PARTITIONING LOOP NESTS CONTAINING CONDITIONALS FOR AUTOMATIC PARALLELISATION

RIZOS SAKELLARIOU
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, U.K.
E-mail: `rizos@cs.man.ac.uk`

Abstract

This paper presents a novel approach for partitioning a class of loop nests whose iterations perform a varying amount of work due to the presence of `IF` statements. This approach is based on a pre-partitioning loop transformation, index set splitting, which removes the conditional. Preliminary experimental results on a virtual shared memory parallel computer demonstrate the performance improvements achieved using this approach.

1 Introduction

Automatic parallelisation [1] has been promoted as a means of emancipating programmers from the laborious task of converting existing sequential programs into semantically equivalent parallel ones; this conversion is achieved through the development of appropriately designed *parallelising compilers* [13, 14, 15], capable of transforming a sequential program into a parallel one automatically. In the last years, research into this area has been directed mainly towards the parallelisation of scientific programs written in FORTRAN; the approach followed for their parallelisation is centred around the exploitation of *loop parallelism* [8]. Particular emphasis has been placed on how to detect whether a loop can be executed in parallel [3] and how to transform a loop in such a way that parallelism is uncovered [2].

However, the main issue in designing a parallelising compiler is its internal strategy and in particular the order in which the various transformations are applied. Although this is a long standing problem, as a general guideline, three main phases are considered to be the core of the automatic parallelisation process: *detection of parallelism*, *code optimisation* and *mapping* (see [4] for a more detailed description). Decisions taken during these phases can be based on a machine economic model and aim at determining a set of transformations which minimise the program execution time (given as an approximation from an execution time model).

In this paper we deal with the mapping phase, that is, the phase where the detected loop parallelism is mapped onto a parallel architecture. We regard mapping as a two-stage process: *partitioning*, which is the stage where sequential tasks are formed, and *scheduling*, which is the stage where these tasks are assigned to processors; the former stage precedes the latter. It is noted that the concept of mapping varies among the parallel computing community; this reflects the historical development of parallel architectures. In the context of mapping loop parallelism, the name commonly used in the literature is *loop scheduling* [8]. Although this term implicitly assumes the existence of a partitioning stage, it emphasizes the importance of scheduling loop iterations. The reason appears to be that the bulk of research into this area has been directed towards schemes of scheduling the loop iterations at run-time (*dynamic loop scheduling*) in an attempt to balance each processor's workload in cases where each loop iteration does not perform the same amount of work; some of these schemes include self-scheduling [11], guided self-scheduling [10], trapezoid self-scheduling [12], and affinity scheduling [9]. Apart from the run-time overhead incurred by these schemes, their main disadvantage, in the context of automatic parallelisation, is that they deprive the compiler of any information regarding the mapping phase. The availability of this information may be necessary to evaluate cross-processor data dependences (that is, dependences whose source and sink are executed on different processors), or to apply post-partitioning transformations [4]; thus, partitioning at compile-time appears to be preferable.

The remainder of this section will present an overhead-based cost model for automatic parallelisation. Using this model, we consider loop partitioning as an optimisation problem which is solved at compile-time. This methodology, in conjunction with an appropriate loop transformation, is subsequently applied to loop nests containing `IF` statements and two partitioning strategies are devised; these strategies are theoretically and experimentally evaluated on a benchmark program.

1.1 A Model for Automatic Parallelisation

A common approach to model the execution time, t_p, of a parallel program running on p processors is to regard it as $t_p = t_o + t_s/p$, where t_o is the time spent on overheads, and t_s is the time required to execute the sequential version of the program; when parallelising a sequential program the ultimate goal is to minimise t_o. Thus, it is helpful to classify the overheads into a number of classes according to their source. Following an approach proposed in [5], we consider five classes of overheads: *unparallelised code*, *load imbalance*, *communication*, *synchronisation*, and *parallelism start-up*. Assuming that the cost functions F_{UC}, F_{LI}, F_C, F_S and F_{PS} represent an estimate of the time spent as a result of the occurrence of each source of overhead respectively, then an estimate for the overall overhead, t_o, can be computed by $t_o = F_{UC} + F_{LI} + F_C + F_S + F_{PS}$. Such a model has been used in MARS, an experimental parallelising compiler developed jointly at the University of Manchester and IRISA, Rennes, France [4]. By making reasonable assumptions, the compiler can use this model to restrict the search space and optimise programs in an acceptable time.

2 The Problem of Loop Partitioning

Loop partitioning refers to the formation of groups of loop iterations which can be executed in parallel. These groups are formed on the basis of minimising the overall overhead, t_o, as given in the previous section. We consider load imbalance as the dominant overhead, however, we have to avoid options which may increase other sources of overhead. Thus, to avoid false sharing [7], we require that each group contains as many consecutive loop iterations as possible, while, to achieve scalability, we require that each group has almost the same number of iterations.

