Proceedings of the 11th International Parallel Processing Symposium, IPPS'97 (Geneva, April 1997), IEEE Computer Society Press, pp. 633–637.

A Compile-Time Partitioning Strategy for Non-Rectangular Loop Nests

Rizos Sakellariou
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, U.K.
rizos@cs.man.ac.uk

Abstract

This paper presents a compile-time scheme for partitioning non-rectangular loop nests which consist of inner loops whose bounds depend on the index of the outermost, parallel loop. The minimisation of load imbalance, on the basis of symbolic cost estimates, is considered the main objective; however, options which may increase other sources of overhead are avoided. Experimental results on a virtual shared memory computer are also presented.

1. Introduction

Cost estimates provided by symbolic analysis at compile-time may lead to robust compile-time schemes for mapping certain classes of loop nests; these may be particularly helpful in the context of *parallelising compilers* [8]. Such a scheme, balanced chunk scheduling, is suggested in [3] for mapping triangular perfect loop nests (a loop nest whose iteration points, when depicted on a cartesian system of coordinates, correspond to a triangle is called *triangular*; conversely, a loop nest whose iteration points correspond to a rectangle is called *rectangular*); its main disadvantage lies in not distributing evenly the iterations of the parallel loop, which is often a non-desirable option in practice.

In this paper we develop a novel scheme for mapping at compile-time parallel loops in the body of which there are loops whose execution depends on the index of the parallel loop. A brief definition of load imbalance, whose minimisation is considered the main objective, is given in Section 2. Section 3 analyses the proposed scheme and considers transformations which may be needed. Section 4 presents some experiments, and, finally, Section 5 epitomises the results.

2. Background

With the term loop partitioning we refer to this stage of the mapping phase which deals with the formation of p groups of loop iterations which can be executed in parallel by p processors. When forming these groups the objective is to minimise any overheads, thus increasing performance. In this paper, we consider load imbalance as the dominant overhead; however, to avoid false sharing, we require that each group contains as many consecutive loop iterations as possible, while, to achieve scalability, we require that each group is assigned the same (or almost the same) number of iterations of the loop to be partitioned.

Assuming that the total amount of computation (i.e. the *workload*) in a loop nest is W_{tot}, which is distributed amongst p processors in such a way that each processor i, $0 \leq i < p$, is assigned a workload equal to W_i (clearly, $\sum_{i=0}^{p-1} W_i = W_{tot}$), then we say that this distribution exhibits a *load imbalance*, L, equal to

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

where $W_{max} = \max(W_0, W_1, \ldots, W_{p-1})$. When $L = 0$, that is, for all i, $W_i = W_{tot}/p$, we say that there exists a *perfect load balance*. To assess the impact of load imbalance, we also introduce *relative load imbalance*, L_R, defined as

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

which takes values in $[0, 1 - 1/p]$; values close to zero denote a small impact of load imbalance on performance.

In order to estimate the values of W_i in (1), we consider the number of times that each part of the loop body is executed. Techniques to compute this number (which corresponds to the number of integer points in a polytope) are described in [5, 6, 7]; they are based on the evaluation of nested sums with each sum corresponding to a loop.

Based on the above, the iterations of a single loop with lower bound l and upper bound u can be partitioned across p processors, with processor k, $0 \leq k < p$, executing a loop whose bounds, l_k, u_k, can be computed by $u_k = l_{k+1} - 1$, for $0 \leq k < p - 1$, $u_{p-1} = u$, and, either

$$l_k = l + k\lfloor n/p \rfloor + \min(n \bmod p, k), \text{ or}$$
$$l_k = l + k\lfloor n/p \rfloor + \max(0, k - p + (n \bmod p)).$$

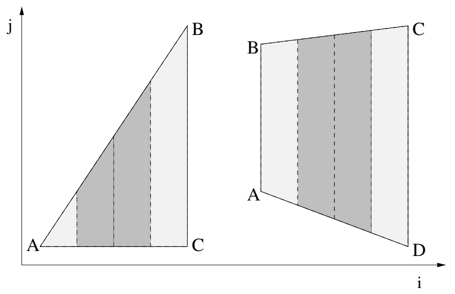

Figure 1. Partitioning non-rectangular iteration spaces into 2 partitions of equal area.

If n is a multiple of p, both the above relations reduce to $l_k = l + kn/p$. In this case, as well as for any rectangular loop nest, perfect load balance is achieved.

3. Methodology

3.1. Generalities

In the case of a non-rectangular loop nest, the partitioning schemes described in the previous section lead to a high value of load imbalance; for instance, for a triangular perfect loop nest the relative load imbalance has a lower bound of $1/4$. However, these schemes can serve as a basis for a more suitable partitioning strategy. In order to illustrate this, consider the triangle and the trapezium shown in Figure 1; the horizontal axis corresponds to the outermost parallel loop. Drawing lines parallel to BC (resp. CD for the trapezium) which cut AC (resp. AD) into 4 equal parts, it is possible to divide the triangle (resp. the trapezium) into 2 partitions of equal area (the light-shaded area and the dark-shaded area); this can be generalised for any number of partitions, as well as for a convex polygon (by dividing it into triangles and trapeziums). This strategy is analysed in the next sections.

3.2. Partitioning canonical loop nests

The class of loop nests examined in this section have the general form of the loop nest shown in Figure 2. The `DOALL` construct denotes a parallel loop which has to be partitioned. It is assumed that the collective sets of statements labelled `statements.1`, `statements.2`, and `statements.3` do not include statements whose execution depends on the value of the index of a surrounding loop (this implies that they may include `DO` ... `ENDDO` loops, which perform the same number of iterations regardless

```
DOALL i = l_1, u_1
      (statements.1)
      DO j = l_21 i + l_22, u_21 i + u_22
         (statements.2)
      ENDDO
      (statements.3)
ENDDO
```

Figure 2. The general form of a triangular/trapezoidal loop nest.

```
DOALL i = l_1, u_1
      (statements.1)
      DO j_1 = l_11 i + l_12, u_11 i + u_12
         (statements.2)
      ENDDO
      (statements.3)
      DO j_2 = l_21 i + l_22, u_21 i + u_22
         (statements.4)
      ENDDO
      ...
      DO j_m = l_m1 i + l_m2, u_m1 i + u_m2
         (statements.2m)
      ENDDO
      (statements.2m+1)
ENDDO
```

Figure 3. The general form of multiple triangular/trapezoidal loop nests.

of the value of `I`). It is also assumed that the second set (`statements.2`) contains at least one statement. Then:

Definition 3.1 *Consider the loop nest shown in Figure 2; this loop nest is* **canonical** *if and only if* $u_1 > l_1$, $u_{21} \neq l_{21}$ *and, for all* i, $u_{21}i + u_{22} \geq l_{21}i + l_{22}$ *always holds.*

Theorem 3.1 *Consider a canonical loop nest; if the index of the outer loop can be partitioned into* $2p$ *equal partitions, then the loop nest can be partitioned into* p *partitions of equal workload, where the* k'*-th partition of the loop nest,* $0 \leq k' < p$*, consists of the* k*-th and* $(2p-k-1)$*-th partition along the index of the outermost loop.*

Theorem 3.1 makes use of a property of canonical loop nests, namely that, upon partitioning into equal partitions along the index of the outermost loop, the k-th partition, $0 \leq k < p$, has a workload equal to $Ak+B$, A, B constants. Based on this property, we can extend the definition of the loop nest used in Theorem 3.1 to cover cases where there are more than one consecutive inner loops:

Corollary 3.1 *Consider the loop nest shown in Figure 3. Assume that the body of the outer loop contains* m, $m > 1$*, consecutive loops whose bounds depend on the index of the outer loop and, for all loops and for all* i *and* k, $1 \leq k \leq m$, $l_{k1}i + l_{22} \leq u_{k1}i + u_{k2}$*; then, if the index of the outer loop*

```
C  U1 is greater than L1
       DOALL I=L1,U1
             DO J=L1,I
                (statements)
             ENDDO
       ENDDO
```

a) Unpartitioned loop nest.

```
C    ---assuming that MODULO(N,2*P)=0---
       N=U1-L1+1
C*KSR* PARALLEL REGION(NUMTHREADS=P,
C*KSR*&PRIVATE=I,J,K,LK,UK)
       K=IPR_MID()
       LK=L1+K*N/(2*P)
       UK=L1+(K+1)*N/(2*P)-1
       DO I=LK,UK
          DO J=L1,I
             (statements)
          ENDDO
       ENDDO
       K=2*P-K-1
       LK=L1+K*N/(2*P)
       UK=L1+(K+1)*N/(2*P)-1
       DO I=LK,UK
          DO J=L1,I
             (statements)
          ENDDO
       ENDDO
C*KSR* END PARALLEL REGION
```

b) After loop partitioning.

