Proceedings of the 11th ACM International Conference on Supercomputing, ICS'97 (Vienna, July 1997), ACM Press, pp. 277–284.

Compile-Time Minimisation of Load Imbalance in Loop Nests

Rizos Sakellariou, John R. Gurd
Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, U.K.
e-mail: {rizos,john}@cs.man.ac.uk

Abstract

Parallelising compilers typically need some performance estimation capability in order to evaluate the trade-offs between different transformations. Such a capability requires sophisticated techniques for analysing the program and providing quantitative estimates to the compiler's internal cost model. Making use of techniques for symbolic evaluation of the number of iterations in a loop, this paper describes a novel compile-time scheme for partitioning loop nests in such a way that load imbalance is minimised. The scheme is based on a property of the class of *canonical* loop nests, namely that, upon partitioning into essentially equal-sized partitions along the index of the outermost loop, these can be combined in such a way as to achieve a balanced distribution of the computational load in the loop nest as-a-whole. A technique for handling non-canonical loop nests is also presented; essentially, this makes it possible to create a load-balanced partition for any loop nest which consists of loops whose bounds are linear functions of the loop indices. Experimental results on a virtual shared memory parallel computer demonstrate that the proposed scheme can achieve better performance than other compile-time schemes.

1 Introduction

In order to evaluate the performance trade-offs of different transformations, parallelising compilers are usually armed with some performance estimation capability; this issue has been addressed recently by a number of researchers [3, 7, 19]. Although the implementation details of these schemes vary, generally they attempt to identify sources of performance loss, such as load imbalance, interprocessor communication, cache misses, etc. [4, 6]. This has two implications for a parallelising compiler. Firstly, the compiler must be capable of extracting *quantitative* information from programs — since parallelising compilers usually target the parallelisation of *loop nests*, significant information lies in the number of times each loop will be executed; this can be used, for instance, to estimate the amount of work assigned to each processor, or the number of non-local accesses to data [7]. Secondly, the compiler should avoid postponing critical decisions concerning the parallelisation process until run-time, since doing so reduces the information available and, consequently, the accuracy of its performance prediction. One such critical decision is the *mapping* of loop nests onto a parallel architecture; that is, the way that the loop iterations are allocated to processors for execution.

The problem of mapping loop nests has attracted significant interest; traditionally, researchers have inclined towards run-time based schemes [8, 11, 13, 18]. Their underlying argument has been that information not available at compile-time may permit a more *balanced* distribution of the workload, especially in cases where different iterations of the parallel loop perform different amounts of work. Although the balanced distribution may be achieved at the expense of other overheads (e.g., increasing the number of memory accesses so as to balance the computational work), run-time mapping schemes seem to be preferable whenever the execution of the loop nest depends on expressions whose value is unknown at compile-time. However, in a number of cases, the resulting *pattern* of workload variation is highly regular, and it may be feasible to devise compile-time mapping schemes. This is the case, for example, when each iteration of the parallel loop performs a different amount of work as a result of enclosed loops whose execution depends on the value of the index of the parallel loop.

This theme has been investigated by Haghighat and Polychronopoulos [9, 10], who suggest that symbolic cost estimates can be used to design robust compile-time strategies for mapping loop nests. They propose *balanced chunk scheduling*, a mapping scheme for triangular perfect loop nests; the main idea is to partition the outermost loop in such a way that each processor executes the innermost loop body the same (or almost the same) number of times. However, balanced chunk scheduling can only be applied to a limited class of loop nests and, in practice, it is often more desirable that the outermost, parallel loop be distributed in equal-sized partitions; for instance, in a data parallel environment, if a triangular loop nest is followed by a rectangular loop nest in which the same arrays are involved, an unequal partitioning (in this respect) would lead to significant communication and/or load imbalance overhead.

In this paper we develop a general approach for compile-time mapping of parallel outer loops in the body of which there are inner loops whose execution depends on the outer loop index. We make use of symbolic cost estimates as a means of evaluating our strategy, and minimisation of load imbalance is the main objective. By avoiding options which would tend to increase other sources of overhead, this scheme achieves good practical results.

The remainder of the paper is structured as follows: Section 2 provides a brief background on measuring load imbalance and on symbolic counting of the number of loop iterations. Section 3 forms the main body of this paper; after first describing a strategy for mapping the class of *canonical* loop nests, we then show how to transform non-canonical loop nests into multiple canonical loop nests. This strategy is evaluated and compared with other mapping schemes in Section 4. Section 5 summarises the paper and its results.

2 Background

2.1 Load Imbalance

We define the total (computational) *workload* in a parallel code fragment (in this paper, this will always be a parallel loop nest) to be W_{tot}, distributed among p processors in such a way that each processor i, $0 \leq i < p$, is assigned a workload equal to W_i.[1] Clearly, $\sum_{i=0}^{p-1} W_i = W_{tot}$. This distribution embodies a *load imbalance*, L, given by

$$L = \max_i (W_i - \frac{W_{tot}}{p}) = W_{max} - \frac{W_{tot}}{p}, \qquad (1)$$

and a *relative load imbalance*, L_R, defined as

$$L_R = \frac{L}{W_{max}} = \frac{W_{max} - W_{tot}/p}{W_{max}} = 1 - \frac{W_{tot}}{pW_{max}}. \qquad (2)$$

where $W_{max} = \max(W_0, W_1, \ldots, W_{p-1})$. It can be easily shown that L_R takes values in the interval $[0, 1-1/p]$; values close to zero denote a small impact on performance. When, $L = L_R = 0$, that is, for all i, $W_i = W_{tot}/p$, the code exhibits *perfect load balance.*

2.2 Counting the Number of Loop Iterations

In the case of loops, an estimate for the values of W_i in (1) can be derived by considering the number of times that each part of the loop body is executed. This corresponds to the (complex) problem of enumerating the integer points of a polytope [1]. In the context of loop nests, some techniques to compute this are described in [5, 14, 17].

