Estimating the Parallel Start-Up Overhead for Parallelizing Compilers

Rizos Sakellariou

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

Abstract. A technique for estimating the cost of executing a loop nest in parallel (*parallel start-up overhead*) is described in this paper. This technique is of utmost importance for parallelizing compilers which take decisions on the basis of predicting performance through the quantification of overheads. Such a model is analyzed and the necessary conditions for computing an estimate for the parallel start-up overhead are presented. Based on this estimate, it is shown how to transform parallelizable loop nests in such a way that the start-up cost for parallel execution does not outweigh the gains. Experimental results demonstrate that this transformation results in performance improvements. Finally, it is also shown that such an estimate is essential to predict the performance of codes whose parallelization is based on the multiple execution of a small parallelized program fragment.

1 Introduction

Parallelizing compilers [28, 30] have been promoted as a promising means for emancipating programmers from the laborious task of the parallelization of codes written in sequential languages. Ideally, such compilers must be capable of converting a sequential program into a semantically equivalent parallel program automatically (or with minimal user intervention); furthermore, the performance of the transformed parallel program must be comparable to that achieved by expert parallel programmers.

However, parallelizing compilers usually fail to accomplish the latter expectation. The main line of thought attributes this 'deficiency' to their inability to detect fully the inherent parallelism of programs [5]. Although more robust techniques for detecting parallelism are undoubtedly needed, an issue of equal, if not greater concern, is the compiler's *internal strategy.* A strategy of parallelizing anything parallelizable does not necessarily yield the best performance results; instead, it may even slow down significantly the execution time of the parallel program. The reason for this phenomenon is that parallelism may often trade off with other overheads; thus, it may be preferable to execute potentially parallel parts of the code sequentially, in order to reduce, for instance, the number of cache misses or avoid interprocessor communication.

Existing commercial parallelizing compilers typically attempt to parallelize all loops whose parallel execution does not alter program semantics, and are

```
 N=500                              DOALL K1=-990,480
 DO J1=10,N                            DO K2=MAX(10,5-FLOOR(K1/2)),
    DO J2=10,N                      &          MIN(500,250-CEILING(K1/2))
       A(J1,J2)=A(J1-2,J2-4)                   A(K2,K1+2*K2)=A(K2-2,K1+2*K2-4)
&                 +A(J1-1,J2-2)     &                         +A(K2-1,K1+2*K2-2)
    ENDDO                              ENDDO
 ENDDO                              ENDDO
```

Fig. 1. Semantically equivalent codes which cause different overheads.

also capable of applying a number of loop transformations; more sophisticated program restructuring transformations [2, 3] are applied by experimental parallelizing compilers [4, 22]. In any case, the disadvantage of the *ad hoc* program restructuring is that different transformations may significantly affect program performance.

To illustrate this, consider the example shown in Figure 1. The two codes are semantically equivalent; the transformed loop nest, shown on the right-hand side, is the result of a unimodular transformation applied to the original loop nest, shown on the left-hand side [24]. Although the transformed loop nest can run in parallel (this is denoted by the `DOALL` construct in the code), a non-unit stride access pattern for array `A` is established; instead, simply applying *loop interchange* [28, Section 9.5] to the original loop nest, this amended (albeit still sequential) version runs approximately 5 times faster than the parallel version on a virtual shared memory multiprocessor, the KSR1 (see [23, 29] for details of its architecture).

Thus, in order to evaluate the trade-offs among different transformations, a parallelizing compiler needs some capability of performance estimation. In this paper, we consider such a model based on quantitative estimates of different sources of overheads; this is described in the next section. Attention is subsequently focused on the estimation of the overhead due to the parallelization of a code. The latter may be important to avoid performance losses, for instance, from the parallelization of loop nests having a relatively small number of iterations. Although this problem has been pointed out in an effectiveness study of parallelizing compilers [5], as well as observed from a scientific user's experience [17, 20], to the best of our knowledge, there has been no work in the literature attempting to estimate it in a systematic way; this paper provides a contribution to this. It also shows how to transform a code using this estimate, while the performance benefits from incorporating this estimation capability into a parallelizing compiler are presented in Section 4.

2 A Model for Compile-Time Performance Prediction

Predicting the performance of a parallel program has been an issue well addressed in the literature [13, 21]. However, accurate performance prediction often requires either a particular machine model [7] or a specific application domain [27]. Both

these restrictions are not desirable in the context of a parallelizing compiler where portability and the ability of performing equally well on a wide range of programs and architectures are expected. An additional problem, posed by the nature of parallelizing compilers *per se*, is that, during the process of evaluating different sequences of transformations, the performance prediction model may have to be called repeatedly. Consequently, its execution time cannot be prohibitively high and the model must be simple, albeit still accurate.

