically by the ALLCACHE memory system.

The compiler's partitioning trade-off algorithm appears naive. It imposes an order on considerations which is program independent, takes no account of any machine parameters, and treats all significant

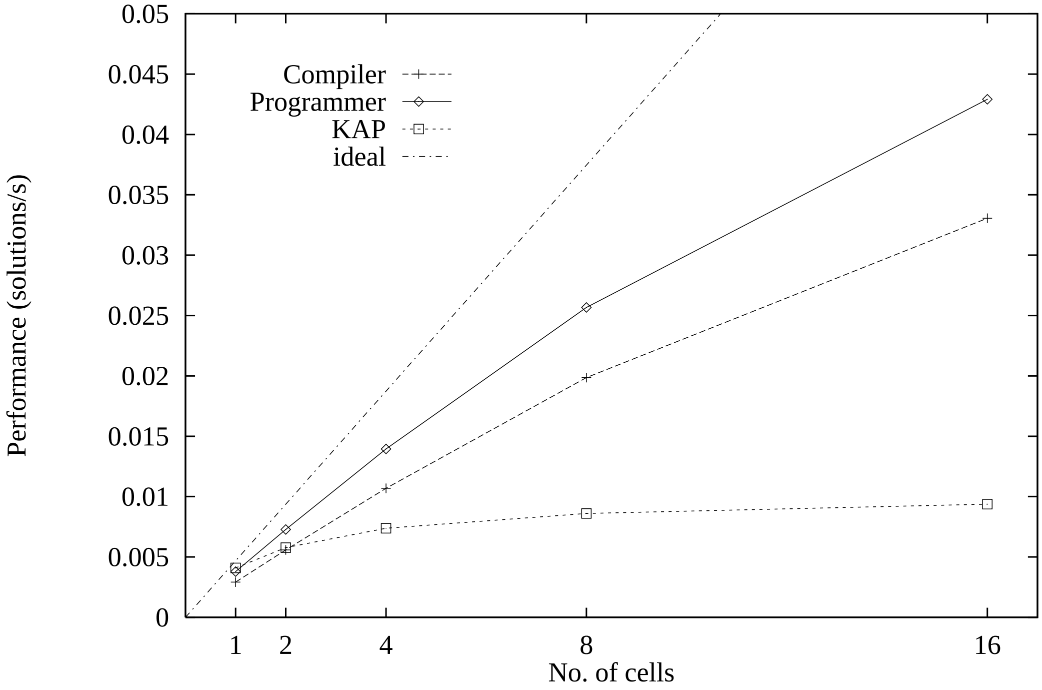

Fig. 18. Temporal performance of TRED2, $N = 512$

regions as equally significant. Decisions are based on a scoring mechanism: a partition is chosen because it is optimal for the most significant regions, even though it could be disastrous for other regions, and therefore not the overall optimal choice. The compiler assumes that communication is more important than synchronisation. For the KSR-1, which has relatively fast communication, but relatively slow synchronisation, compared with other parallel architectures, this may not be a sound assumption. Having said all this, the compiler algorithm does do a good job on TRED2! The compiler does of course lack the advantage of profiling information to guide it, which the programmer usefully utilises. Nevertheless, it would seem likely that the compiler would benefit, in the general case, from a more quantitative assessment of the overheads associated with each partitioning.

The compiler's programming model does have some marked benefits. The compiler is able to do transformations which would be very unpleasant to perform manually, and has no worries about the correctness of any transformations made. It appears considerably easier for the compiler to spot unnecessary synchronisation and eliminate it. The compiler is also much better equipped to detect false sharing, and the local write rule makes it relatively easy to avoid. For the programmer, false sharing can be quite difficult to detect. Note, however that the compiler does not take into account false sharing at the page (rather than subpage) level, which can result in excessive replication of data, though this problem is not encountered in TRED2.

Overall, the programmer's method results in faster code with far fewer changes to the original source. However, much of the method is reliant on intuition and experience, combined with the availability of detailed profiling information. The compiler is restricted to performance prediction rather than performance measurement. Its programming model appears unnecessarily restrictive, and it is clear that some architectural parameters should be taken into account in order to provide better performance prediction. Having said this for the test case we have used in this experiment, the compiler far outperforms the current commercially available compiler KAP, and comes quite close to the performance achieved by the programmer.

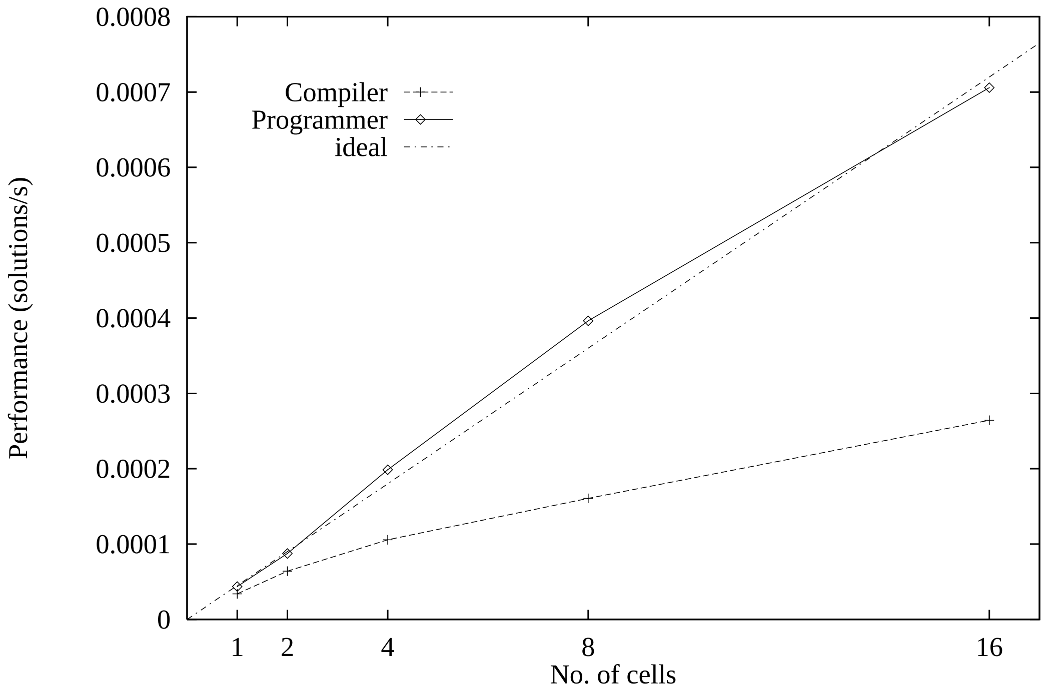

Fig. 19. Temporal performance of TRED2, $N = 2048$

7.2 On the Programmer Method

One obvious difference between the two approaches is that the compiler approach applies a lot more analysis and transformations but is out-performed by the programmer. The major feature of the programmer method is the heavy use of profiling to guide program optimisation. On the other hand it is somewhat ad hoc in nature. It seems that much of the knowledge of program and machine behaviour is informal in nature and is not applied in a systematic step-wise manner so that it may be generalised for other programmers. However, this is a reasonable approach, if given a certain time constraint and a particular program to optimise, rather than programs in general. Clearly, modeling the KSR-1 as a single level distributed memory machine gives encouraging results, but is too inaccurate to compete with the more intimate knowledge of the programmer.

The KSR programming model leaves much to be desired. It is considered that tiling innermost loops is not worthwhile due to tile overhead. However much of this tile overhead is due to unnecessary barrier synchronisations after parallel loops. This leads to the fusion of loops, trying to tile outer loops, and leaving inner parallel loops untiled. This unnecessary sequentialisation also increases communication. All data referenced within a sequential loop will move to wherever the master thread is scheduled. An affinity based approach keeps the data still and sequentialises by synchronisation between processors. This is a KSR Fortran problem rather than a KSR-1 feature, yet is rarely considered worth tackling.

This compiler approach uses a static allocation of data based on the assumption that data movement due to write invalidation or false sharing is very expensive. This is certainly true in software-based SVM systems such as KOAN, but on the KSR-1 it seems that this is not always the case. In the third fragment of Figure 11 the programmer found that the overhead due to load imbalance using an affinity based schedule is greater than that using a loop based scheme.

Overall the main difference is that the programming approach is more effective per program in terms of performance and effort, but seems relatively ad hoc and, of course, has to be repeated for each application program. Initially there seems little that can be gained from one implementation to guide the next on a possibly different SVM machine. However, by carefully examining the techniques used, it seems possible that some may be incorporated into a compiler allowing the programmer to optimise at a higher level.

7.3 On Current Differences and Future Developments

At first glance, there may seem to be little commonality between the approaches used by optimising compilers and programmers. This is particularly evident if one compares the resulting code. Perhaps the most significant difference is that programmers optimise in cyclic manner (modify, run, repeat until satisfied) which is in stark contrast to the compiler's single pass approach. The programmer uses profiling information extensively, while the compiler does not use it at all. The approaches to program parallelisation are notably different, with the compiler relying on loop distribution, while the programmer determines parallelism at the highest loop level, maintaining as much original structure as possible. Breaking a problem (multi-statement imperfectly nested loop) into many simpler sub-problems (multiple perfectly nested loops) is an attractive solution to the compiler, but for the programmer, the original code structure has meaning and it is easier to work with a few complex loop nests than multiple simpler ones. Nevertheless, there is definite common ground in that both approaches are concerned with reducing overheads, and both use a similar classification of overheads in order to guide decisions. Both programmer and compiler also use broadly similar techniques to reduce overheads, though the fact that each uses a different programming model tends to obscure this.

This experiment highlights a problem common to all compiler schemes which try to be applicable to a range of architectures however narrowly defined. The relative impact of any overhead will vary from machine to machine and should influence any compiler strategy. Rather than generating a new strategy for every machine type, what is needed is an explicit cost model based upon machine parameters, which will guide compilation. This model must incorporate those parameters that are relevant to the particular compilation model used and must guide rather than evaluate program transformations. Clearly, generating a possibly infinite number of programs and then checking it against a cost model is not practically feasible. Understanding the important costs can only be gained from those using the machine—programmers. By incorporating their knowledge into a compiler, some of the time consuming process of parallelising and mapping a program may be saved.

This experiment also exposes the essentially ad-hoc nature of human parallel program development. Although we have shown that the programmer, through profiling and analysis of overhead sources, has a more or less systematic method of identifying problems in a parallel implementation, the solution of these problems still relies to a large extent on experience and intuition. There remains much potential to make this process more systematic, and hence more accessible to the non-expert. The ability to work together with a compiler that is able to automate sophisticated transformations, give useful and intelligible feedback on its decisions, and possibly achieve some degree of performance prediction, will be of significant benefit.

It is clear that the role of optimising compilers in parallel scientific computing is as a tool to assist programmers, rather than as a replacement for them. If the compiler is to be an effective tool, then it will have to fit into the programmer's cyclic development method—this has a number of implications for the compiler. Firstly, the compiler must be able to follow directives placed in the program as well as perform optimisations automatically. Programmers will always have domain specific knowledge, unavailable to the compiler and must therefore be allowed to override any compiler decision. Naturally, this implies that the compiler must report its decisions and the reasons why it made them in a manner that is understandable to the programmer. This information will allow the programmer to provide appropriate information/directives leading to a better implementation without requiring the programmer to completely restructure the code. Ideally, optimising the program would consist of a number of iterations where different directives are inserted, rather than rewriting sections of the program. If we were to extend this idea, the programmer would be encouraged to write their program in languages such as HPF, Fortran90 or even higher level ones. This will aid portability and maintainability, as only the directives should have to change.

To facilitate this, optimising compilers will have to be more open; each stage (parallelising, mapping etc.) must be able to take information from previous stages or from directives. They will also have to provide sensible implementations in the presence of partial direction. If, for example, the programmer specifies how one section of code is to be mapped, then the mapping of the remaining program sections should accommodate this information. This is essential if compilers are to evolve beyond concentrating on local loop analysis to global program-wide knowledge.

A prototype of the compiler strategy described is the paper is now implemented. The compiler is to be extended in the light of this experiment. Each of the compiler phases will be broken into units where important decisions are based on input values rather than hard-coded internal algorithms. These values may be generated by earlier phases or directly from directives. This allows variable control by the programmer and, because of modularity, simplifies targeting the compiler for different platforms and incremental addition of functionality.

Since the compiler can at best contain a relatively simple model of machine performance, programmers will want to continue to use profile information to guide optimisation even if they are working together with a sophisticated compiler. Thus for maximum benefit a compiler should be part of an integrated development environment that supports editing, interactive compilation, running programs, performance monitoring, and presentation of profiling information. The challenge is to define a directive system which allows higher level control than, say, enforcing a particular loop transformation. This will include control of execution model (SPMD, fork/join, multi-threading) and strategy where the relative cost of overheads can be defined (i.e. load balance, communication, synchronisation etc.). Giving the 'right' information back to the user is essential and this must be defined by the user. Experience within our group has show that different users have different requirements and that interfaces must allow some customisation. The programmer will have control on the level of detail in compiler feedback and will be able to view the program in a number of ways; by significant regions, variable name, synchronisation point etc. The FORGE system is an early example of such a system [1].

8 Conclusion

For the case of TRED2, for which it is possible to completely statically analyse data accesses, but is otherwise non-trivial to optimise for parallel execution, the compiler strategy described in Section 3 is shown to give almost comparable performance to an expert programmer, and to outperform the commercially available parallelising compiler, KAP by a significant margin.

Through the detailed description of the application of programmer and compiler strategies to TRED2, both differences and common ground between the two strategies have been highlighted. Strengths and weaknesses of the strategies have been identified, giving insight into how the two approaches should be integrated in a unified program development environment.

Future experiments on different programs and architectures will aid the development of the compiler, and its user interface. This system will allow the programmer to concentrate on those areas where the compiler is poor and try complex strategies which would be too costly to try by hand. In this way it is hoped that a more useful partnership between programmer and compiler will be achieved, resulting in a systematic combined strategy allowing rapid development of efficient parallel programs.

References

1. A.P.R., *Forge 90: Automatic Parallelizer for High Performance Fortran 77 on Distributed Memory System User's Guide*, Applied Parallel Research, 5550 Main St., Suite I, Placerville, CA 95667, 1993.
2. Bodin F. and O'Boyle M.F.P., *The Abstract System: Design and Implementation of a Parallelising Compiler*, in preparation.
3. Bodin, P. Beckman, D. Gannon, Srinivas J.G.S., *Sage++: A Class Library for Building Fortran and C++ Restructuring Tools*, Second Object-Oriented Numerics Conference, Oregon (USA), April 1994.
4. Burkhart H. and Millen R. *Performance-Measurement Tools in a Multiprocessor Environment*, IEEE Trans. on Computers, 38(5) pp725-737, 1989.
5. Crovella M.E. and LeBlanc T.J., *The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation*, University of Rochester, Department of Computer Science, Technical Report 479, 1993.
6. Egan G.K., Riley G.D. and Bull J.M., *Parallelisation of the SDEM Distinct Element Stress Analysis Code on the KSR-1*, Proceedings of ACM International Conf. on Supercomputing, 1994.
7. Falcó Korn C., Bull J.M., Riley G.D. and Stansby P.K., *Parallelisation of a Three-dimensional Shallow Water Estuary Model on the KSR-1*, to appear in Scientific Programming.
8. Gupta M., *Automatic Data Partitioning on Distributed Memory Multicomputers*, University of Illinois at Urbana-Champaign, Centre for Reliable and High Performance Computing, UILU-ENG-92-2237,CRHC-92-19, 1992.
9. K.S.R., *KSR Fortran Programming*, Kendall Square Research, 170 Tracer Lane, Waltham, MA, 1992.
10. Li J. and Chen M., *Index Domain Alignment: Minimising Cost of Cross-Referencing between Distributed Arrays*, IEEE Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, 1990.

11. Lu L-C., *A Unified Framework for Systematic Loop Transformations*, 3rd ACM Symposium on the Principles and Practice of Parallel Programming, 1991.
12. O'Boyle M.F.P., *Program and Data Transformation for Efficient Execution on Distributed Memory Architectures*, Tech Rep University of Manchester, 93-1-6, 1993.
13. O'Boyle M.F.P., *A Data Partitioning Algorithm for Distributed Memory Compilation"*, Proceedings of PARLE 94, Springer Verlag, 1994.
14. O'Boyle M.F.P., *Hierarchical Locality Optimisations for NUMAs*, Proceedings of EuroMicro PDP, Sanremo, 1995.
15. Padua D.A. and Wolfe M.J., *Advanced Compiler Optimizations for Superomputers*, Communications of the ACM Vol.29 No.12 pp 1184-1201, 1986.
16. Pugh W., *Uniform Techniques for Loop Optimisation*, International Conference on Supercomputing, 1991.
17. Pugh W. and Wonnacott D., *Eliminating False Data Dependences using the Omega Test*, ACM SIGPLAN Programming Languages Design and Implementation, 1992.
18. Kelly W. and Pugh W., *Generating Schedules and Code within a Unified Reordering Transformation Framework*, University of Maryland Technical Report UMIACS-TR-92-126, 1992.
19. Wolfe M., *Optimising Supercompilers for Supercomputers*, Research Monograph in Parallel and Distributed Computing, Pitman, London, 1989.
20. Wolf M.E. and Lam M., *An Algorithmic Approach to Compound Loop Transformations*, in *Advances in Languages and Compilers for Parallel Processing*, eds Nicolau A., Gelernter D., Gross T. and Padua D., Pitman London, 1991.

Appendix: TRED2 listing

```
      DO I = 1,N
        DO J = 1,N
          Z(J,I) = A(J,I)
        ENDDO
        D(I) = A(N,I)
      ENDDO
      IF (N .NE. 1) THEN
      DO II = 2, N
         H = 0.0D0
         SCALE = 0.0D0
         IF (II .LE. N-1) THEN
           DO K = 1,N+1-II
             SCALE = SCALE + DABS(D(K))
           ENDDO
           IF (SCALE .EQ. 0.0D0) THEN
             E(N+2-II) = D(N+1-II)
             DO J = 1, N+1-II
               D(J) = Z(N+1-II,J)
               Z(N+2-II,J) = 0.0D0
               Z(J,N+2-II) = 0.0D0
             ENDDO
           ELSE
             DO K = 1, N+1-II
               D(K) = D(K) / SCALE
               H = H + D(K) * D(K)
             ENDDO
             G = -DSIGN(DSQRT(H),D(N+1-II))
             E(N+2-II) = SCALE * G
             H = H - D(N+1-II) * G
             D(N+1-II) = D(N+1-II) - G
             DO J = 1, N+1-II
               E(J) = 0.0D0
             ENDDO
             DO J = 1, N+1-II
               Z(J,N+2-II) = D(J)
               G = E(J) + Z(J,J) * D(J)
               IF (II+J .LE. N) THEN
                 DO K = J+1 ,N+1-II
b:                 G = G + Z(K,J) * D(K)
a:                 E(K) = E(K) + Z(K,J) * D(J)
                 ENDDO
               ENDIF
               E(J) = G
             ENDDO
             F = 0.0D0
             DO J = 1,N+1-II
               E(J) = E(J)/H
               F = F + E(J) * D(J)
             ENDDO
             HH = F / (H + H)
             DO J =1, N+1-II
               E(J) = E(J) - HH * D(J)
             ENDDO
             DO J = 1,N+1-II
               DO K = J,N+1-II
c:               Z(K,J) = Z(K,J)-D(J)*E(K)-E(J)*D(K)
               ENDDO
               D(J) = Z(N+1-II,J)
               Z(N+2-II,J) = 0.0D0
             ENDDO
             ENDIF
           ELSE
             E(N+2-II) = D(N+1-II)
             DO J = 1, N+1-II
               D(J) = Z(N+1-II,J)
               Z(N+2-II,J) = 0.0D0
               Z(J,N+2-II) = 0.0D0
             ENDDO
           ENDIF
           D(N+2-II) = H
         ENDDO
         DO I = 2,N
           Z(N,I-1) = Z(I-1,I-1)
           Z(I-1,I-1) = 1.0D0
           IF (D(I) .NE. 0.0D0) THEN
             DO K = 1,I-1
               D(K) = Z(K,I) / D(I)
             ENDDO
             DO J = 1, I-1
               G = 0.0D0
               DO K = 1, I-1
d:               G = G + Z(K,I) * Z(K,J)
               ENDDO
               DO K = 1, I-1
e:               Z(K,J) = Z(K,J) - G * D(K)
               ENDDO
             ENDDO
           ENDIF
           DO K = 1,I-1
             Z(K,I) = 0.0D0
           ENDDO
         ENDDO
        ENDIF
        DO I = 1, N
          D(I) = Z(N,I)
          Z(N,I) = 0.0D0
        ENDDO
        Z(N,N) = 1.0D0
        E(1) = 0.0D0
```

This article was processed using the LaTeX macro package with LLNCS style