In general, the number of times, n, that a single statement surrounded by m loops is executed is given by:

$$n = \sum_{i_1=l_1}^{u_1} \sum_{i_2=l_2}^{u_2} \cdots \sum_{i_m=l_m}^{u_m} 1, \qquad (3)$$

where l_j, u_j are the lower and upper bounds, respectively, of the j-th loop, $1 \leq j \leq m$. If, for every loop, the loop bounds are constant, integer expressions whose values are known at compile-time, the loop nest is *rectangular* (in m-dimensions) and it is trivial to show that $n = \prod_{j=1}^{m} \sum_{i_j=l_j}^{u_j} 1$. However, in a variety of situations, the loop bounds may be non-constant (for instance, dependent on the index of an outer loop), or may contain expressions whose value is not known at compile-time. For the latter, it is important to know, when evaluating sums such as those in (3), whether $u_j \geq l_j$. In this paper, whenever this is the case, we split the loop iterations in such a way that the upper bound is always greater than or equal to the lower bound; this is discussed in Section 3.4.

[1] The *units* of workload may be the number of operations executed, or the CPU cycles needed to execute the code on some machine.

3 Partitioning for Load Balance

3.1 Rectangular Loop Nests

Based on the discussion in Section 2, the problem of partitioning the iterations of a single loop with lower bound l and upper bound u among p processors can be expressed as that of finding $l_k, u_k, 0 \leq k < p$, such that

$$\sum_{i=l}^{u} 1 = \sum_{i=l_0}^{u_0} 1 + \sum_{i=l_1}^{u_1} 1 + \ldots + \sum_{i=l_{p-1}}^{u_{p-1}} 1 = \sum_{k=0}^{p-1} \sum_{j=l_k}^{u_k} 1, \qquad (4)$$

where $l_0 = l$, $u_k = l_{k+1} - 1$, for $0 \leq k < p-1$, $u_{p-1} = u$, and, for all l_k, u_k,

$$\left\lfloor \frac{n}{p} \right\rfloor \leq \sum_{j=l_k}^{u_k} 1 \leq \left\lceil \frac{n}{p} \right\rceil, \qquad (5)$$

where $n = u - l + 1$. The following satisfy both (4) and (5):

- *Partitioning by decreasing order:*

$$l_k = l + k\lfloor n/p \rfloor + \min(n \bmod p, k), \quad k = 0, 1, 2, \ldots, p-1,$$

where the first ($n \bmod p$) partitions contain $\lceil n/p \rceil$ iterations, while the rest contain $\lfloor n/p \rfloor$.

- *Partitioning by increasing order:*

$$l_k = l + k\lfloor n/p \rfloor + \max(0, k-p+(n \bmod p)), k = 0, 1, 2, \ldots, p-1,$$

where the last ($n \bmod p$) partitions contain $\lceil n/p \rceil$ iterations, while the rest contain $\lfloor n/p \rfloor$.

If n is a multiple of p, both relations reduce to

$$l_k = l + kn/p, \quad k = 0, 1, 2, \ldots, p-1,$$

and the loop is divided into *equal partitions.* In this case, assuming that the body of the loop nest does not contain statements whose execution depends on the value of the index of the outer loop, perfect load balance is achieved. If n is not a multiple of p, then a relative load imbalance equal to $(p - n \bmod p)/(n + p - n \bmod p)$ is expected, a quantity which approaches zero if $n \gg p$.

These partitioning techniques can also be applied to the outermost loop of a loop nest, whether or not it is perfectly nested. If the bounds of the inner loops are not a function of the index of the outer loop, nor are there any conditional statements in the loop body whose execution depends on the value of the index of the outer loop, then perfect load balance may be achieved.

For perfectly nested loops, partitioning may be applied to the iterations of more than one loop at the same time. To illustrate this, assume that partitioning for p processors takes place over two loops, the first being executed n times, and the second being executed m times. Minimising L in (1) requires us to find p_1, p_2, where $p = p_1 \cdot p_2$, such that $\lceil \frac{n}{p_1} \rceil \lceil \frac{m}{p_2} \rceil$ is a minimum. Instead of solving this, it may be preferable to apply loop coalescing [12] before partitioning; since the inequality $\lceil nm/(p_1p_2) \rceil \leq \lceil n/p_1 \rceil \lceil m/p_2 \rceil$ always holds [15], the transformed coalesced loop normally results in L smaller than the original loop nest.

In the following sections, partitioning is considered only with respect to the index of the outermost loop.

```
DOALL I=L,U
   (statements.1)
   IF (I.GT.A) THEN
      (statements.2)
   ELSE
      (statements.3)
   ENDIF
ENDDO
```

a) Loop with conditional.

```
DOALL I=L,A
   (statements.1)
   (statements.3)
ENDDO
DOALL I=A+1,U
   (statements.1)
   (statements.2)
ENDDO
```

b) After index set splitting.

```
      N=A-L+1
      M=U-A
C*KSR* PARALLEL REGION(NUMTHREADS=P,PRIVATE=I,K,LK,UK)
      K=IPR_MID()
      LK=L+K*FLOOR(N/P)+MIN(MOD(N,P),K)
      UK=L+(K+1)*FLOOR(N/P)+MIN(MOD(N,P),K+1)
      DO I=LK,UK
         (statements.1)
         (statements.3)
      ENDDO
      LK=A+1+K*FLOOR(M/P)+MAX(0,K-P+MOD(M,P))
      UK=A+1+(K+1)*FLOOR(M/P)+MAX(0,K+1-P+MOD(M,P))
      DO I=LK,UK
         (statements.1)
         (statements.2)
      ENDDO
C*KSR* END PARALLEL REGION
```

c) After loop partitioning.