A model based on the classification of *overheads* has been incorporated into the MARS experimental parallelizing compiler, developed jointly at the University of Manchester and IRISA, Rennes [6]. The underlying idea is rather simple; assuming that t_o is the time spent on overheads and t_s is the time required to execute the sequential version of a program, then the running time, t_p, of the parallelized version of the program on p processors is given by

$$t_p = t_o + t_s/p. \quad (1)$$

The crucial point for the efficiency of the model is the estimation of t_o. Historically, this approach goes back to Gene Amdahl, who associated t_o with the inherently sequential parts of the program [1]; more generalized models, also based on the serial and the parallel fractions of a program, are considered in [10, 12]. However, it is essential to consider distinctly all possible sources of overhead, as, for instance, attempted in [8, 9, 11]. Thus, the following classes are identified:

- *Unparallelized code*: this refers to the overhead caused by sections of the program which are (or have to be) executed sequentially.
- *Load imbalance*: this refers to the overhead caused by poor distribution of the parallelized computational work among processors.
- *Communication*: this refers to the overhead which occurs when a processor is waiting for data to be moved from memory; it is subdivided into two further classes, depending on whether data are moved between two different processors' memories, or between various levels of the memory hierarchy within a single processor (e.g. between cache and main memory).
- *Synchronization*: this refers to the overhead caused when a processor is waiting to acquire a lock or at a barrier.
- *Parallel start-up*: this refers to the overhead caused because of any additional computation required for the exploitation of parallelism.

The total time spent on overheads, t_o, is given by the sum of the time spent on each of the above classes, i.e.,

$$t_o = t_{UC} + t_{LI} + t_C + t_S + t_{PS}. \quad (2)$$

To estimate the time spent as a result of the occurrence of each particular source of overhead, we model time as a function of some machine-dependent and some machine-independent parameters; the latter represent a quantitative metric for each individual source of overhead. For instance, in the case of load imbalance, this metric is determined by the amount of computational work (say in machine

instructions) assigned to the most 'overloaded' processor [24, 25]; in the case of synchronization, it may be determined by the number of synchronization points required [26], while, in the case of communication, by the number of cache misses [14] (but not only). In the latter case, the cost, in CPU cycles, of moving data from the main memory to the cache on a particular parallel platform multiplied by the number of cache misses may provide an estimate for t_C (if no other communication costs occur).

The ultimate goal of any parallelization techniques is to minimize t_o in (2). Given that the machine-dependent parameters have a fixed cost, this implies that the compiler aims to transform the program in such a way that the particular quantitative metrics are minimized. However, trade-off situations may arise; reducing one source of overhead may increase another and *vice versa*. Dealing with all the overheads at once may be intractable; thus, it is usually attempted to reduce one source of overhead at a time, avoiding actions for which there is evidence that another source of overhead may increase.

3 Dealing with the Parallel Start-Up Overhead

From the five sources of overhead identified in the previous section, the parallel start-up overhead is a notable exception in that it is independent of the particular program characteristics; it largely depends on the parallelization mechanism applied (from those provided by the particular parallel machine in use) and, possibly, the number of processors employed. These properties, alongside the assessment that this overhead is unavoidable but relatively low, have led researchers to underestimate its importance [6]. Nonetheless, having an estimate for it can be helpful in deciding which loops should be executed in parallel, a need remarked in [5, 17, 20], as well as to evaluate the performance of codes whose parallelized loops are enclosed in the body of large sequential loops.

In order to estimate the parallel start-up overhead, we use Equations (1) and (2). Assume that, in a parallelized code fragment, the value of all the remaining overheads is *zero* and the time, t_s, for executing the sequential version of the code is known; then, for a given number of processors, p, the time spent due to the parallel start-up overhead, t_{PS}, is given by

$$t_{PS} = t_p - t_s/p. \tag{3}$$

Based on the above, any program fragment used for computing t_{PS} must take into account the following:

- No unparallelized code should exist; a program fragment consisting of a single loop without data dependences provides a simple solution to this.
- Perfect load balance (i.e. zero load imbalance) should be possible for any number of processors. In the case of a single loop, the number of iterations must be a multiple of the number of processors, and the loop body should not contain statements whose execution depends on the value of the loop index.

- No communication should occur. Thus, the loop body should not contain array elements.
- Synchronization must be avoided.
- A 'useful' computation (that is, one which yields some output results) must be performed to avoid dead-code elimination.

Parallel construct	Number of processors									
	2		3		4		5		6	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
PAR.REGION	17829	1132	18912	1327	19583	965	21796	1041	23060	884
TILE	22456	1722	24081	1683	24534	1984	26231	1833	27043	1323
	7		8		9		10		11	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
PAR.REGION	23766	1023	24963	987	25680	620	26424	893	26693	692
TILE	27896	1171	29368	1302	30551	983	31891	1124	33145	870
	12		13		14		15		16	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
PAR.REGION	27101	731	27425	758	28857	831	30334	543	31943	723
TILE	34776	954	36092	832	37820	820	38842	773	40246	821

Table 1. Parallel start-up overhead, in CPU cycles, on the KSR1.

Using a model code respecting the above constraints, we compute the parallel start-up overhead for the two types of constructs, *parallel regions* and *tile families* [20, 19], employed for loop parallelization on the KSR1. The mean, μ, as well as the standard deviation, σ, of the values of t_{PS}, in CPU cycles (note that each KSR1 processor has a 20 MHz clock), are shown in Table 1; these values are computed based on (3), after running the code 10000 times (for each given number of processors) and measuring t_s and t_p. For both parallel regions and tile families, the parallel start-up overhead keeps increasing as the number of processors increases. Moreover, the cost for tiling is higher than that of a parallel region; this difference is reasonable considering the cost for barrier synchronization involved in the low-level implementation of the two constructs [16].

4 Application Examples

We evaluate the benefits from using an estimate for the parallel start-up overhead on two codes:

- TRED2, an approximately 100-line-long FORTRAN program taken from the eigenvalue solver package EISPACK, which reduces a symmetric matrix to symmetric tridiagonal form, and,

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

a) First loop nest.

```
 L=N+1-II
 DOALL J=1,L
    TEMP1=D(J)
    TEMP2=E(J)
    DO K=J,L
       Z(K,J)=Z(K,J)
&         -TEMP1*E(K)
&         -TEMP2*D(K)
    ENDDO
 ENDDO
```

b) Second loop nest.

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

c) Third loop nest.

Fig. 2. Parallelized loops in TRED2.

– a program performing numerical weather prediction using a barotropic model.

In the first case, it is shown how an estimate for the parallel start-up overhead can be used to transform the code in such a way that potentially parallel fragments of a program are parallelized *only* if some performance gains are expected. In the second case, it is shown how such an estimate may be essential for good compile-time performance prediction.

4.1 TRED2

The version of TRED2 used in our experiments is analyzed in [19]. Three loop nests, which account for over 93% of the execution time on a single processor for a problem size N=128, are parallelized; these are shown in Figure 2. All three loop nests are contained in the body of an outer sequential DO loop with lower bound 2, upper bound N, and loop index II (for the first two), and I (for the third).

The three loop nests are parallelized by KAP (a commercial parallelizing compiler available on the KSR1) and MARS regardless of the value of L. However, the value of L varies from 1 to N-1; for small values of L the parallel start-up cost may outweigh the gains and, in this case, it may be more efficient to execute the loops sequentially. In order to find for which values of L the latter approach must be followed, we consider again Equation (3); then, for a given number of processors, p, parallel execution is preferred whenever

$$t_s > t_{PS} + t_s/p \Longleftrightarrow t_s(1 - 1/p) - t_{PS} > 0. \qquad (4)$$

Certainly, the three loop nests may incur other overheads (e.g. communication), which the above formulation of the problem does not consider. Here, our aim is to find *only* a lower bound for the values of L which may result in performance gains.

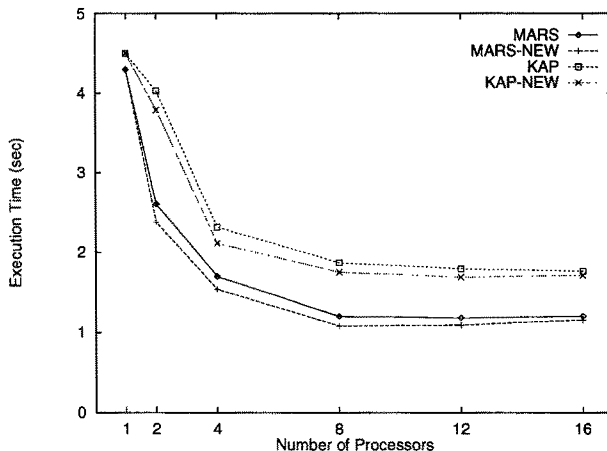

Fig. 3. Execution time of TRED2 on the KSR1.

We modelled t_s in terms of L by considering the amount of work performed by the body of each loop. Thus, for the first loop nest, let W_J represent the amount of work performed by the three assignment statements which are executed only by the J loop, W_{K_1} represent the amount of work performed by the body of the first K loop, and W_{K_2} represent the amount of work performed by the body of the second K loop; similarly, we consider W_J and W_K for the second loop nest, and W_J, W_{K_1} and W_{K_2} for the third loop nest. An estimate for each of these values is based on the number of the corresponding CPU cycles, namely: $W_J = 97$, $W_{K_1} = 51$, and $W_{K_2} = 51$, for the first loop nest; $W_J = 50$, and $W_K = 53$, for the second loop nest; and $W_J = 17$, $W_{K_1} = 46$, and $W_{K_2} = 39$, for the third loop nest. Then, the work performed by each loop nest is given by

$$\begin{aligned} 97\mathrm{L} + 51\mathrm{L}^2, &\quad \text{for the first loop nest,} \\ 50\mathrm{L} + 53\mathrm{L}(\mathrm{L}+1)/2, &\quad \text{for the second loop nest,} \\ 17\mathrm{L} + 85\mathrm{L}^2, &\quad \text{for the third loop nest.} \end{aligned}$$

Considering the above expressions as an approximation of t_s, substituting into (4) and using the results of Table 1, a lower bound for L, for each loop nest and parallelization strategy, can easily be computed. Based on this value, TRED2 was modified by including each parallelizable loop nest in the body of an IF statement which determines whether its execution is performed in parallel or sequentially.

The modified versions, as well as the original versions for both KAP and MARS, were run on the KSR1 for a problem size N=128; their execution times are shown in Figure 3. Both the modified versions, KAP-NEW and MARS-NEW, lead to a clear performance improvement over the original versions.

```
      DOALL J=2,M1
         DOALL I=2,L1
            IF (((I+J)/2)*2.EQ.(I+J)) THEN
               IF (I-L1+1) 12,11,11
11                R=(X(2,J)+X(L1,J)-2.*X(I,J))/DX(J)**2
   &                +(X(I,J+1)+X(I,J-1)-2.*X(I,J))/DY**2
                  R=(R-Y(I,J))*DY*DX(J)
                  GOTO 13
12                R=(X(I+1,J)+X(I-1,J)-2.*X(I,J))/DX(J)**2
   &                +(X(I,J+1)+X(I,J-1)-2.*X(I,J))/DY**2
                  R=(R-Y(I,J))*DY*DX(J)
13             IF (LSC-NSC) 14,14,15
14                X(I,J)=X(I,J)+ALFA*R
15             IF (ABS(R).LE.EPS) THEN
                  NREL=NREL+1
               ENDIF
            ENDIF
         ENDDO
      ENDDO
```

Fig. 4. The parallelized loop of the barotropic weather prediction program.

4.2 A Barotropic Model for Numerical Weather Prediction.

The version of the program we use for numerical weather prediction based on a barotropic model is analyzed in [18]. In order to parallelize the program, we concentrate on the parallelization of a double loop nest which accounts for over 85% of the program execution time on a single processor for a problem size `L1=50`, `M1=20`. This loop nest implements a *Successive Over-Relaxation* (SOR) type of operation based on a five-point stencil [15]. Thus, the original sequential loop nest can be split into two consecutive parallel loop nests. The first of these loop nests is shown in Figure 4; the sole difference with the second loop nest is the logical condition of the third statement (which is changed to `(((I+J)/2)*2.NE.(I+J))`). For the given problem size used in our experiments, both loop nests are executed a total of 2741 times in order to achieve a desired precision.

Contrary to the parallel loops of TRED2, the parallel loops of this example are always executed for the same number of iterations; the average execution time of each parallel loop is 0.00416 sec. Thus, applying (4), it can be easily seen that the performance gains from parallel execution are expected to outweigh the losses due to the parallel start-up overhead. However, because of the multiple execution of the parallel loop nests, it is anticipated that the parallel start-up overhead will constitute a large fraction of the overall overhead.

Using the performance prediction model presented in Section 2, we estimate the execution time of the parallel program based on Equations (1) and (2). We consider $t_{LI} = t_C = t_S = 0$, $t_{UC} = 3.92$, and $t_s = 22.81$ (after profiling the sequential code); from Table 1 we also compute a value for t_{PS}. This estimate is compared with the execution time of the parallel program in Figure 5 (the

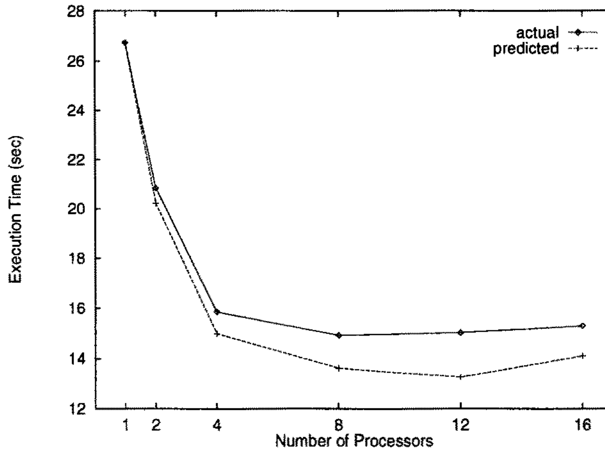

Fig. 5. Actual and predicted execution time of the barotropic weather prediction model.

former noted as `predicted`, the latter noted as `actual`). It can be seen that, although only two sources of overhead are considered, the predicted performance of the parallel program is close to the actual performance; this is because the particular overheads dominate the parallel execution.

5 Conclusion

This paper described an overhead based model for predicting performance for parallelizing compilers, and concentrated on estimating the overhead caused by the initialization of parallelism. Based on such an estimate, a transformation has been suggested; incorporating this into existing parallelizing compilers may result in performance gains. Although, in absolute terms, this overhead is rather small, it becomes increasingly important in codes where parallelizable loop nests with a relatively small amount of work are contained in the body of sequential loops which are executed a large number of times. Then, as demonstrated by our experiments, the parallel start-up overhead may be necessary to evaluate the program's performance behaviour.

In future work, we aim to consider the parallel start-up overhead in conjunction with other sources of overhead. Our main objective is to establish overhead analysis as a means of reasoning for parallelizing compilers in a similar way to what can be used when expert programmers tune programs for parallel execution [23].

Acknowledgements: The author would like to thank the members of the Centre for Novel Computing at the University of Manchester for their help during the preparation of this paper.

References

1. G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, **30**, 1967, pp. 483–485.
2. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.
3. U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, **81**(2), Feb. 1993, pp. 211–243.
4. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, **29**(12), Dec. 1996, pp. 78–82.
5. W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, **3**(6), Nov. 1992, pp. 643–656.
6. F. Bodin and M. O'Boyle. A Compiler Strategy for Shared Virtual Memories. In B. K. Szymanski and B. Sinharoy (Eds.), *Languages, Compilers and Run-Time Systems for Scalable Computers*, Kluwer Academic Publishers, 1996, pp. 57–69.
7. F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, and D. Gannon. Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000. In *Proceedings of the 1990 International Conference on Supercomputing*, ACM Press, pp. 401–413.
8. 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.
9. H. Burkhart and R. Millen. Performance-Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, **38**(5), May 1989, pp. 725–737.
10. E. A. Carmona and M. D. Rice. Modeling the Serial and Parallel Fractions of a Parallel Algorithm. *Journal of Parallel and Distributed Computing*, **13**(3), Nov. 1991, pp. 286–298.
11. M. E. Crovella and 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.
12. M. A. Driscoll and W. R. Daasch. Accurate Predictions of Parallel Program Execution Time. *Journal of Parallel and Distributed Computing*, **25**(1), Feb. 1995, pp. 16–30.
13. T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, **28**(11), Nov. 1995, pp. 47–56.
14. T. Fahringer. Estimating Cache Performance for Sequential and Data Parallel Programs. In B. Hertzberger and P. Sloot (Eds.), *High-Performance Computing and Networking*, Springer-Verlag, Lecture Notes in Computer Science **1225**, 1997, pp. 840–849.
15. T. L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall International, 1992.
16. D. Grunwald and S. Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. Technical Report CU-CS-703-94-93, Department of Computer Science, University of Colorado, Sep. 1993.
17. N. J. Higham and P. Papadimitriou. A parallel algorithm for computing the polar decomposition. *Parallel Computing*, **20**(8), 1994, pp. 1161–1173.

18. T. Mavroudakis. *Parallelisation of the Barotropic Model for Numerical Weather Prediction.* MSc Thesis, Department of Mathematics, University of Manchester, 1996.
19. M. F. P. O'Boyle and J. M. Bull. Expert Programmer versus Parallelizing Compiler: A Comparative Study of Two Approaches for Distributed Shared Memory. *Scientific Programming*, **5**(1), Spring 1996, pp. 63–88.
20. P. Papadimitriou. *Parallel Solution of SVD-Related Problems, with Applications.* PhD Thesis, Department of Mathematics, University of Manchester, 1993.
21. M. Parashar and S. Hariri. Compile-Time Performance Prediction of HPF/Fortran 90D. *IEEE Parallel & Distributed Technology*, **4**(1), