Department of Computer Science
University of Manchester

Manchester M13 9PL, England

Technical Report Series

UMCS–93–7–1

Michael O'Boyle

A Data Partitioning Algorithm for Distributed Memory Compilation

A Data Partitioning Algorithm for Distributed Memory Compilation *

Michael O'Boyle
Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK

Abstract

This paper proposes a compiler strategy for mapping FORTRAN programs onto distributed memory computers. Once the available parallelism has been identified, it has to be scheduled so as to minimise overhead. The minimisation of different costs will suggest different data and computation partitions. This is further complicated, as the effectiveness of the partition will depend on later compiler optimisations. For this reason, partitioning is at the crux point of compilation and has led several researchers to propose that it be left to the user. It is our contention that it is possible to automatically determine a good data partition and furthermore to make use of the analysis in later optimisation stages. This paper describes an automatic data partition algorithm which is based on four different analysis techniques. By determining the relative merit of each form of analysis, a data partitioning decision is made. By integrating this algorithm into an overall compilation strategy, effective use can be made of any data partitioning decision. To test the strategy, it has been applied to a real non-trivial program, TRED2. The resulting program was ran on a 32 cell KSR-1 where this strategy was compared with the native parallelising compiler and a hand coded version of the program. It is shown that the performance of the approach described in this paper is comparable to that of hand-coded techniques.

Technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `m1.cs.man.ac.uk` (130.88.13.4) in the directory `/pub/TR`. The files stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

1 Introduction

The exploitation of parallelism to achieve faster computation has been a subject of research for many years. In particular, distributed memory computers hold the promise of scalable performance. However they have been difficult to program and recent effort has concentrated on exploiting data parallelism within FORTRAN programs by providing user inserted data distribution pragmas. Notable examples are High Performance FORTRAN (HPF), FORTRAN D [4] and Vienna FORTRAN [1].

The scheduling of data and work to processors can have a dramatic effect on program performance. Furthermore, there are conflicting factors which have to be resolved for a successful implementation. It is for this reason that many researchers have suggested that the decision be left to the user. However, work in shared memory compilation has shown that there is a close interaction between partitioning and other compiler transformations [23]. It is the view of this paper that automatic data partitioning is possible and furthermore it must be considered in the context of the whole compilation process.

Initial stages of program transformation are involved in exposing and maximising parallelism. The later stages are involved in reducing overhead. Partitioning is at the mid-way point and must balance conflicting costs in a sensible manner to give a reasonable implementation. For this reason, it is at the crux point of compilation. While partitioning is an important transformation, its effectiveness depends on those transforms that occur before and after. If no program parallelism is exposed then partitioning will not be able to map it to a parallel machine. Similarly, if later transformations do not reduce unnecessary communication and synchronisation then the potential performance of the implementation will be lost due to overheads.

Recently other researchers have been interested in automating data partitioning. In [7] a method based on training sets is described. Understanding how a set of kernel programs behave on a particular machine will help determine a good data partition for user programs, it is argued. In [10] the data partition decision is based on pattern matching of array references and synthesised communication costs. In [22], the main concern is determining at what point a different data partition should be used based on program and machine size. Finally in [5] a graph representation of communication costs between arrays is used, where the weights on arcs describe the potential cost if an array were not partitioned in its desired manner, to determine the best data partition.

The major contribution of this paper is that it describes four analysis techniques upon which data partitioning can be based and a data partitioning algorithm that resolves any trade-offs. Furthermore, it is shown that by integrating the partition stage within an overall compilation strategy, it is possible to achieve performance comparable to a hand coded version of a non-trivial scientific program TRED2. TRED2 is a sample routine from the EISPACK library. It reduces a symmetric matrix to a symmetric tridiagonal matrix using and accumulating orthogonal similarity transformations.

1.1 Compilation Strategy

An application of a transformation must be considered in terms of what cost(s) it is influencing. There are an infinite number of transformations that can be applied to a program and therefore an undirected search of this space is clearly infeasible. Furthermore, the cost function that we are trying to minimise, i.e. parallel time, may be intractable. It is therefore desirable that any transformation scheme has a simpler cost model where transformations can be applied in a finite step-wise manner. Obviously there is a trade-off between accuracy and complexity of such a scheme. The method we propose addresses those costs which are most significant in distributed memory architectures and does not require any backtracking.

In this paper the sequence of transformations can be considered as minimising costs in a recursive or top-down manner. Firstly the architecture is considered as a number of processor/memory pairs where minimising load imbalance and interprocessor communication is of primary importance. Once a node program is derived, more architectural detail is incorporated. Any further transformations will be determined by the architecture's memory model, in particular by whether it is based on message passing or has a uniform addressing mechanism. Many of these costs/issues can be examined in a memory access independent manner. The general strategy is as follows:

Pre-Process

Initially a program may contain GOTOs and induction variables which make subsequent analysis difficult. At this stage control flow normalisation and induction variable recognition are performed so as to aid the parallelisation and subsequent stages.

Parallelise

Only loop parallelism is investigated at this stage. Transformations for parallelism are non-trivial and beyond the scope of this paper. An example is provided to demonstrate techniques that can be used.

Align

Data is eventually allocated to a machine processor and determining where it finally resides is broken into two parts, alignment and partitioning. Alignment transformations are applied so as to maximise the number of references that will be local.

Partition

At this stage four different analyses are applied to those regions of the program that will eventually dominate execution time. These analyses are concerned with parallelism, load imbalance, alignment and the region of access. The first two are self-explanatory, while the later two are concerned with minimising non-local access. However, in general they will give conflicting advice on partitioning, so a data partioning algorithm is described which arbitrates between conflicting analyses based on simple heuristics.

Interleave

If the eventual partitioned index is not load balanced, then an interleaving transformation that spreads the work load across the processors is applied.

Non-local Data Re-use

If a non-local array element is to be referenced many times, the program is transformed such that it is stored locally. Such a scheme exploits temporal locality by detecting data re-use in the program.

Memory Access Specific Optimisations
Depending on the access model of the architecture, different transformations are required to best exploit memory usage. As the target machine described in this paper is the KSR, cache based transformations are applied.
Synchronisation
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.

Section two describes the notation used in this paper and gives a brief description of the SPMD model of computation. Section three outlines the transformations necessary to uncover parallelism and align data before partitioning is performed. Section four describes a data partitioning algorithm and section five presents post partitioning transformations that improve performance. Section six gives a description of the issues specific to the KSR 1, while section seven describes the synchronisation method used in this strategy. In section eight the method put forward by this paper is tested against a parallelising compiler and a hand coded version of TRED2 and the results are followed by some concluding remarks.

2 Language and Models

Although the presentation of material is fairly informal throughout this paper, a brief outline is given of the notation used in describing the various analysis and transformation stages. For more details of the notation see [13].

2.1 Iteration Space

The loops appearing in a program surrounding a statement can be represented by an $m \times 1$ vector where m is the number of surrounding loops.

$$J = [j_1, j_2, \ldots, j_m]^T \tag{1}$$

are the *iterators* surrounding a statement.

The affine bounds of iterators can described by the following set of inequalities:

$$LJ \geq l \tag{2}$$

$$UJ \leq u \tag{3}$$

where L and U are $(m \times m)$ lower unit triangular matrices; J,l and u are $(m \times 1)$ vectors. The range of values taken on by J defines the **iteration space** of a statement. This set of inequalities can be re-written as:

$$\left[\begin{array}{c} -L \\ \hline U \\ \hline \mathcal{E} \end{array}\right] J \leq \left[\begin{array}{c} -l \\ \hline u \\ \hline \varepsilon \end{array}\right] \tag{4}$$

```
DOUBLE PRECISION A(64,128),B(456,200),C(200,264)
.
.
.
DO I = 1,N-1
  DO J = 1, 2*I-1
    DO K = I, I+J+6
      IF (I+3 .LE. K+J) THEN
        A(I,J) = A(I,J) + B(2*J+K,I+J)* C(K+1,I+K)
      ENDIF
    ENDDO
  ENDDO
ENDDO
.
```

Figure 1: Example Program Fragment

where $\mathcal{E}, \varepsilon$ are a matrix/vector pair consisting of extra constraints due to loop body conditionals.

2.2 Index Domain

To allow data partitioning it is necessary to record the size and dimensionality of each array. Each array index domain, v, has the following form:

$$[i_1^v, i_2^v, \dots, i_{N_v}^v]^T = \mathcal{I}^v \tag{5}$$

where N_v is the number of array dimensions, i_x^v is the xth index of the array v, and $\mathcal{I}^v$ is the vector containing all the indices of that array. The indices have a certain range which describe the size of the array as follows:

$$\lambda \leq \mathcal{I} \leq \upsilon \tag{6}$$

where $\lambda = [\lambda_1, \dots, \lambda_{N_v}]^T$ and $\upsilon = [\upsilon_1, \dots, \upsilon_{N_v}]^T$ are $N_v \times 1$ vectors. As $\mathcal{I}$ is restricted to rectangular forms, only constant vectors are required to describe its bounds.

The affine subscripts of an array reference within a statement can be represented in a linear algebraic fashion by a *subscript matrix/vector pair*.

If θ is the number of array references in a statement then $(\mathcal{U}^r, \nu^r), r \in 0, \dots, \theta - 1$ is the matrix/vector pair corresponding to the rth array reference in the statement.

To illustrate the preceding notation, consider the program fragment in figure 1. The iteration space formed by the three loops i, j, k and the conditional expression can be represented as follows:

$$
\left[\begin{array}{rrr}
-1 & 0 & 0 \\
0 & -1 & 0 \\
1 & 0 & -1 \\
\hline
1 & 0 & 0 \\
-2 & 1 & 0 \\
-1 & -1 & 1 \\
\hline
1 & -1 & -1
\end{array}\right]
\left[\begin{array}{c} i \\ j \\ k \end{array}\right]
\leq
\left[\begin{array}{r}
-1 \\ -1 \\ 0 \\ \hline n-1 \\ -1 \\ 6 \\ \hline -3
\end{array}\right]
\tag{7}
$$

From the declaration of the array variables, the index domains are trivially determined:

$$
\left[\begin{array}{c} 1 \\ 1 \end{array}\right] \leq \left[\begin{array}{c} i_1^a \\ i_2^a \end{array}\right] \leq \left[\begin{array}{r} 64 \\ 128 \end{array}\right] \tag{8}
$$

$$
\left[\begin{array}{c} 1 \\ 1 \end{array}\right] \leq \left[\begin{array}{c} i_1^b \\ i_2^b \end{array}\right] \leq \left[\begin{array}{r} 456 \\ 200 \end{array}\right] \tag{9}
$$

$$
\left[\begin{array}{c} 1 \\ 1 \end{array}\right] \leq \left[\begin{array}{c} i_1^c \\ i_2^c \end{array}\right] \leq \left[\begin{array}{r} 200 \\ 264 \end{array}\right] \tag{10}
$$

Each of the four array references can be written in matrix form: For example the reference to $A(i, j)$ is represented by $\mathcal{U}^1 J + v^1$ where

$$
\mathcal{U}^1 J = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right], v^1 = \left[\begin{array}{c} 0 \\ 0 \end{array}\right] \tag{11}
$$

Usually each array reference is represented by the pair $(\mathcal{U}^r, v^r)$, without reference to J. The remaining references are as follows:

$$
A(i, j) : \mathcal{U}^2 = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right], v^2 = \left[\begin{array}{c} 0 \\ 0 \end{array}\right] \tag{12}
$$

$$
B(2 * j + k, i + j) : \mathcal{U}^3 = \left[\begin{array}{ccc} 0 & 2 & 1 \\ 1 & 1 & 0 \end{array}\right], v^3 = \left[\begin{array}{c} 0 \\ 0 \end{array}\right] \tag{13}
$$

$$
C(k + 1, i + k) : \mathcal{U}^4 = \left[\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 0 & 1 \end{array}\right], v^4 = \left[\begin{array}{c} 1 \\ 0 \end{array}\right] \tag{14}
$$

2.3 SPMD Computation

Once any data parallelism has been detected by the compiler, a method to exploit this parallelism on a distributed memory computer must be used. The method of evaluation has often been single program multiple data, SPMD, where the array data is distributed across the processors so that each processor works upon a separate section of the array in parallel. Essentially each process runs the same program but operates on different data which are independent portions of an array. Each processor has one process and local data. In this paper, two useful methods of exploiting parallelism are considered.

- **Creation Parallelism** Each process accesses any data required for the calculation of its local portion of a distributed array. After accessing the necessary data, each local array is calculated in parallel.

- **Reduction Parallelism** Each process first performs computation upon its own local data in parallel, the result of which is then accessed by another processor.

Creation parallelism can be viewed as accessing data before computation is performed at the site where the write will take place. In contrast, reduction parallelism implies that every processor performs part of the total computation on its own local data in parallel before its results are accessed by the processor which will perform the write. Essentially if a parallel iterator occurs on the left-hand side of an assignment it is involved in creation or "owner-computes" parallelism, while a parallel iterator appearing exclusively on the right-hand side allows only vector type or reduction parallelism.

There is no fork/join overhead which has been traditionally associated with shared memory parallel models. However, it is less dynamic, and may require barrier synchronisation, but sensible analysis can remove many of the synchronisations.

3 Pre-Partitioning Transformations

In this section the necessary transformations to uncover parallelism and align data before partitioning are described.

3.1 Pre-Processing and Parallelism

It is not the purpose of this paper to propose any particular parallelisation method . An enormous amount of research has been undertaken in this area [25],[11],[21],[26],[19], and there still remains much to do. Rather, an outline of the techniques used to parallelise TRED2, the program used in the example section, is given to illustrate the source of the difficulties.

A considerable amount of preprocessing must take place before TRED2 can be sensibly parallelised. Firstly control-flow normalisation or "de-spagettisation" is attempted where the GOTOs are replaced by conditionals leaving the structure of the code as much intact as is possible. Secondly, induction variable substitution takes place which is then followed by forward substitution of scalar temporaries which hold array values. Loop distribution and scalar expansion are employed in a manner similar to [11] whereupon DO loops are determined as being either parallel, reductive or sequential. Once this has taken place, it is possible to analyse and partition the program. The original TRED2 code and the parallelised form can be found in appendix 1.

It is desirable that the final form should be as loop distributed as possible. This allows the easy calculation of all local iteration bounds when using an owner computes implementation. This approach is similar to that used by vectorisation compilers. It is also the form achieved by modern parallelisation techniques [11] . It is a fallacy to assume that loop distribution necessarily causes greater overheads on parallel machines. Only if each DO is considered to

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

Figure 2: Original TRED2 Program Fragment

be a fork and each ENDDO a join or barrier is this true. In the synchronisation section, it is shown that many of these synchronisation points can be removed.

Extracting parallelism by loop distribution is by no means trivial; whilst it is relatively easy to see, in figure 2, that the *K* loop will provide some parallelism and its two statements do not interfere, the *II* and *J* loops look very sequential. This is in fact the conclusion of [5] when using the Parafrase environment. However, if several highly non-trivial transformations are applied, the program in figure 3 can be obtained where the distributed *J* loop is parallel as well.

To get to this form requires careful analysis of the order of access to each of the arrays. The reduction scalar *G* is scalar expanded to an array to facilitate loop distribution and parallelisation.

3.2 Alignment

In the compilation model used in this paper, data is eventually allocated to a processor. Determining where data finally resides is broken into two parts, alignment and partitioning.

Alignment is concerned with the relative orientation of arrays with respect to a common index space. If two array elements are accessed in a particular statement and they are aligned, this means that they will always be scheduled to the same processor, no matter how the data is partitioned. Alignment is concerned with maximising the number of references that are local.

Alignment has direct global impact; where a value is stored must be consistent throughout

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

Figure 3: Parallelised TRED2 Program Fragment

the multiple usages in a program. In [17] a linear algebraic solution to alignment is described. Other methods exist and can be found in [9] and [10]. In this paper, a more restricted method is used whereby only index reordering is considered. This is to simplify subsequent mapping and allow a simple translation to the graph based approach needed for determining global alignment. Later work will incorporate more analysis.

In this section alignment for creation parallelism is described. If only reduction parallelism is available then it is preferable to align with respect to one of the arrays which makes reference to the reduction iterator. Thus the form of parallelism available will have a definite impact on the appropriate alignment decisions. In this scheme alignment is made by default to the array being written for creation parallelism. Reduction alignment is ignored unless it is the sole significant part of a program. When mapping, the type of parallelism will obviously determine the code produced.

3.2.1 Significant Regions

To reduce the complexity of deciding the overall alignment, only the most "significant" parts of the program are examined. These are those sections where the most communication and computation is expected and correspond to the biggest loop nests and largest arrays. Static estimates based on control-flow information is used in the presence of conditionals to determine the significant regions. A global alignment graph is created where each arc corresponds to two indices wishing to be aligned. Rather than placing weights on a complete data alignment graph as in [5], only significant regions are considered.

Figure 4 shows the significant regions of TRED2, where the alignment preferences for each statement may be determined. For example in statement 18, *GG* makes reference to iterator *J* as does the second index of *Z*. At present the array references have the following form where the first index of *Z* is aligned with *GG*

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_{GG} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}_{Z} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}_{D}$$

GG and *D* have all zeros in their second row as they are only one dimensional arrays. It is desireable to swap the first and second index of *Z* so that *GG* and *Z* will be perfectly aligned:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_{GG} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_{Z}$$

The transformation that performs this is the permutation matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}_{Z} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_{Z}$$

This is equivalent to re-writing statement 18 in the form

```
18             GG(J) = GG(J) +Z(J,K)*D(K)
```

Therefore the prefered alignment of *Z* in statement 18 is is represented by the permutation matrix:

```
  IF (N .NE. 1) THEN
    DO II = 2, n
      IF (II .LE. N-1) THEN
        IF (SCALE .EQ. 0) THEN
        ELSE
          DO J = 1, N+1-II
            DO K = J+1 ,N+1-II
16            E(K) = E(K)+Z(K,J)*D(J)
            ENDDO
          ENDDO
          DO J = 1, N+1-II
            DO K = J+1 ,N+1-II
18            GG(J) = GG(J) +Z(K,J)*D(K)
            ENDDO
          ENDDO
          DO J = 1,N+1-II
            DO K = J,N+1-II
23            Z(K,J) = Z(K,J)-D(J)*E(K)-E(J)*D(K)
            ENDDO
          ENDDO
        ENDIF
      ELSE
     ENDIF
    ENDDO
    DO I = 2,n
      IF (D(I) .NE. 0) THEN
        DO J = 1,I-1
          DO K = 1, I-1
36          GG(J) = GG(J) +Z(K,I)*Z(K,J)
          ENDDO
        ENDDO
        DO J = 1,I-1
          DO K = 1,I-1
37          Z(K,J) = Z(K,J) -GG(J)*D(K)
          ENDDO
        ENDDO
      ENDIF
    ENDDO
  ENDIF
```

Figure 4: Significant Regions of TRED2

18: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_Z$

As D does not make a reference to J, no transformation will improve its alignment, and in statement 18, it has no preferred alignment. The remaining preferred alignments can be represented as follows:

16: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_Z$

23 : $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_D \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_E \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_E \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_D$

36: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_Z$

37: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{GG} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_D$

After applying a variant of the index domain alignment algorithm [10], the following global alignment was determined

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_E \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_Z \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_D \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}_{GG}$

All alignment is made with respect to a common index space which, when projected across processor space, determines which indices are partitioned.

4 Partitioning at the crux

Each of the costs addressed at this stage implies a particular data partition. The relative significance of these costs will determine a particular partitioning. In general there is a trade-off, each cost will suggest a conflicting data layout. In this paper, only inter-processor costs are considered at this stage, i.e. load balancing and inter-processor communication or non-local access. It is argued that costs, such as local cache usage or instruction scheduling, can be considered later. It is important to have a separation of concerns wherever possible so as to reduce complexity. However the availability of certain post partition transformations may have an effect on partitioning and must be considered where appropriate.

Firstly parallelism is considered; in particular, those array dimensions that can be evaluated in parallel are determined. The issue of load balancing is then addressed and how the data should be partitioned so as to minimise load imbalance. Next, using information gained from data alignment, a method to reduce communication by internalising as many memory references as possible is described. Finally, the last method tries to determine the best data partition based on the assumption that data re-use will take place **after** partitioning so as to reduce communication. Once each analysis stage has described a data partitioning strategy, a global algorithm is proposed. The usefulness of such an algorithm is tested by experiment in section eight.

Once the relative alignment between arrays is determined, the arrays must be mapped to processor space. At this stage only orthogonal partitions of the arrays are considered. Thus if an array has N dimensions, it may be partitioned in $2^N - 1$ different ways. If possible, the partitioning of data should be along those dimensions of the array that may be evaluated in parallel and minimise overhead. It is possible to describe partitioning in a linear algebraic manner. Partitioning an array into k dimensions is the act of projecting an $N \times m$ index matrix so that there are k non-zero rows in the new index matrix, where each of the k rows corresponds to that dimension of the array being distributed over the p processors and the remaining $N-k$ rows are serialised.