Figure 1: Partitioning loop nests containing conditionals.

3.2 Loop Nests Containing Conditionals

If the loop body contains conditionals whose execution does *not* depend on the value of the index of the outer loop, then the load imbalance resulting from the partitioning schemes described so far is not affected; each iteration of the outer loop still performs the same amount of work and, consequently, perfect load balance can be achieved. In the special case where an inner loop conditional involves the index of the outer loop and a constant, then, by applying *index set splitting* [20] prior to partitioning, the conditional may be removed [2].

For instance, let $l \leq i \leq u$ bound the index i of the outermost, parallel loop, and $l_x \leq i \leq u_x$ correspond to the logical expression evaluated by a conditional in the inner loop body. Then, the original loop nest can be split into three consecutive loop nests, whose indices take values in the following intervals, respectively:

$$[l, \min(l_x - 1, u)],\ [\max(l, l_x), \min(u, u_x)],\ [\max(u_x + 1, l), u]. \tag{6}$$

The second interval contains the values of i which satisfy both $l \leq i \leq u$ and $l_x \leq i \leq u_x$. In some cases, the upper bound of an interval will be smaller than its lower bound, and the corresponding loop nest will never be executed; in this case, it can be eliminated. Assuming that at least two of the intervals are non-empty, the question is how to partition the resulting loop nests. There are two main approaches which are illustrated in the following example.

Consider the code shown in Figure 1.a;[2] applying (6) and assuming that, for the values of `L`, `U`, `A`, it is known at compile-time that `L`≤`A`<`U`, the code shown in Figure 1.b results. One approach to partitioning this code is to partition each loop using either of the schemes described in Section 3.1, and then assign the same partition of each loop to the same processor; for both loops, if the number of iterations is a multiple of the number of processors, p, then perfect load balance is achieved. In the general case, let n, m be the number of loop executions, and W_1, W_2 be the workload in the body of each loop, respectively; assuming that the same partitioning scheme (either by decreasing or increasing order) is applied to both loops, then the resulting L is given by $L = \lceil \frac{n}{p} \rceil W_1 + \lceil \frac{m}{p} \rceil W_2 - \frac{nW_1 + mW_2}{p}$. Whenever $(n \bmod p) + (m \bmod p) \leq p$, the load imbalance can be reduced by partitioning one of the loops by decreasing order and the other one by increasing order. This approach is followed in the code shown in Figure 1.c.[3]

[2] The `DOALL` construct denotes a parallel loop.

```
DOALL i = l_1, u_1
      (statements.1)
      DO j_2 = l_21 i + l_22, u_21 i + u_22
         (statements.2)
         DO j_3 = l_31 i + l_32 j_2 + l_33, u_31 i + u_32 j_2 + u_33
            ...
            (statements, including m - 4 DO loops)
            ...
            DO  j_m = l_m1 i + l_m2 j_2 + ... + l_m,m-1 j_m-1 + l_mm,
&                 u_m1 i + u_m2 j_2 + ... + u_m,m-1 j_m-1 + u_mm
               (statements.m)
            ENDDO
            ...
            (statements, including m - 4 ENDDO)
            ...
         ENDDO
         (statements.2m-2)
      ENDDO
      (statements.2m-1)
ENDDO
```

Figure 2: A canonical loop nest of depth m.

3.3 Canonical Loop Nests

The partitioning schemes for rectangular loops presented in Section 3.1 result in a small value of load imbalance, when each iteration of the parallel loop performs the same amount of work. A simple counter-example is that of a *triangular* loop nest in which the index of the outer loop, i, takes values from 1 to n, while the index of the inner loop takes values from 1 to i. When this is mapped onto p processors, then, for $p, n \geq 2$, the relative load imbalance has a lower bound of $1/4$. It is apparent that the partitioning schemes described so far are inadequate for minimising load imbalance; nevertheless, using them as a basis, more effective schemes can be devised.

In the remainder of this paper we examine loop nests of depth m having the general form shown in Figure 2. It is assumed that the sets of statements labelled `statements.1`, `statements.2`, ..., `statements.2m-1` do *not* include statements whose execution depends on the value of the index of a surrounding loop; hence, the workload corresponding to each set of statements remains the same for any single execution of the outer loop. This does not exclude the possibility that, inside any of the above-mentioned sets of state-

[3] The KSR directives are used to denote parallelism in the partitioned code. Thus, the code enclosed within the `PARALLEL REGION` and `END PARALLEL REGION` directives is executed by all `P` processors, but using different data for each processor; this is achieved by means of a library function, `IPR_MID()`, which returns an integer between `0` and `P-1` depending on which processor executes the code. The variables `I`, `K`, `LK`, and `UK` are declared as `PRIVATE`, meaning that each processor has its own local copy of the variable.

ments, `DO` ... `ENDDO` loops which perform the same number of iterations regardless of the value of i, may exist. Thus, literally, the depth of a loop nest may be higher than m. However, in the following, only those loops having bounds dependent (directly or indirectly) on the index of the outermost loop are considered; any remaining loops do not affect the strategy and they are omitted.

Definition 1 *Consider the loop nest of depth m, $m \geq 2$, shown in Figure 2, in which the body of the outermost loop contains $m-1$ nested loops; this loop nest is a* **canonical loop nest of depth m**, *if and only if $u_1 > l_1$ and, for all $i, j_2, j_3, \ldots, j_m$, the following inequalities always hold:*

$$l_{21}i + l_{22} \leq u_{21}i + u_{22}$$
$$l_{31}i + l_{32}j_2 + l_{33} \leq u_{31}i + u_{32}j_2 + u_{33}$$
$$\ldots$$
$$l_{m1}i + l_{m2}j_2 + \ldots + l_{mm} \leq u_{m1}i + u_{m2}j_2 + \ldots + u_{mm}$$

where, for each of j_k, $2 \leq k \leq m$, at least one of the differences $(l_{k1} - u_{k1})$, $(l_{k2} - u_{k2}), \ldots, (l_{k,k-1} - u_{k,k-1})$ is non-zero, and the set `(statements.m)` *is not empty.*

Based on Definition 1, we can prove [15]:

Theorem 1 *Consider any canonical loop nest of depth m. Assume that the outer loop is partitioned into $2p^{m-1}$ equal partitions; then, for all k, $0 \leq k \leq 2p^{m-1}-1$, the k-th partition embodies workload W_k given by*

$$W_k = \sum_{i=0}^{m-1} C_i k^i, \quad C_i \text{ constants.}$$

In order to illustrate Theorem 1, consider again the triangular loop nest in which the index, i, of the outer loop takes values from 1 to n, and the index of the inner loop takes values from 1 to i. Assuming that i is partitioned into p equal partitions, the k-th partition, $0 \leq k < p$, embodies workload $W_k = C_1 k + C_0$, for C_0, C_1 constants. This property leads to the following [15]:

Theorem 2 *Consider any loop nest which is partitioned along the index of the outer loop into $2p^{m-1}$, $m \geq 2$, equal partitions. If, for all k, $0 \leq k \leq 2p^{m-1}-1$, the k-th partition embodies workload, W_k, given by*

$$W_k = \sum_{i=0}^{m-1} C_i k^i, \quad \text{where } C_i \text{ constants,}$$

then the loop nest can be partitioned into p partitions of equal workload; the set of partitions along the index of the outermost loop which compose the k'-th, $0 \leq k' < p$, partition of the loop nest is given by

$$S_{k'} = \bigcup_{i=0}^{p^{m-2}-1} \left\{ \{2pi + (k' + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor) \bmod p\} \cup \{2p(i+1) - 1 - (k' + \sum_{j=0}^{m-3} \lfloor i/p^j \rfloor) \bmod p\} \right\}. \quad (7)$$

For $m = 2$, (7) reduces to $S_{k'} = \{k'\} \cup \{2p - k' - 1\}$ [16].

```
      DOALL I=1,N
         DO J=-2,3*I-1
            DO K=J+I,5*I+2
               (statements)
            ENDDO
         ENDDO
      ENDDO
```

a) Unpartitioned loop nest.

```
C  --- assuming that MODULO(N,2*P*P)=0 ---
C*KSR* PARALLEL REGION(NUMTHREADS=P,
C*KSR*&    PRIVATE=I,J,K,LK,UK,K1,K2)
      K1=IPR_MID()
      DO II=0,P-1
         K2=2*P*II+MOD(K1+II,P)
         LK=1+K2*N/(2*P*P)
         UK=1+(K2+1)*N/(2*P*P)-1
         DO I=LK,UK
            DO J=-2,3*I-1
               DO K=J+I,5*I+2
                  (statements)
               ENDDO
            ENDDO
         ENDDO
         K2=2*P*(II+1)-1-MOD(K1+II,P)
         LK=1+K2*N/(2*P*P)
         UK=1+(K2+1)*N/(2*P*P)-1
         DO I=LK,UK
            DO J=-2,3*I-1
               DO K=J+I,5*I+2
                  (statements)
               ENDDO
            ENDDO
         ENDDO
      ENDDO
C*KSR* END PARALLEL REGION
```

b) After loop partitioning.

Figure 3: Example of partitioning a loop nest of depth 3.

Corollary 1 *Consider a canonical loop nest of depth m; if the index of the outer loop can be partitioned into $2p^{m-1}$ equal partitions, then the loop nest can be partitioned into p partitions of equal workload.*

The following example illustrates Theorems 1 and 2:

Example 1 Consider the loop nest shown in Figure 3.a. Assuming that `N` > 1, then the inequalities `-2` $\leq$ `3*I-1` and `J+I` $\leq$ `5*I+2` always hold, while, for each inequality, the coefficients of `I` are non-zero; hence, the requirements of Definition 1 are satisfied and the loop nest is canonical of depth 3. Thus, based on Theorems 1 and 2, and assuming that the number of iterations of the outer loop, `N`, is a multiple of $2p^2$, where p is the number of processors, partitioning the loop nest according to (7) leads to perfect load balance; the partitioned loop nest is shown in Figure 3.b. □

In the general case, where the number of iterations of the outer loop is not a multiple of $2p^{m-1}$, the partitioning technique suggested by Theorem 2 can be applied, provided that the outer loop is partitioned according to one of the partitioning schemes described in Section 3.1. In this case, a small value of load imbalance is expected.