Figure 4. Partitioning a triangular loop.

can be partitioned into $2p$ *equal partitions, the loop nest can be partitioned into* p *partitions of equal workload.*

Corollary 3.1 also applies to loop nests where the bound of an inner loop depends on the indices of two outer loops. Unrolling the innermost of the two outer loops, the resulting code has the general form of the loop nest shown in Figure 3.

Example 3.1 Consider the triangular loop nest shown in Figure 4.a. Based on the general form shown in Figure 2, this example is a special case for $l_{21} = 0, l_{22} = l_1, u_{21} = 1$, $u_{22} = 0$; these values satisfy the conditions for a canonical loop nest, as required by Definition 3.1. Thus, assuming that the number of iterations of the outer loop, n, is a multiple of $2p$, where p is the number of processors used, the partitioning scheme described by Theorem 3.1 leads to perfect load balance; the partitioned code is shown in Figure 4.b [1]. □

[1] We use the KSR directives to denote parallelism in the 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; the latter 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`, `J`, `K`, `LK`, and `UK` are declared as private, that is, each processor has its own copy of the variable.

The same partitioning technique may be applied in the general case, where n is not a multiple of $2p$, provided that the outer loop is partitioned according to one of the two partitioning schemes described in Section 2. Then, a relatively small value of load imbalance is expected; for instance, in the case of a canonical perfect loop nest, the relative load imbalance does not exceed $1/(1 + n/2p)$ [6].

3.3. Generalised loop nests

This section re-considers loop nests having the general form shown in Figure 2, but without the restrictions associated with Definition 3.1. First, we prove that, if there are no values of i for which the loop nest is canonical, then the loop nest is rectangular; it is assumed that $l_1 \leq u_1$.

Theorem 3.2 *Consider the loop nest shown in Figure 2; then, either there is a subset of the iteration space of the outer loop for which the loop nest is canonical, or the loop nest is rectangular.*

Theorem 3.2 provides the basis for partitioning any loop nest of the form shown in Figure 2 whether canonical or not. If the original loop nest is neither rectangular nor canonical then, applying *index set splitting* [8, p. 312], it may be split into a canonical and a rectangular loop nest. Considering again the loop nest shown in Figure 2, and assuming that there are some values of i for which the loop nest is canonical, these must satisfy the inequality $u_{21}i+u_{22} \geq l_{21}i+l_{22}$. Thus, if $u_{21} > l_{21}$, the i loop can be split into a loop with bounds l_1, $\lceil r \rceil - 1$ and a loop with bounds $\lceil r \rceil$, u_1, while, if $u_{21} < l_{21}$, the i loop can be split into a loop with bounds $\lfloor r \rfloor + 1$, u_1 and a loop with bounds l_1, $\lfloor r \rfloor$, where $r = (l_{22} - u_{22})/(u_{21} - l_{21})$; in either case, the first loop is rectangular and the second is canonical. Partitioning each of the resulting loop nests by means of an appropriate scheme and using p processors, in the case where $u_{21} > l_{21}$, perfect load balance is achieved if $p|(\lceil r \rceil - l_1)$ and $p|(u_1 - \lceil r \rceil + 1)$, while, in the case where $u_{21} < l_{21}$, perfect load balance is achieved if $p|(u_1 - \lfloor r \rfloor)$ and $p|(\lfloor r \rfloor - l_1 + 1)$.