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

$$L = \max_i \{W_i - \frac{W}{p}\} = W_{max} - \frac{W}{p}, \tag{1}$$

where W_{max} is equal to $\max(W_0, W_1, \ldots, W_{p-1})$. The value of L is bounded between 0 and $W(1-1/p)$. In the former case, that is, when, for all i, $W_i = W/p$, we say that there exists a *perfect load balance*. Thus, reducing the overhead due to load imbalance is equivalent to finding $W_0, W_1, \ldots, W_{p-1}$ such that L in (1) is minimised.

To assess the impact of load imbalance on performance, we introduce *relative load imbalance*, L_R, defined as

$$L_R = \frac{L}{W_{max}} = \frac{W_{max} - \frac{W}{p}}{W_{max}} = 1 - \frac{W}{pW_{max}}. \tag{2}$$

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 of load imbalance on performance.

Based on the above, the problem of partitioning the loop iterations of a single loop with lower bound l and upper bound u amongst p processors [1] can be expressed as that of finding an analytical expression for $l_k, u_k, 0 \le k < p$, such that

$$\sum_{i=l}^{u} 1 = \sum_{i=l_0}^{u_0} 1 + \sum_{i=l_1}^{u_1} 1 + \ldots + \sum_{i=l_{p-1}}^{u_{p-1}} 1 = \sum_{k=0}^{p-1} \sum_{j=l_k}^{u_k} 1, \tag{3}$$

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

$$\left\lfloor \frac{n}{p} \right\rfloor \le \sum_{j=l_k}^{u_k} 1 \le \left\lceil \frac{n}{p} \right\rceil, \tag{4}$$

where $n = u - l + 1 = u_{p-1} - l_0$. The following relations satisfy both (3) and (4):

[1] The loops considered in this paper are of the `DO` ... `ENDDO` type commonly found in FORTRAN programs (equivalently, `DOALL` ... `ENDDO` if they can be executed in parallel).

• **Partitioning by decreasing order:**

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

In this case, the partitions are arranged in decreasing order of their number of iterations; the first $(n \bmod p)$ loops have $\lceil n/p \rceil$ iterations, whilst the rest have $\lfloor n/p \rfloor$. The loop with bounds (l_k, u_k), i.e. the k^{th} partition, $0 \leq k < p$, has $\lceil (n-k)/p \rceil$ iterations.

• **Partitioning by increasing order:**

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

In this case, the partitions are arranged in increasing order of their number of iterations; the last $(n \bmod p)$ loops have $\lceil n/p \rceil$ iterations, whilst the rest have $\lfloor n/p \rfloor$. In general, the loop with bounds (l_k, u_k), $0 \leq k < p$, has $\lfloor (n+k)/p \rfloor$ iterations.

It can be observed that, if n is a multiple of p, both relations reduce to

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

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. Otherwise, if n is not a multiple of p, a relative load imbalance equal to $(p - n \bmod p)/(n + p - n \bmod p)$ is expected, a quantity which approaches zero if $n \gg p$.

3 Dealing with Conditionals

The partitioning schemes described in the previous section result in a small load imbalance provided that each loop iteration performs the same amount of work. However, if this is not the case, they may exhibit significant variations in the load distribution and, consequently, a high load imbalance; in the past, dynamic scheduling seemed to be the only alternative [6].

In this section we consider loop nests where the amount of work in the loop body may vary from iteration to iteration due to the presence of `IF` statements. Clearly, conditionals whose execution does not depend on the index of a loop to be partitioned will not result in a variation of the work assigned to each partition. Thus, we concentrate on conditionals whose execution depends on the index of a loop to be partitioned; these conditionals can be eliminated by applying a pre-partitioning loop transformation, *index set splitting*, which transforms a single loop nest into multiple adjacent loop nests with each loop nest performing a subset of the original iterations.

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

a) Loop with conditional.

```
DOALL I=L,MIN(LX-1,U)
   (statements.1)
   (statements.3)
ENDDO
DOALL I=MAX(L,LX),U
   (statements.1)
   (statements.2)
ENDDO
```

b) After index set splitting.

Figure 1: Removing conditionals.

Assuming that the order of iterations does not change, index set splitting always preserves the loop semantics. Index set splitting has been found useful in breaking data dependences [13, p. 65], but has not been considered to balance the workload.