Theorems 1 and 2 also apply to loop nests in which there are more than one inner loop at the same level (i.e., loops which are surrounded only by the same outer loops) whose bounds depend on the index of a surrounding loop; the nec-

```
DOALL I=1,1000
      DO J=1,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,1000
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
```

a) Unpartitioned loop nest.

```
DOALL I=1,750
      DO J=MAX(1,2*I-1000),I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,MIN(1000,1000-I)
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
DOALL I=751,1000
      DO J=MAX(1,2*I-1000),I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
ENDDO
```

b) After initial index set splitting.

```
DOALL I=1,500
      DO J=1,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,1000-I
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
```

```
DOALL I=501,750
      DO J=2*I-1000,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
      DO J=2*I-500,1000-I
         DO K=I+J,1000
            (statements.3)
         ENDDO
      ENDDO
ENDDO
DOALL I=751,1000
      DO J=2*I-1000,I
         DO K=2*I-J,1000
            (statements.1)
         ENDDO
      ENDDO
      (statements.2)
ENDDO
```

c) Transformed code.

Figure 4: Transforming a non-canonical loop nest to canonical loop nests.

essary proviso is that, for any loop in the nest, the lower bound is always less than or equal to the upper bound.

3.4 Non-Canonical Loop Nests

Section 3.3 describes an approach to partitioning canonical loop nests, as introduced in Definition 1. In this section we re-consider loop nests having the general form shown in Figure 2, but without the restrictions associated with Definition 1, apart from the requirement that $l_1 \leq u_1$ (the loop nest is non-empty). Applying index set splitting, the original loop nest can be transformed into multiple adjacent loop nests each of which *either* satisfies the requirements for partitioning inherent in Theorem 2 *or* is rectangular.

Consider the loop nest shown in Figure 2; the first step consists of finding the values of i which satisfy the inequalities $l_1 \leq i \leq u_1$ and $l_{21}i + l_{22} \leq u_{21}i + u_{22}$. If no such values exist, then the loop with index j_2 is never executed. If there are such values, given by $l_1 \leq i \leq u_1$, the loop with index j_2 is always executed; therefore, this loop and the outermost loop together meet the criteria for a canonical loop nest of depth 2, and no index set splitting is required. Conversely, if there is a subset of the values of i which satisfies both inequalities, say $l_1 \leq i \leq u'_1$, where $u'_1 < u_1$, then the outer loop must be split into two consecutive loops, the first of which corresponds to the values given by $l_1 \leq i \leq u'_1$, and the second of which corresponds to $u'_1 + 1 \leq i \leq u_1$; for the former values, the loop with index j_2 and the outer loop together meet the criteria for a canonical loop nest of depth 2, while, for the latter values, the loop with index j_2 is never executed.

If, as a result of the previous step, there are some values of i for which the two outermost loops form a canonical loop nest of depth 2, then the next step consists of finding values of i and j_2 for which the three outermost loops form a canonical loop nest of depth 3. These values must satisfy the system of inequalities:

$$l'_1 \leq i \leq u'_1$$
$$l_{21}i + l_{22} \leq j_2 \leq u_{21}i + u_{22}$$
$$l_{31}i + l_{32}j_2 + l_{33} \leq u_{31}i + u_{32}j_2 + u_{33},$$

where the first inequality corresponds to those values of i that make the two outermost loops a canonical loop nest of depth 2.

The same procedure is repeated for each loop, successively, until there are no remaining loops or else a given system of, say k, $2 \leq k \leq m$, inequalities has no solutions (this would imply that the loop with index j_k is never executed for the values of $i, j_2, \ldots, j_{k-1}$ that make the $k-1$ outermost loops form a canonical loop nest of depth $k-1$). Note that, in the case where the original loop nest contains more than one consecutive loop at some level, the same procedure should be applied for each loop separately.

These ideas are illustrated in the following example:

Example 2 Consider the loop nest of depth 3 shown in Figure 4.a. Since there are two consecutive loop nests in the body of the `I` loop, the procedure described above must be applied separately for each of them.

For the first loop nest, the `J` loop and the outermost loop form a canonical loop nest of depth 2; and the `K` loop joins them to form a canonical loop nest of depth 3 whenever `2*I-J` $\leq$ `1000` $\iff$ `J` $\geq$ `2*I-1000`. Thus, the `J` loop must be split into two consecutive loops depending on appropriate values of `J`; the bounds of the first such loop will be `1` and `MAX(1,2*I-1000)-1`, and of the second such loop `MAX(1,2*I-1000)` and `I`. Since the body of the `J` loop does not contain statements other than the `K` loop, no statements are executed in the case when `1` $\leq$ `J` $\leq$ `MAX(1,2*I-1000)-1`; hence, the corresponding loop can be eliminated.

For the second loop nest, the `J` loop and the outermost loop form a canonical loop nest of depth 2 when `2*I-500` $\leq$ `1000` $\iff$ `I` $\leq$ `750`; the index of the `I` loop is split accordingly. The `K` loop joins in to form a canonical loop nest of depth 3 when `I+J` $\leq$ `1000` $\iff$ `J` $\leq$ `1000-I`.

The code resulting after applying the necessary transformations is shown in Figure 4.b. Evaluating `MAX(1,2*I-1000)`, by replacing it with appropriate conditionals which are then removed using index set splitting (see Section 3.2), results in the code shown in Figure 4.c (note that `MIN(1000,1000-I)` is always equal to `1000-I` since `I`

N	Mapping scheme	Number of processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
256	KAP/MARS	1056768	.428	923648	.566	577024	.620	356749.3	.602	319360	.644
	CYC	8256	.006	12416	.017	14560	.040	15331.3	.061	15760	.082
	BCS	382	.000	13271	.018	7183	.020	6329.3	.026	9828	.053
	CAN-2	262144	.156	229376	.245	143360	.288	82091.3	.258	79360	.310
	CAN-3	0	.000	0	.000	0	.000	50.3	.000	512	.003
1024	KAP/MARS	67239936	.428	58818560	.567	36757504	.621	22978604	.606	20346880	.645
	CYC	131328	.001	197120	.004	230272	.010	241550	.016	247360	.022
	BCS	151255	.002	118940	.003	193075	.009	322800	.021	281770	.025
	CAN-2	16777216	.158	14680064	.247	9175040	.290	6228806	.294	5079040	.312
	CAN-3	0	.000	0	.000	0	.000	48713	.003	0	.000

Table 1: Values of L and L_R for upper triangular matrix multiplication.

takes only positive values).

Finally, partitioning I for each loop nest, as shown in Section 3.2, and grouping the partitions according to (7) for $m = 3$, the partitioned code leads to perfect load balance when using 5 processors, and, in general, a relatively low value of load imbalance [15]. □

4 Evaluation and Experimental Results

A series of experiments has been conducted in order to evaluate the performance obtained by the partitioning strategy described above, compared with other compile-time approaches. Two routes have been adopted for analysing the results when applying different mapping schemes: the first compares the values of load imbalance, L, and relative load imbalance, L_R, computed as shown in Section 2.1; the second compares the resulting performance on a virtual shared memory computer, the KSR1. Our objectives have been not only to evaluate the practical efficacy of the new partitioning schemes, but also to establish whether the theoretical values for L and/or L_R are a sound means for justifying the selection of a particular mapping scheme.

Two benchmark programs are used (see below). The compared approaches are denoted KAP, MARS, CYC, BCS, and CAN: KAP corresponds to the mapping strategy of the KAP auto-parallelising compiler; MARS corresponds to the mapping strategy of the MARS experimental parallelising compiler [3]; CYC corresponds to a cyclic scheme for mapping the iterations onto processors (i.e., processor 0 executes iterations $1, p+1, 2p+1, \ldots$, processor 1 executes iterations $2, p+2, 2p+2, \ldots$, in general, processor i, $0 \leq i \leq p-1$, executes iterations $i+1+kp, k = 0, 1, 2, \ldots, n/p-1$ [11]); BCS corresponds to balanced chunk scheduling [10] (extended to support loop nests of depth 3); and the general term CAN corresponds to the partitioning scheme described by (7). A suffix is added to CAN to distinguish between different values of m and/or transformations applied; these are described below, as appropriate.

4.1 Upper Triangular Matrix Multiplication

The code for the first benchmark, shown below, performs the multiplication of two upper triangular $n \times n$ matrices.

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

Clearly, the loop nest is canonical of depth 3 (see Definition 1), and the partitioning scheme CAN-3, based on (7) for $m = 3$, may lead to perfect load balance. For comparison, the partitioning scheme CAN-2, corresponding to $m = 2$, is also implemented.

The load imbalance, L, in terms of the number of times the assignment statement of the loop body is executed, and the corresponding relative load imbalance, L_R, for two different values of N, 256 and 1024, are shown in Table 1. MARS and KAP exhibit high L and L_R, CAN-2 exhibits relatively smaller values, while the remaining three mapping schemes exhibit significantly smaller values; in all cases, CAN-3 exhibits the smallest values.

The partitioned programs were executed on the KSR1, using the same two values of N; the resulting performance is depicted in Figures 5 and 6, where the *ideal* line assumes linear speed-up. In both graphs, KAP and MARS perform worst of all while CAN-3 performs best; the performance of CAN-3 is comparable with that of CYC and BCS. These results are consistent with the performance that might be anticipated from the values of L and L_R shown in Table 1.

4.2 Banded SYR2K

The second benchmark, banded symmetric rank-$2k$ update (SYR2K), contains non-affine bounds, as shown below:

```
 DOALL I=1,MIN(N,2*BB-1)
    DO J=MAX(1-BB,1-N),MIN(BB-I,N-I)
       DO K=MAX(1,I+J),MIN(N+J,N)
          C(-I-J+K+1,I)=C(-I-J+K+1,I)+A(K,-I-J+BB+1)*B(K,-J+BB)
&                                    +A(K,-J+BB)*B(K,-I-J+BB+1)
       ENDDO
    ENDDO
 ENDDO
```

Clearly, this loop nest is not canonical. However, converting the MIN and MAX functions to IF statements, and removing the latter by index set splitting (see Section 3.2), the code can be transformed into four consecutive canonical loop nests of depth 3, assuming that N > 2*BB-1 [15]; this version is denoted CAN-3t. For comparison, two additional mapping schemes are also implemented; they are based on direct application of the partitioning schemes described by (7), for $m = 2$ (CAN-2) and $m = 3$ (CAN-3), to the original loop nest, regardless of the fact that the latter is not canonical. No version based on balanced chunk scheduling was implemented since loop nests having bounds containing MIN and MAX functions do not conform to its requirements.

The load imbalance, L, in terms of the number of times the assignment statement of the loop body is executed, and the corresponding L_R, for two pairs of values for N and BB, $\{512, 64\}$, and $\{1024, 256\}$, are shown in Table 2. MARS

N, BB	Mapping scheme	Number of processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
512, 64	KAP/MARS	1004896	.350	764592	.450	447832	.490	331685	.516	240300	.507
	CYC	15360	.008	23056	.024	26936	.055	28645	.084	28940	.110
	CAN-2	992	.001	19216	.020	17920	.037	12597	.039	9920	.041
	CAN-3	8192	.004	1024	.001	128	.000	1633	.005	560	.002
	CAN-3t	2080	.001	31248	.032	60360	.114	73227	.191	31668	.120
1024, 256	KAP/MARS	30758272	.367	23767744	.473	13981024	.513	9924928	.529	7514800	.531
	CYC	114688	.002	172096	.006	200928	.015	211168	.023	215600	.031
	CAN-2	1851264	.034	1478464	.053	1146880	.079	692496	.073	537360	.075
	CAN-3	524288	.010	65536	.002	8192	.001	22392	.003	1024	.000
	CAN-3t	128	.000	244800	.009	252960	.019	510110	.054	452880	.064

Table 2: Values of L and L_R for banded SYR2K.

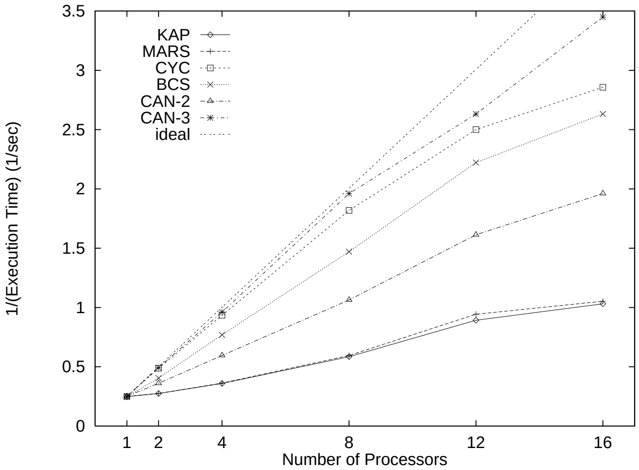

Figure 5: Performance of mapping schemes on the KSR1 for upper triangular matrix multiplication; N = 256.

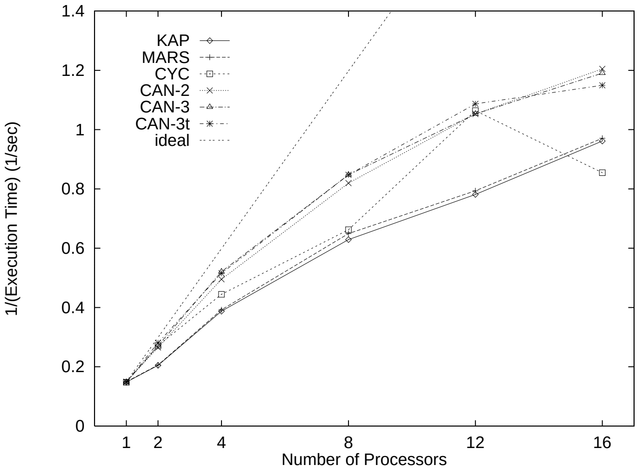

Figure 7: Performance of mapping schemes on the KSR1 for banded SYR2K; N = 512, BB = 64.

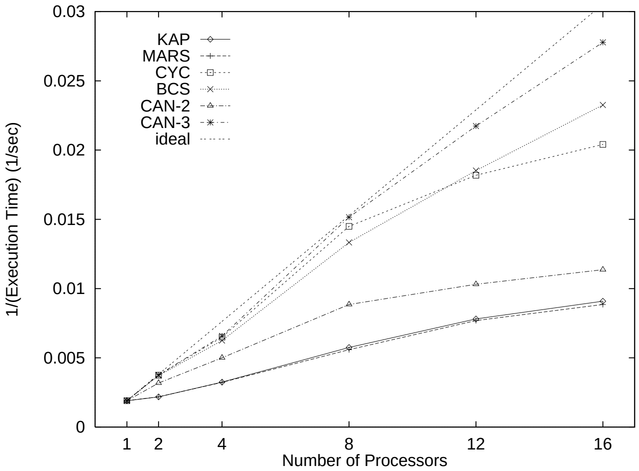

Figure 6: Performance of mapping schemes on the KSR1 for upper triangular matrix multiplication; N = 1024.

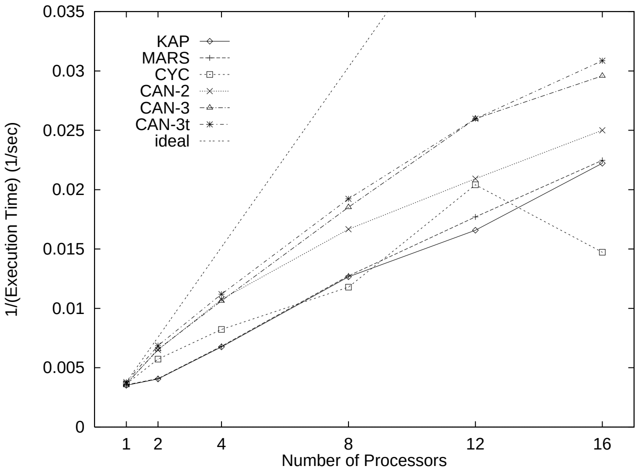

Figure 8: Performance of mapping schemes on the KSR1 for banded SYR2K; N = 1024, BB = 256.

and KAP exhibit high L and L_R, while the remaining four schemes exhibit significantly smaller values; CAN-3 exhibits, on average, the smallest values.