Theorem 3.2 can also be extended to loop nests such as that shown in Figure 3. In this case, in order to find the appropriate ranges of values for splitting the outer loop index i, we consider the values which make each of the inner loops canonical with respect to the outer loop; thus, the j-th inner loop, $1 \leq j \leq m$, is canonical for those values which satisfy the inequalities $l_1 \leq i \leq u_1$ and $l_{j1}i + l_{j2} \leq u_{j1}i + u_{j2}$. Repeating this procedure for each of the inner loops, the interval $[l_1, u_1]$ is split into a maximum of $m + 1$ subintervals in each of which some of the inner loops are canonical with respect to the outer loop, while others are executed and can be eliminated. This is illustrated in the following example:

Example 3.2 Consider the loop nest shown in Figure 5.a. The inner loops do not meet the requirements set by Corol-

```
DOALL I=1,1000
   DO J=200,2*I-1
      (statements.1)
   ENDDO
   DO J=I+100,1000
      (statements.2)
   ENDDO
ENDDO
```

a) Unpartitioned loop nest.

```
DOALL I=1,100
   DO J=I+100,1000
      (statements.2)
   ENDDO
ENDDO
DOALL I=101,900
   DO J=200,2*I-1
      (statements.1)
   ENDDO
   DO J=I+100,1000
      (statements.2)
   ENDDO
ENDDO
DOALL I=901,1000
   DO J=200,2*I-1
      (statements.1)
   ENDDO
ENDDO
```

b) After index set splitting.

Figure 5. Transforming a non-canonical loop nest into multiple canonical loop nests.

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

Figure 6. Adjoint Convolution.

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
KAP/MARS	8000000	.333	6000000	.429	3500000	.467	2446889	.478	1875000	.484
BCS	1653	.000	1000	.000	2114	.001	3030	.001	3417	.002
CAN	0	.000	0	.000	0	.000	2336	.001	0	.000

Table 1. Expected load imbalance and relative load imbalance for adjoint convolution.

lary 3.1; thus, we consider the values of I which make each of the inner loops canonical with respect to the outer loop, namely $101 \leq \texttt{I} \leq 1000$ for the first of the inner loops and $1 \leq \texttt{I} \leq 900$ for the second. Based on these values, the loop nest can be split as shown in Figure 5.b. Assuming that the number of processors, p, divides 50, the resulting code can be partitioned such that perfect load balance is achieved.

Instead, if the partitioning scheme described by Theorem 3.1 had been applied directly to the loop nest shown in Figure 5.a, and assuming that $p = 10$ processors were used and the amount of work in the body of each of the inner loops was W, a load imbalance equal to $9690W$ would result; applying the partitioning schemes described in Section 2, a load imbalance equal to $48465W$ would result. □

4. Evaluation and Experimental Results

A series of experiments on a virtual shared memory computer, the KSR1, has been conducted. Three benchmark programs which comprise non-rectangular loop nests were used. The different schemes compared are denoted by the shorthands KAP, MARS, BCS, and CAN; KAP corresponds to the mapping strategy of the KAP commercial parallelising compiler, MARS corresponds to the mapping strategy of the MARS experimental parallelising compiler [1], BCS corresponds to balanced chunk scheduling, and CAN to the partitioning scheme described by Theorem 3.1.

Adjoint convolution has been used to evaluate the effectiveness of run-time loop mapping schemes [2, 4], as well as balanced chunk scheduling [3]. The version of the code used is shown in Figure 6. First, we compute the load imbalance, L, in terms of the work associated with the assignment statement of the loop body, and the corresponding relative load imbalance, L_R, for $\texttt{N} = 8000$; the results are shown in Table 1. The performance of the partitioned programs on the KSR1, for the same value of N, is shown in Figure 7; the *ideal* line corresponds to linear speed-up. KAP and MARS perform worst of all while BCS and CAN do best of all; these results are consistent with the anticipated performance from the values of load imbalance shown in Table 1.

The second benchmark examined, a program adding two, upper triangular, $n \times n$ matrices, has been chosen as an example of a loop nest where the size of the data involved (as opposed to adjoint convolution) is considerably larger; the corresponding code is shown in Figure 8. The load imbalance, L, in terms of the work associated with the assignment statement of the loop body, and the corresponding relative load imbalance, L_R, for $\texttt{N} = 1600$, are shown in Table 2. The performance on the KSR1 is depicted in Figure 9; CAN performs best of all, and KAP performs worst of all. While this might have been anticipated from the values of load imbalance shown in Table 2, it appears that the latter do not suffice to provide an adequate justification for the performance of MARS and BCS. This is because the large amount of time spent on memory handling (due to the relatively large size of the arrays involved) renders load imbalance a small fraction of the overall overhead.