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

$$[l, \min(l_x - 1, u)], \;\; [\max(l, l_x), \min(u, u_x)], \;\; [\max(u_x + 1, l), u]. \tag{5}$$

The second interval contains the values which satisfy both the inequalities given at the beginning of this paragraph. Note that, in some cases, the upper bound of an interval can be smaller than its lower bound; then, the corresponding loop nest will never be executed and, therefore, it can be eliminated. Additional constraints, present in the same conditional and connected with the previous one by $\wedge$ (AND), can be evaluated by applying the same process to the second interval only (since the first and the third intervals do not satisfy the second inequality and, therefore, the conditional cannot be true if all constraints are connected by $\wedge$). If, initially, the conditional contains constraints connected by $\vee$ (OR), we can transform these to constraints connected by $\wedge$ applying DeMorgan's Law ($A \vee B = \neg(\neg A \wedge \neg B)$); the $\neg$ may be removed. For instance, assuming that A represents the constraint $i \geq l$, then $\neg A$ is equivalent to $i < l$; similarly, the conditional `IF ¬A THEN` ... `ELSE` ... `ENDIF` can be rewritten as `IF A THEN` ... `ELSE` ... `ENDIF` by swapping round the statements after the `THEN` and the `ELSE`. Alternatively, conditionals containing constraints connected by $\vee$, say $A \vee B$, can be split to three successive conditionals each one having constraints connected by $\wedge$, namely: $\neg A \wedge B$, $A \wedge \neg B$, and $A \wedge B$. Clearly, during one program run, only one of these expressions can be true.

To illustrate the above, consider the code shown in figure 1.a; splitting the iterations of the outer loop according to (5) into those rendering the condition of the

IF statement true and those rendering it false, the conditional can be removed and the code shown in figure 1.b results (note that the loop corresponding to the third interval of (5) is never executed and, therefore, it can be eliminated). Each of the loops of the transformed code may be partitioned using anyone of the two partitioning schemes described in the previous section; thus, using p processors, processor k, $0 \leq k < p$, will execute the k^{th} partition of each loop. For both loops, if the number of iterations is a multiple of p, perfect load balance may be achieved. In the general case, let n, m be the number of loop iterations and W_1, W_2 the amount of work present in the body of each loop, respectively (in our example, $n =$ `MIN(LX-1,U)-L+1`, and $m =$ `U-MAX(L,LX)+1`). Assuming that the same partitioning scheme (by either decreasing or increasing order) has been applied to both loops, then the load imbalance, L, is equal to:

$$L = \left\lceil \frac{n}{p} \right\rceil W_1 + \left\lceil \frac{m}{p} \right\rceil W_2 - \frac{nW_1 + mW_2}{p}.$$

If $(n \bmod p) + (m \bmod p) \leq p$, the load imbalance, L, can be reduced to

$$L = \max\left(\left\lceil \frac{n}{p} \right\rceil W_1 + \left\lfloor \frac{m}{p} \right\rfloor W_2, \left\lfloor \frac{n}{p} \right\rfloor W_1 + \left\lceil \frac{m}{p} \right\rceil W_2 \right) - \frac{nW_1 + mW_2}{p},$$

by partitioning one loop by decreasing order and the other by increasing order.

An alternative approach to partitioning the code shown in figure 1.b is to make use of the information that the two loops can be executed concurrently and assign to each loop a number of processors proportional to the corresponding work. Thus, the number of processors, p_1, for the first loop, may be either $\lfloor p\frac{nW_1}{nW_1+mW_2} \rfloor$ or $\lceil p\frac{nW_1}{nW_1+mW_2} \rceil$, while the number of processors for the second loop is $p_2 = p - p_1$; the choice of p_1, p_2 should minimise the load imbalance, which, if $p_1, p_2 \neq 0$, is given by

$$L = \max\left(\left\lceil \frac{n}{p_1} \right\rceil W_1, \left\lceil \frac{m}{p_2} \right\rceil W_2 \right) - \frac{nW_1 + mW_2}{p}.$$

The drawback of this approach is that finding p_1, p_2 requires estimates for W_1, W_2 which may not always be easily derivable.

4 Evaluation and Experimental Results

In order to evaluate the partitioning methods described in the previous section we conducted experiments on the KSR1, a 32-processor virtual shared memory parallel computer, using the code shown in figure 2 which is based on loop nest $L1$ used in [10]. In our experiments, `N1` $= 100$, `N2` $= 50$, `N3` $= 4$, `L1` $= 30$, and `L2` $= 30$; the subroutine `X()` performs a number of multiplications equal to the value of the parameter passed to it. Two sets of experiments were conducted; in the first set

```
DOALL I1=1,N1
   DOALL I2=1,N2
      DOALL I3=1,N3
         CALL X(W1)
         IF ((I1.GT.L1).AND.(I2.GT.L2)) CALL X(W2)
      ENDDO
   ENDDO
ENDDO
```

Figure 2: Loop nest used for the experiments.

W1, W2	Partitioning scheme	Number of Processors									
		2		4		8		12		16	
		L	L_R	L	L_R	L	L_R	L	L_R	L	L_R
1000, 1000	IS-1	0	.000	24	.036	12	.036	14.67	.064	20	.111
	IS-2	300	.190	32	.048	16	.048	10.67	.048	8	.048
	KAP-1	120	.086	60	.086	44	.121	14.67	.064	8	.048
1000, 10000	IS-1	0	.000	60	.031	30	.031	26.67	.040	65	.120
	IS-2	300	.073	500	.208	50	.050	66.67	.095	25	.050
	KAP-1	1200	.240	600	.240	350	.269	166.67	.208	125	.208

Table 1: Expected load imbalance and relative load imbalance for the experiments.

W1 = 1000 and W2 = 1000, while, in the second set, W1 = 1000 and W2 = 10000. In each set of experiments, the performance of four partitioning schemes was compared. The first two schemes (IS-1 and IS-2) are based on the methods described in the previous section, which were applied after splitting accordingly the iterations of the I1 and I2 loops. The remaining two schemes are based on the direct parallelisation of the loop nest shown in figure 2 as performed by the commercial parallelising compiler KAP; in the first case the iterations of the loop nest were mapped by applying partitioning by decreasing order (KAP-1), while, in the second case, a dynamic mapping scheme based on self-scheduling was applied (KAP-2).

We first computed the load imbalance in terms of work associated with 10000 multiplications of the subroutine X(). The resulting load imbalance, L, and the corresponding relative load imbalance, L_R, applying IS-1 (partitioning the first loop by decreasing order and the second loop by increasing order), IS-2, and KAP-1, are shown in table 1. The performance of all four partitioning schemes on the KSR-1 is depicted in figure 3; the *Ideal* line assumes linear speed-up from the best sequential version. It can be seen that, in both experiments, IS-1 performs best of all except when $p = 16$ in the second experiment; these results appear to be consistent with the anticipated performance from the values of load imbalance shown in table 1. Finally, KAP-2 performs well for a small number of processors, however it suffers from a high synchronisation overhead and shows a low speed-up increase.

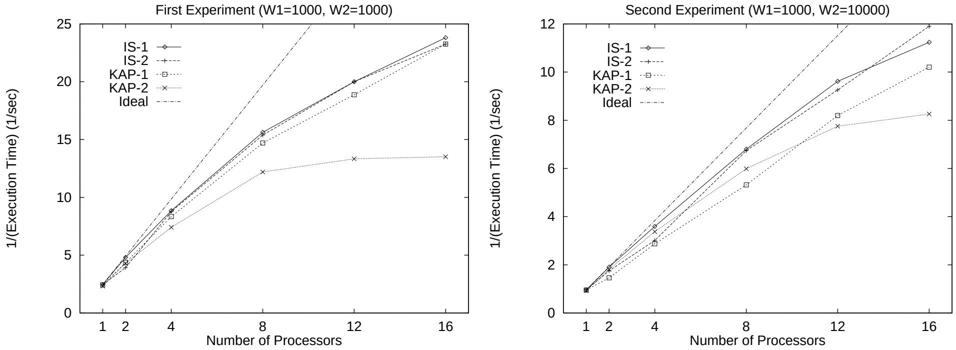

Figure 3: Performance of partitioning schemes on the KSR1.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau, D. Padua, "Automatic Program Parallelization", *Proceedings of the IEEE*, **81**-2, Feb. 1993, pp. 211–243.

[2] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Kluwer Academic Publishers, 1993.

[3] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, P. Tu, "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing", *IEEE Parallel & Distributed Technology*, **2**-3, Fall 1994, pp. 37–47.

[4] 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.

[5] M. E. Crovella, T. J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles Analysis", in *Proceedings of Supercomputing '94*, IEEE Computer Society Press, pp. 600–609.

[6] Z. Fang, P. Tang, P.-C. Yew, C.-Q. Zhu, "Dynamic Processor Self-Scheduling for General Parallel Nested Loops", *IEEE Transactions on Computers*, **39**-7, Jul. 1990, pp. 919–929.

[7] W. Li, "Compiler Optimizations for Cache Locality and Coherence", Technical Report 504, Department of Computer Science, University of Rochester, Apr. 1994.

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

[9] 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, Apr. 1994, pp. 379–400.

[10] 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.

[11] P. Tang, P.-C. Yew, "Processor Self-Scheduling for Multiple Nested Parallel Loops", in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 528–535.

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

[13] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Research Monographs in Parallel and Distributed Computing Series, MIT Press, 1989.

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

[15] H. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press & Addison-Wesley, 1990.