The partitioned programs were executed on the KSR1, using the same two pairs of values for `N` and `BB`; the resulting performance is depicted in Figures 7 and 8. In the first case (Figure 7), KAP and MARS perform worst of all, except when running on 16 processors, where CYC performs worst of all. CAN-3t performs best of all when using fewer than 16 processors; equally good results are achieved by CAN-3 and, to some extent, CAN-2. CYC exhibits odd behaviour; it performs nearly best of all when running on 12 processors, but worst of all when running on 16 processors, and nearly worst when running on 8 processors. This is due to the significant number of cache misses when the number of processors is a power of 2. Similar remarks can be made about the results in Figure 8. CAN-3t performs best of all; CAN-3 exhibits comparable performance, but CAN-2 performs significantly worse. KAP and MARS perform worst of all except when using 8 or 16 processors; in these cases, CYC, which also suffers from a high number of cache misses, performs worst of all.

Comparing the computed values of L and L_R in Table 2 and the actual performance shown in Figures 7 and 8, another interesting observation is that, although CAN-3t nearly always exhibits higher load imbalance than CAN-3, its actual performance is generally better than that of CAN-3 (except when running on more than 12 processors, where the difference in load imbalance between the two partitioning schemes becomes relatively higher); the superior performance of CAN-3t is due to the elimination of `MIN` and `MAX` functions from the loop bounds (apart from those necessary for partitioning the outermost, parallel loop).

5 Conclusion

This paper has presented a partitioning scheme for loop nests in which, upon partitioning into equal partitions along the index of the outermost loop, each partition has a computational load which can be expressed in terms of a polynomial expression; these loop nests, termed *canonical*, are composed of loops for which the upper bound is always greater than or equal to the lower bound. It has also been shown how to apply index set splitting to transform non-canonical loop nests in such a way that the above criterion is satisfied. Although minimising load imbalance has been the primary target of the scheme, it seems that, by partitioning into groups having consecutive iterations (in contrast to the cyclic partitioning scheme [11]), as well as into as near as possible equal-sized partitions along the index of the outermost loop (in contrast to balanced chunk scheduling [9, 10]), our approach has also been effective in reducing other forms of overhead.

References

[1] A. I. Barvinok. Computing the Volume, Counting Integral Points, and Exponential Sums. *Discrete & Computational Geometry*, **10**-2, 1993, pp. 123–141.

[2] A. J. C. Bik, H. A. G. Wijshoff. Iteration Space Partitioning. In H. Liddell, A. Colbrook, B. Hertzberger, P. Sloot (Eds.) *High-Performance Computing and Networking*, LNCS **1067**, Springer-Verlag, 1996, pp. 475–484.

[3] F. Bodin, M. O'Boyle. A Compiler Strategy for Shared Virtual Memories. In B. K. Szymanski, B. Sinharoy (Eds.), *Languages, Compilers and Run-Time Systems for Scalable Computers*, Kluwer Academic Publishers, 1996, pp. 57–69.

[4] J. M. Bull. A hierarchical classification of overheads in parallel programs. In I. Jelly, I. Gorton and P. Croll (eds.), *Software Engineering for Parallel and Distributed Systems*, Chapman & Hall, 1996, pp. 208–219.

[5] P. Clauss. Counting Solutions to Linear and Nonlinear Constraints through Ehrhart polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 1996 International Conference on Supercomputing* (Philadelphia, May 1996), ACM Press, pp. 278–285.

[6] M. E. Crovella, T. J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proceedings of Supercomputing '94* (Washington D. C., Nov. 1994), IEEE Computer Society Press, pp. 600–609.

[7] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, **28**-11, Nov. 1995, pp. 47–56.

[8] S. Flynn Hummel, E. Schonberg, L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, **35**-8, Aug. 1992, pp. 90–101.

[9] M. R. Haghighat, C. D. Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization, and Scheduling of Programs. In U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), *Languages and Compilers for Parallel Computing* (6th International Workshop, Aug. 1993), LNCS **768**, Springer-Verlag, 1994, pp. 567–585.

[10] M. R. Haghighat, C. D. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems*, **18**-4, July 1996, pp. 477–518.

[11] D. J. Lilja. Exploiting the Parallelism Available in Loops. *IEEE Computer*, **27**-2, Feb. 1994, pp. 13–26.

[12] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[13] C. D. Polychronopoulos, D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, **36**-12, Dec. 1987, pp. 1425–1439.

[14] W. Pugh. Counting Solutions to Presburger Formulas: How and Why. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (Orlando, June 1994), *ACM SIGPLAN Notices*, **29**-6, June 1994, pp. 121–134.

[15] R. Sakellariou. *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations.* PhD Thesis, Department of Computer Science, University of Manchester, 1996.

[16] R. Sakellariou. A Compile-Time Partitioning Strategy for Non-Rectangular Loop Nests. In *Proceedings of the 11th International Parallel Processing Symposium* (Geneva, April 1997), IEEE Computer Society Press, 1997, pp. 633–637.

[17] N. Tawbi. Estimation of Nested Loops Execution Time by Integer Arithmetic in Convex Polyhedra. In *Proceedings of the 8th International Parallel Processing Symposium*, IEEE Computer Society Press, 1994, pp. 217–221.

[18] T. H. Tzen, L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, **4**-1, Jan. 1993, pp. 87–98.

[19] K.-Y. Wang. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (Orlando, June 1994), *ACM SIGPLAN Notices*, **29**-6, June 1994, pp. 73–84.

[20] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.