The particular form of this transformations is chosen to be $N \times N$ integer matrices $\mathcal{P}$ and $\mathcal{S}$ respectively, i.e.

$$\mathcal{P} \times \mathcal{I} = \overline{\mathcal{I}} \tag{15}$$

$$\mathcal{S} \times \mathcal{I} = \underline{\mathcal{I}} \tag{16}$$

where $\mathcal{P}$ is the partitioning matrix, $\mathcal{S}$ the serialising matrix, and $\overline{\mathcal{I}}, \underline{\mathcal{I}}$ the partitioned and serialised indices of an array.

$$\mathcal{P} + \mathcal{S} = I \tag{17}$$

where I is the identity matrix. The rows of $\mathcal{P}$ are either rows of identity or rows of the null matrix. After pre-multiplying the matrix, the new partitioned matrix has both zero and non-zero rows. Non-zero rows are those that are to be partitioned.

Problems occur when the arrays to be partitioned have a different number of dimensions. For those arrays of a smaller dimension, care must be taken to ensure that when they are partitioned they are distributed across the processors rather than mapped to just one processor. For more details see [13].

4.1 Parallelism

Partitioning for parallelism is based on identifying which array indices can be evaluated in parallel. The parallel iterators surrounding a statement must be identified; this information is readily available from the parallelisation phase. Those indices that are accessed by parallel iterators are candidates for partitioning. This can be formalised as follows

$$\mathcal{P}_i = \begin{cases} e_i^T & \exists k, t \,|\, \mathcal{U}_{i,k}^t \neq 0 \wedge J_k \in ParJ \\ 0 & otherwise \end{cases} \quad \forall i \in 1, \ldots, N \tag{18}$$

where $ParJ$ are those iterators surrounding a statement that are parallel and e_i^T is ith row of the identity matrix [13]. In other words partition along those indices where there exists a reference to a parallel iterator.

For example in figure 5 where E, D are aligned to the first and second indices of Z respectively the partitioning matrix will be of the form:

$$\mathcal{P} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{19}$$

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

Figure 5: Parallel Loop Example

In this fragment both iterators j and k are parallel and appear in both index dimensions, therefore either or both indices may be partitioned for parallelism giving the following useful set of candidate partitions.

$$\mathcal{P} = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right\} \tag{20}$$

This may be simply represented as $\mathcal{P} = \{1, 2, 1 \wedge 2\}$

4.2 Load Balance

The minimisation of load imbalance is considered at two stages in the compilation process. In the partitioning phase those iterators that are "work invariant" are looked for; if the partitioning takes place with respect to these iterators the same amount of work will take place in each processor. If it is eventually decided to partition by a non-invariant iterator, then the iteration space can be interleaved to give better load balancing.

A parallel iterator is work invariant if it does not refer to another iterator in its bounds and is not referenced in the loop bounds of another iterator or in a conditional expression. It is possible to transform for invariancy [16] but as this may skew the data space it is not attempted in this scheme as it may increase the complexity at later stages.

Invariant iterators suggest the index(indices) of an array that later partitioning transformations should decompose by, so as to ensure a load balanced implementation.

This can be expressed formally as follows

$$\mathcal{P}_i = \begin{cases} e_i^T & \exists k, t \mid \mathcal{U}_{i,k}^t \neq 0 \wedge J_k \in LBJ \\ 0 & otherwise \end{cases} \tag{21}$$

where LBJ are those iterators surrounding a statement that are load balanced. Using the notation earlier introduced in section 2, the load balanced criteria can be expressed formally as:

$$LBJ = \left\{ J_i \mid e_i^T L = e_i^T U = e_i^T \wedge Le_i = Ue_i = e_i \wedge e_i^T \mathcal{E} = 0 \wedge J_i \in ParJ \right\} \tag{22}$$

This complex term essentially states that an iterator is load balanced if firstly no other iterator depends upon it, secondly that it does not depend on any other iterator and finally that no conditional expressions depend on it and it is parallel.

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,I-1
      C(I,J) = C(I,J) + A(K) * B(I-K)
    ENDDO
  ENDDO
ENDDO
```

Figure 6: Load Balance Example

For example in figure 6, only iterator J satisfies the criteria of being parallel and work invariant. As J appears in the second dimension of the C array, partitioning should be along this index. The partition matrix is given by:

$$\mathcal{P} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{23}$$

4.3 Alignment

If two arrays are aligned with respect to a particular index, then no matter how those individual array elements are partitioned, any reference with respect to this index will always be local. Partitioning based on alignment tries to maximise the amount of local accesses by picking those indices where there is maximum alignment. A simple measure of alignment is the number of corresponding non-zero rows that are equal.

Let $\delta(x, y)$ be defined as follows:

$$\delta(x, y) = \begin{cases} 1 & x = y \wedge x \neq 0 \\ 0 & otherwise \end{cases} \tag{24}$$

This function determines whether two subscripts are non-zero and equal.

The function $H(i)$ defined as:

$$H(i) = \sum_t \delta(\mathcal{U}_i^0, \mathcal{U}_i^t) \tag{25}$$

measures how well a particular index is aligned.

For each index the value of H is calculated, the index (indices) with the highest value being the one(s) to partition along. For example consider the example in figure 7:

In this example, $H(1) = 3$ and $H(2) = 2$ and therefore partitioning should be along the first index as it is the most aligned.

If a particular index of the arrays is the most aligned ($MaxH$) let the corresponding row of $\mathcal{P}$ be the relevant row of Identity, otherwise set it equal to zero i.e.

$$\mathcal{P}_i = \begin{cases} e_i^T & H(i) = MaxH \\ 0 & otherwise \end{cases} \tag{26}$$

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,I-1
      C(I,J) = C(I,J) +A(I,K) * B(I,J)
    ENDDO
  ENDDO
ENDDO
```

Figure 7: Alignment Example

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,N
      C(I,J) = C(I,J) + A(J,K) *B(I,I)
    ENDDO
  ENDDO
ENDDO
```

Figure 8: Region of Access Example

This gives the following partition matrix for this example.

$$\mathcal{P} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{27}$$

4.4 Region of Access

In this section, a partitioning analysis based on the region of access is described. The region of access corresponds to the size of the region of the data accessed rather than the number of times the data is accessed. It is the role of this strategy to determine which dimension the arrays are to be partitioned across so as to minimise non-local access.

This partitioning scheme depends on data re-use to remove redundant non-local accesses. Essentially it searches through several possible data partitions and estimates the amount of non-local data that will be accessed for that partition, based on the assumption that once a data item has been accessed it remains local as long as it is required.

Using this method, for instance, would suggest partitioning by blocks (rows and columns) in matrix multiplication which gives the best hand-coded implementation.

The amount of an array v accessed by a processor, when compiling for creation parallelism, is the number of points described by $\mathcal{U}^v\underline{J}$ with respect to the local iteration space $\underline{J}$. The amount of non-local access is the total access minus that which is local; this scheme tries to find a partition that minimises this amount.

For example consider the program in figure 8; if it is decided to partition by the first dimension of the aligned arrays, as partitioning based on alignment would suggest, then the size of the region of access to arrays A and B will be:

$$||\mathcal{U}^A\underline{J}|| = n^2 \tag{28}$$

$$||\mathcal{U}^B\underline{J}|| = \frac{n}{p} \tag{29}$$

where p is the number of processors. If, instead, it is decided to partition by the second dimension of the aligned arrays, the region of access will be:

$$||\mathcal{U}^A\underline{J}|| = \frac{n^2}{p} \tag{30}$$

$$||\mathcal{U}^B\underline{J}|| = n \tag{31}$$

The region of A accessed (entire array) is greater than that of B (diagonal). Choosing the second index to partition along, has the effect of reducing the local range of the J iterator and as J appears in the subscripts of array A, the reduction in the range of J implies that there will be a smaller amount of access to A. The I iterator will have a smaller range if C is partitioned by the first index, but as I only appears in the subscripts of array B this has little effect on the total access.

In general, $\underline{\mathcal{I}}$ are the local elements of an array v so partitioning should minimise:

$$||\mathcal{U}^v\underline{J} - \underline{\mathcal{I}}|| \tag{32}$$

The amount of access is a function of the processor, so the maximum value over the processors should be minimised. This should be carried out for all array references in that particular statement. Therefore partitioning should minimise:

$$\sum_{r=1}^{\theta-1} \max_{z\in P}(||\mathcal{U}^r\underline{J} - \underline{\mathcal{I}}^r||) \tag{33}$$

where P is the set of processors and z an element of P.

If there are N indices along which the data may be partitioned, then there are $2^N - 1$ possible data partitions available. In general the calculation of $||\mathcal{U}^v\underline{J}||$ is very expensive and a partitioning decision is made based on the following observations:

1. The region of access is related to the rank of the subscript matrices. The higher the rank, the greater the number of different data points accessed. For example in the program in figure 8, there are n^2 accesses to A but only n accesses to B. Therefore partitioning is only with respect to the highest rank array references as they will dominate the non-local access cost.

2. If an iterator is referenced in an array subscript , then partitioning with respect to that iterator will reduce the volume of access to that array . Therefore partitioning by all those iterators that are common to the left hand and right hand sides of a statement will be beneficial.

```
Determine the sets P, R, A, L
if |P| ≥ 1 then
   if |P ∩ R| ≥ 1 then
      if |P ∩ R ∩ A| ≥ 1 then
         if |P ∩ R ∩ A ∩ L| ≥ 1 then
            select (P ∩ R ∩ A ∩ L)
         else select (P ∩ R ∩ A)
      else if |P ∩ R ∩ L| ≥ 1 then
               select (P ∩ R ∩ L)
            else select (P ∩ R)
   else if |P ∩ A| ≥ 1 then
            if |P ∩ A ∩ L| ≥ 1 then
               select (P ∩ A ∩ L)
            else select (P ∩ A)
         else if |P ∩ L| ≥ 1 then
                  select (P ∩ L)
               else select (P)
else no partition decision
```

Figure 9: Data Partition Algorithm

This can be expressed formally as

$$\mathcal{P}_i = \begin{cases} e_i^T & \exists j, x, y, t | \mathcal{U}^0_{i,j} = \mathcal{U}^t_{x,y} \wedge rank(\mathcal{U}) = \mathcal{R} \wedge J_j \in ParJ \\ 0 & otherwise \end{cases} \tag{34}$$

In other words, if a referenced array is of highest rank $\mathcal{R}$ then partition by those indices where the subscripts are common to the array being written.

Partitioning so as to minimise the region of access coupled with later data re-use transformations is equivalent to the "blocking" strategy used by numerical analysts and compiler writers [3][27]. The blocking transformation has been broken into two parts, the region of access partitioning and the post -partitioning data re-use transformations.

4.5 Global Data Partitioning

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 the information provided by the previous analysis a global partitioning strategy is constructed. The indices must be parallel and, as far as as interprocessor costs are concerned, it is assumed that minimising communication is more important than load imbalance; this is based on the assumption that a non-local access costs more than one computation assuming no latency

Stat.	Para	Load	Align	Region
16	1,2,1∧2	∅	1	1
18	1,2,1∧2	∅	2	2
23	1,2,1∧2	∅	1	1∧2
36	1,2,1∧2	∅	2	2
37	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

Figure 10: Partition decisions for each analysis

tolerance. This is simplistic and the dominant cost may vary. More accurate information on the actual amount of communication/computation may give a more accurate partitioning decision. However, this would be extremely expensive and is therefore rejected in this scheme.

The eventual data partition is based on the preceding analysis. Each form of 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. P is the set of the most popular partitions if the parallelism analysis is applied, A is the set based on alignment, L on load balance and R on the region of access.

Initially the parallelism criterion is used to judge the best data partition. If it does not select any unique partition, i.e. there are more than one equally good candidates, then further analysis is used. As data re-use post partitioning transformations are available, the region of access analysis is considered to be the most useful in minimising communication. If, however, it does not select a unique partition, then the alignment analysis is used. Again if there are multiple candidates, load balancing is used.

The full algorithm is given in figure 9 where each conditional tests whether further analysis is required to make a decision. The function *select* chooses the common partition and, if there is more than one, selects one arbitrarily. Alternatively, further analysis on less significant regions could be applied to resolve any decision, but it is doubtful if this is worth the cost.

The table in figure 10 shows the results of applying each form of analysis for the TRED2 code based on the significant regions as shown in figure 4. As far as parallelism is concerned, all partitions were equally acceptable i.e. $P = \{1, 2, 1 \wedge 2\}$. The load balance analysis found no suitable iterators i.e. $L = \emptyset$. Alignment preferred either the first index or the second index but not both i.e. $A = \{1, 2\}$. Finally the region of access preferred index 2 or indices 1 and 2 i.e. $R = \{2, 1 \wedge 2\}$

Applying the algorithm to the TRED2 example gives $(P \cap R \cap A) = \{2\}$, in other words the data should be partitioned along the second index.

It is interesting to note that none of the analyses gave a unique choice, implying that certain sections have conflicting requirements. Other programs will have different behaviours.

```
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
```

Figure 11: Before Partitioning

```
DO J = ZLO,ZHI
  DO K = J,N+1-II
    Z(K,J) = Z(K,J)-D(J)*Get(E(K))-E(J)*Get(D(K))
  ENDDO
ENDDO
DO J = DLO,DHI
  IF (J .LE. N+1-II) THEN
    D(J) = Z(N+1-II,J)
  ENDIF
ENDDO
```

Figure 12: After Partitioning

Once the global partition has been decided it is necessary to generate the local node program. The remaining sections describe that generation and subsequent post-partitioning transformations .

4.6 Mapping

In this section only a brief outline of the local program generation is given. For more detail refer to [14]. The form of the local program will depend upon the data partition, in particular the local iterator bounds will depend on the local array bounds given an only-local write implementation. Essentially the range of the iterators is limited by the bounds of the local arrays.

Rectangular loop bounds are relatively easy to partition, but not all iteration spaces are rectangular. However it is possible to express all iterations spaces as rectangular ones which are cut by hyperplanes, where the hyperplanes correspond to **if** conditions. Once this is performed the local iterator bounds can be calculated. For example consider the example in figure 11, where Z is to be partitioned along its second index and hence J will be partitioned. D is aligned and will be similarly partitioned.

After removing redundant constraints ($J \geq 1, J \leq N + 1 - II$) in the first loop nest, the program in figure 12 is generated. Here xLO and xHI are the local lower and upper bounds of

```
DO J = ZLO,ZHI
  DO K = #J,N+1-II
    Z(K,#J) = Z(K,#J)-D(#J)*Get(E(K))-E(#J)*Get(D(K))
  ENDDO
ENDDO
DO J = DLO,DHI
  IF (#J .LE. N+1-II) THEN
    D(#J) = Z(N+1-II,#J)
  ENDIF
ENDDO
```

Figure 13: Interleaving J

array data. The Get function prefaces any potentially non-local array access. All data that is perfectly aligned on a particular index is local and does not require special treatment. In this example the second index of Z has been partitioned so references to $D(J)$ and $E(J)$ are local but $D(K)$ and $E(K)$ are non-local. The implementation of Get will depend on the access model of the architecture and is not considered at this stage.

5 Post Partition Transformations

5.1 Interleaving

If the partitioned index is not load balanced, then interleaving will help spread the work load more evenly [12]. Most methods require many guard conditionals and can destroy locality. The method used here interleaves the iteration space such that local writes are preserved and the local iteration space remains convex. This is very important as the introduction of non-linearity in the iteration space would require a different compilation model in subsequent transformations.

Only a very brief explanation of interleaving is given in this section. For more information see [18] where the interleave transformation is described in detail. Interleaving an iterator, using the interleave function #, implies that every reference to that iterator is interleaved. This includes array references, conditionals and general usage in expressions. Although the global iteration space is no longer convex each **local** iteration space after partitioning is in fact convex and it can be shown [18] that the interleave function can be expressed as a linear function:

$$\#\underline{j} = z + p(\underline{j} - \underline{jlo}) \tag{35}$$

where $\underline{j}$ is the partitioned iterator, z is the processor number, p is the total number of processors and $\underline{jlo}$ is the lower bound of the iterator. For example in the TRED2 extract in figure 12 Z has been partitioned by the second index and if it is to be interleaved the program in figure 13 is produced.

```
DO J = ZLO,ZHI
  DO K = #J,N+1-II
    Z(K,J) = Z(K,J)-D(J)*Get(E(##K))-E(J)*Get(D(##K))
  ENDDO
ENDDO
DO J = DLO,DHI
  IF (#J .LE. N+1-II) THEN
    D(J) = Z(N+1-II,J)
  ENDIF
ENDDO
```

Figure 14: After Inverse Interleaving

At this stage it is not obvious whether the data to be written to is local, but it can be restored to a only-local write form by applying the inverse of #, defined as ##, to the interleaved **indices** of the aligned arrays. This has a similar effect to that of the alignment, in that it has to be propagated throughout the remainder of the program. On applying ## we have the program shown in figure 14:

Although the #,## functions appear expensive especially when used to reference array data, they are amenable to strength reduction and common expression elimination, reducing their cost.

5.2 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. By loop skewing and re-ordering such that the invariant iterators are innermost and by storing access in local temporaries, invariancy and hence re-use can be exploited. Multiple array references in a statement imply different loop orders which can be overcome by scalar expansion of temporaries, but the size of these temporaries is limited because of scalability constraints. Data dependency can also prevent loop re-ordering [24]. For a fuller description of this method see [15]. 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. In other words, only those array references preceded by the Get function are considered. For example, consider the program fragment shown in figure 15.

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 figure 16 is derived. This reduces the number of non-local array accesses by $2 \times (ZHI - ZLO + 1)$. The applicability of the region of access analysis depends on the ability to exploit data re-use and therefore this post-partition transformation is

```
DO J = ZLO,ZHI
  DO K = J,N+1-II
    Z(K,J) = Z(K,J)-D(J)*Get(E(K))-E(J)*Get(D(K))
  ENDDO
ENDDO
```

Figure 15: Data Re-use Example

```
DO K = ZLO,N+1-II
  TD = Get(D(K))
  TE = Get(E(K))
  DO J = ZLO,ZHI
    IF (J .LE. K) THEN
      Z(K,J) = Z(K,J)-D(J)*TE-E(J)*TD
    ENDIF
  ENDDO
ENDDO
```

Figure 16: Post Data Re-use Transformation

integral to the partitioning algorithm described in this paper.

6 Memory Access Transformations

The target architecture reported in the experimental section does not support a message passing model but has a single address space and local cache. Many of the transformations needed to better utilise memory appear in both compiling for message passing and single address based machines. It is beyond the scope of this paper to describe those used in message passing implementations but for a good survey and analysis see [6]. One of the most important consequences of a single address machine is that data references proceeded by the Get function require no further translation as the same mechanism is used to address local and non-local memory.

Program transformations to improve cache utilisation have been extensively studied. Essentially temporal locality occurring from multiple access to the same element within a cache line and spatial locality found in accessing consecutive elements in a cache line are both exploited by a combination of strip-mining and loop interchange [8] [27]. Various algorithms determining the exact form of these transformations have been proposed but in this paper only loop interchange to allow unit stride access has been considered. Only perfectly nested loops are analysed as, in general, reordering will undo non-local data re-use transformations. For example in the previous example 16, it is desirable to have iterator K innermost assuming a column major storage but this would destroy the re-use of $D(K)$ and $E(K)$. Clearly there is a tradeoff, but reducing non-local access is considered more important than utilising local cache.

Carr in his thesis [2] explores in great detail the relative merits of memory management enhancing transformations for cache based uni-processors. It is hoped that in future his work will be integrated into this compiler strategy.

7 Synchronisation

Synchronisation can be a significant overhead in parallel programs. It should be proportional to the amount of communication but often naive analysis leads to the insertion of excess synchronisation points. This section outlines a method that tries to reduce synchronisation by exploiting accurate data dependence information and features of the SPMD model of computation.

Barrier synchronisation is used so as to ensure that data dependence is honoured. The method to reduce synchronisation is as follows. Firstly barrier synchronisations are placed around any sequential iterators which reduces the number of dependence arcs due to cross iteration data dependence. Secondly, only flow and anti-dependences are considered. All output dependence arcs are removed as the SPMD model implies that all writes are local and therefore do not require explicit ordering via synchronisation as they are implicitly ordered by von Neumann execution. Thirdly all true or anti-dependences which are completely local are also removed. This is trivially determined by observing if the partitioned indices are completely aligned or not.

Finally all dependence arcs killed by covering dependences are removed [20]. If a naive synchronisation method whereby a barrier is placed at the end of every parallel loop nest were employed then the number of synchronisation points for the TRED2 code would be 36. By applying the techniques described above, this is reduced to 14.

To illustrate this point consider the TRED2 segment in figure 17. The only synchronisation point required is following the final ENDDO as there is 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 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.

8 Example and Results

This section tests the applicability of the data partitioning algorithm by comparing its performance against an existing parallelising compiler and an experienced programmer.

The original TRED2 code (original) is first given to the KSR 1 KAP compiler which tries to detect parallelism and map it to the machine. Secondly, KAP is again used but this time with a "parallelised" loop distributed version of the code (parallel). The compiler strategy described in this paper is applied next, with and without interleaving (interleave, folded) and is compared to the other methods. These are compared against a implementation written by an expert programmer of the KSR 1 (hand coded). The implementations were tested on the KSR with the data sizes $N = 256, N = 512$ and compared across 1 to 16 processors.

```
          DO J = 1, N+1-II
            DO K = J+1 ,N+1-II
S1            E(K) = E(K)+Z(K,J)*D(J)
            ENDDO
          ENDDO
          DO J = 1, N+1-II
S2          GG(J) = E(J)+Z(J,J)*D(J)
          ENDDO
          DO J = 1, N+1-II
            DO K = J+1 ,N+1-II
S3            GG(J) = GG(J)+Z(K,J)*D(K)
            ENDDO
          ENDDO
          DO J = 1, N+1-II
S4          E(J) = GG(J)
          ENDDO
S5        G = GG(N+1-II)
```

Figure 17: Synchronisation Example

In figure 18 KAP, given the original TRED2 code, fails to achieve any speed up and only manages a modest decrease in parallel time when given a "parallel" version. In the second figure 19, poor performance is also noted for the original code but a small improvement has occurred for the "parallel" one. In both cases only 8 processors could be tested due to a problem with the runtime system.

The hand coded version performs best of all on both data sizes, being efficient on one processor and giving a steady reduction in parallel time as the number of processors is increased. The versions generated by the strategy proposed in this paper behave poorly on one processor. For 2 or more processors they outperform KAP and approach hand-coded performance for 8 or more processors. Interleaving seems to have a greater impact as the number of processors is increased.

The relatively high single processor execution time is due to the particular implementation of synchronisation primitives. These could be made more efficient by testing for the special case of one processor, but it is not our intention to focus attention on improving the performance on one processor implementations.

These results show that the data partition algorithm gives performance comparable to hand coded methods showing that data partitioning can be automated. Furthermore it outperforms the existing parallelising compiler for the KSR 1.

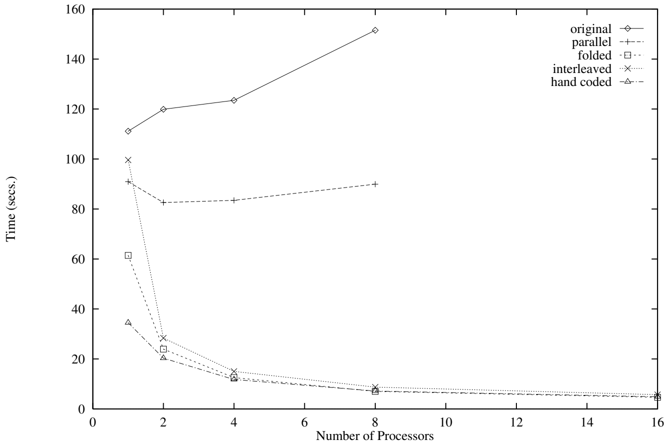

Figure 18: Data Size N=256

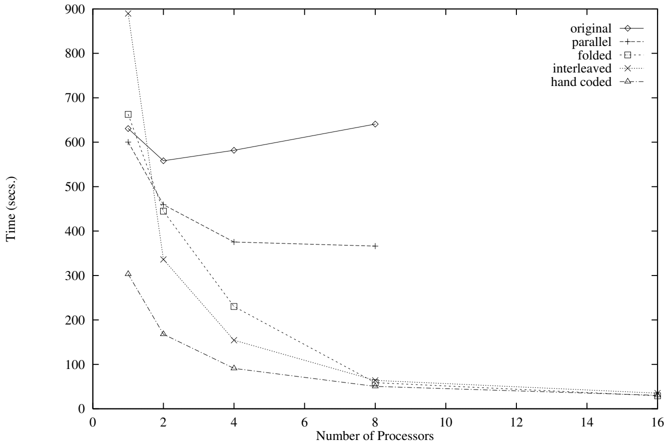

Figure 19: Data Size N =512

9 Conclusions

This paper has described an automatic data partition on algorithm which is based on four different analysis techniques. By determining the relative merit of each form of analysis, a data partitioning decision is made. By integrating this algorithm into an overall compilation strategy, effective use can be made of any data partitioning decision. Furthermore it is shown that the SPMD model of computation used can reduce the number of synchronisations necessary in a program. This strategy was tested for a real program on the KSR where it outperformed the native parallelising compiler and approached hand coded performance for 8 or more processors.

Acknowledgements

I would like to thank Gholam Hedayat, for his help and insight in the preparation of this paper and Mark Bull for implementing the TRED2 code on the KSR 1. This work is supported by an SERC Postdoctoral Research Fellowship.

References

[1] Benkner S., Chapman B. and Zima H.,**"Vienna Fortran 90"** Proceedings of the 1992 Scalable High Performance Computing Conference, April 1992.

[2] Carr S., **"Memory-Hierarchy Management"**, Rice University, Department of Computer Science, Technical Report TR92-190, October 1992.

[3] Carr S. and Kennedy K., **"Compiler Blockability of Numerical Algorithms"**, Supercomputing '92, November 1992.

[4] Fox G., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C-W. and Wu M-Y., **"FORTRAN D Language Specification"**, Rice COMP TR90-141, Department of Computer Science, Rice University, February 1991.

[5] 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, September 1992.

[6] Hiranandani S., Kennedy K. and Tseng C.-W., **"Evaluation of Compiler Optimisations for Fortran D on MIMD Distributed-Memory Machines"**, International Conference on Supercomputing, July 1992.

[7] Kennedy K. and Kremer U., **"Automatic Data Alignment and Distribution For Loosely Synchronous Problems in an Interactive Programming Environment"**, Technical Report TR91-155, Rice University, April 1991.

[8] Kennedy K. and McKinley K.S., **"Optimizing for Parallelism and Data Locality"**, International Conference on Supercomputing, July 1992.

[9] Knobe K., Lukas J.D., and Steele G.L., **"Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines"**, Journal of Parallel and Distributed Computing 8, pp102-118, 1990.

[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, October 1990.

[11] Lu L-C., **"A Unified Framework for Systematic Loop Transformations"**, 3rd ACM Symposium on the Principles and Practice of Parallel Programming, April 1991.

[12] Nicol D.M. and Saltz J.H., **"An Analysis of Scatter Decomposition"**, IEEE Transactions on Computers, Vol 39, No 11 pp 1337-1345, November 1990.

[13] O'Boyle M.F.P., **"Program and Data Transformations for Efficient Execution of Array Computation on Distributed Memory Architectures"**, Technical Report UMCS-93-1-6, University of Manchester, Dept. of Computer Science, January 1992.

[14] O'Boyle M.F.P., **"A Transformational Approach to Compiling Sisal for Distributed Memory Architectures"**, International Conference on Supercomputing, July 1992.

[15] O'Boyle M.F.P. and Hedayat G.A., **"A New Program Transformation to Minimise Communication in Distributed Memory Architectures"**, PARLE'92, Paris, June 1992.

[16] O'Boyle M.F.P. and Hedayat G.A., **"Load Balancing of Parallel Affine Loops by Unimodular Transformations"**, European Workshop on Parallel Computing, Barcelona, March 1992.

[17] O'Boyle M.F.P. and Hedayat G.A., **"Data Alignment: Transformations to Reduce Communication on Distributed Memory Architectures"**, Scalable High Performance Computing Conference, Williamsburg, April 1992.

[18] O'Boyle M.F.P., **"Interleaving: An Affinity Based Schedule for Load Balancing on NUMAs"** (in preparation)

[19] Pugh W., **"Uniform Techniques for Loop Optimisation"**, International Conference on Supercomputing, June 1991.

[20] Pugh W. and Wonnacott D., **"Eliminating False Data Dependences using the Omega Test"**, ACM SIGPLAN Programming Languages Design and Implementation, 1992.

[21] Kelly W. and Pugh W., **"Generating Schedules and Code within a Unified Reordering Transformation Framework"**, University of Maryland Technical Report UMIACS-TR-92-126, November 1992.

[22] Wholey S., **"Automatic Data Mapping for Distributed-Memory Parallel Computers"**, International Conference on Supercomputing, ACM Press, July 1992.

[23] Watts T.M., Soffa M.L. and Gupta R., **"Techniques for Integrating Parallelizing Transformations and Compiler-Based Scheduling Methods"**, Supercomputing '92, November 1992.

[24] Wolfe M., **"Optimising Supercompilers for Supercomputers"**, Research Monograph in Parallel and Distributed Computing, Pitman, London, 1989.

[25] Wolf M.E. and Lam M., **"Maximizing Parallelism via Loop Transformation "**, Third Workshop on Languages and Compilers for Parallel Computing. August 1990.

[26] 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.

[27] Wolf M.E. and Lam M., **"A Data Locality Optimizing Algorithm"**, ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1991.

Appendix

TRED2: Original Version

```
      PROGRAM TRED2
      IMPLICIT NONE
      Integer N
      Parameter (n =10)
      DOUBLE PRECISION Z(N,N)
      DOUBLE PRECISION A(N,N), D(N), E(N)
      DOUBLE PRECISION  H,SCALE,F,G,HH
      INTEGER I, J, K,L,II,JP1

      DO 5 I = 1,N
         DO 3 J = 1,N
 3        Z(J,I) = A(J,I)
         D(I) = A(N,I)
 5    CONTINUE
      IF (N .EQ. 1) GO TO  82
      DO 63 II = 2,N
         I = N+2- II
         L = I-1
         H = 0.0D0
         SCALE = 0.0D0
         IF (L .LT. 2) GO TO  16
         DO 14  K = 1,L
 14        SCALE = SCALE + DABS(D(K))
         IF (SCALE .NE. 0.0D0) GO TO  23
 16      E(I) = D(L)
         DO 21 J = 1,L
           D(J) = Z(L,J)
           Z(I,J) = 0.D0
           Z(J,I) = 0.D0
 21      CONTINUE
         GO TO 62
 23      DO 25 K = 1,L
           D(K) = D(K)/SCALE
 25        H = H + D(K)*D(K)
```

```
          CONTINUE
          F = D(L)
          G = -DSIGN(DSQRT(H),F)
          E(I) = SCALE * G
          H = H-F*G
          D(L) = F-G
          DO 33 J = 1,L
33          E(J) = 0.D0
          DO 45 J = 1,L
            F = D(J)
            Z(J,I) = F
            G = E(J) + Z(J,J)*F
            JP1 = J +1
            IF (L .LT. JP1) GO TO 44
            DO  43 K = JP1 , L
              G = G +Z(K,J)*D(K)
              E(K) = E(K) +Z(K,J)*F
43          CONTINUE
44          E(J) = G
45        CONTINUE
          F = 0.0D0
          DO 50 J = 1,L
            E(J)= E(J)/H
            F = F+ E(J)*D(J)
50        CONTINUE
          HH = F/(H+H)
          DO 53 J = 1,L
53          E(J)= E(J)-HH*D(K)
          DO 61 J = 1,L
            F = D(J)
            G = E(J)
            DO 58 K = J,L
58             Z(K,J)=Z(K,J)-F*E(K)-G*D(K)
            D(J) = Z(L,J)
            Z(I,J)= 0.D0
61        CONTINUE
62        D(I)=H
63     CONTINUE
       DO 81 I = 2,N
         L = I - 1
         Z(N,L)=Z(L,L)
         Z(L,L) = 1.0D0
         H = D(I)
         IF (H .EQ.0.0D0) GO TO 79
         DO 71 K = 1,L
71          D(K)= Z(K,I)/H
```

```
         DO 78 J = 1,L
            G = 0.0D0
            DO 75 K = 1,L
75             G =G +Z(K,I)*Z(K,J)
            DO 77 K = 1,L
77             Z(K,J) = Z(K,J) - G *D(K)
78       CONTINUE
79       DO 80 K =1,L
80          Z(K,I) = 0.0D0
81    CONTINUE
82    DO 85 I = 1,N
        D(I) = Z(N,I)
        Z(N,I) = 0.0D0
85    CONTINUE
      Z(N,N) = 1.0D0
      E(1) = 0.0D0
      END
```

TRED2: Parallel Version

```
      PROGRAM TRED2
      IMPLICIT NONE
      INTEGER N
      PARAMETER (N =10)
      DOUBLE PRECISION Z(N,N)
      DOUBLE PRECISION A(N,N), D(N), E(N),GG(N)
      DOUBLE PRECISION  H,SCALE,F,G,HH
      INTEGER I, J, K,L,II

      Do i = 1,n
        Do j = 1,n
          z(j,i) = a(j,i)
        EndDo
      EndDo
      Do i = 1,n
        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 + abs(d(k))
           EndDo
           if (scale = 0.0D0) then
```

```
    e(n+2-ii) = d(n+1-ii)
    Do j = 1, n+1-ii
      d(j) = z(n+1-ii,j)
    EndDo
    Do j = 1, n+1-ii
      z(n+2-ii,j) = 0.0D0
    EndDo
    Do j = 1, n+1-ii
      z(j,n+2-ii) = 0.0D0
    EndDo
  else
    Do k = 1, n+1-ii
      d(k) = d(k)/scale
    EndDo
    Do k = 1, n+1-ii
      h = h +d(k)*d(k)
    EndDo
    g = -sign(sqrt(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)
    EndDo
    Do j = 1, n+1-ii
      Do k = j+1 ,n+1-ii
        e(k) = e(k)+z(k,j)*d(j)
      EndDo
    EndDo
    Do j = 1, n+1-ii
      gg(j) = e(j) +z(j,j)*d(j)
    EndDo
    Do j = 1, n+1-ii
      Do k = j+1 ,n+1-ii
        gg(j) = gg(j) +z(k,j)*d(k)
      EndDo
    EndDo
    Do j = 1, n+1-ii
      e(j) = gg(j)
    EndDo
    g = g(1)
    f = 0.0D0
    Do j = 1,n+1-ii
```

```
        e(j) = e(j)/h
      EndDo
      Do j = 1,n+1-ii
        f = f+e(j)*d(j)
      EndDo
      hh = f/(h+h)
      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
      Do j = 1,n+1-ii
        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)
    EndDo
    Do j = 1, n+1-ii
      z(n+2-ii,j) = 0.0D0
    EndDo
    Do j = 1, n+1-ii
      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) = 1.0.0D0
  h = d(i)
  if (h .NE. 0.0D0) then
    Do k = 1,n+1-ii
      d(k) = z(k,n+2-ii)/h
    EndDo
    Do j = 1,i-1
      gg(j) = 0.0D0
    EndDo
    Do j = 1,i-1
      Do k = 1, i-1
        gg(j) = gg(j) +z(k,i)*z(k,j)
```

```
      EndDo
    EndDo
    Do j = 1,i-1
      Do k = 1,i-1
        z(k,j) = z(k,j) -gg(j)*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)
EndDo
Do i = 1,n
  z(n,i) = 0.0D0
EndDo
z(n,n) = 1.0D0
e(1) = 0.0D0
END
```