Finally, we examine TRED2, an approximately 140-line-long routine from the eigenvalue solver package EISPACK. In order to parallelise the code we consider three loop nests which account for over 99% of the overall execution time for a problem size $\texttt{N} = 1024$; the first two loop nests are triangular, thus the scheme described by Theorem 3.1 can be applied. CAN performs best of all leading to an improvement of up to 20% over MARS and up to 70% over KAP; detailed results are shown in [6].

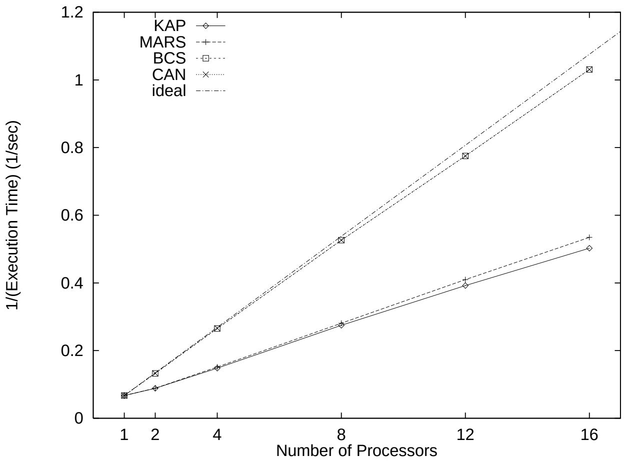

Figure 7. Performance of mapping schemes on the KSR1 for adjoint convolution.

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

Figure 8. Upper Triangular Matrix Addition.

5. Concluding Remarks

This paper presented a strategy for mapping loop nests in which, upon partitioning into p equal partitions along the index of the outermost loop, the k-th partition has a computational load equivalent to $Ak + B$, A, B constants. It has also been shown how to apply index set splitting to transform certain loop nests in a way that the above criterion is satisfied. Our results indicate that the proposed strategy outperforms techniques used by existing parallelising compilers.

Although the strategy has been developed on the basis of minimising load imbalance and evaluated on a virtual shared memory computer, it may also be applicable to a distributed memory environment. Consider, for instance, the code shown in Figure 8. Our strategy implies that if the three arrays are partitioned columnwise into, say, $2p$ equal partitions with processor k, $0 \leq k < p$, being assigned the k-th and the $(2p-k-1)$-th partition, then no interprocessor communication nor load imbalance are expected. The advantage of this scheme is that the same data distribution can be equally efficient for a rectangular loop nest.

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
KAP/MARS	320000	.333	240000	.428	140000	.467	86992	.449	75000	.484
BCS	254	.000	845	.003	499	.003	987	.009	800	.010
CAN	0	.000	0	.000	0	.000	467	.004	0	.000

Table 2. Expected load imbalance and relative load imbalance for triangular matrix addition.

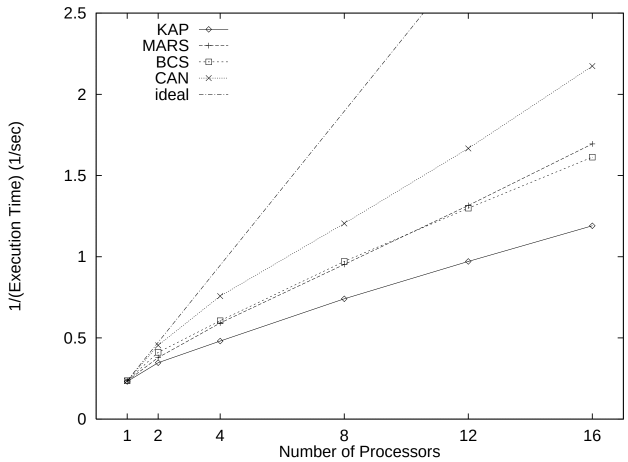

Figure 9. Performance of mapping schemes on the KSR1 for triangular matrix addition.

References

[1] 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*, pp. 57–69. Kluwer Academic Publishers, 1996.

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

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

[4] E. P. Markatos, T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

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

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

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

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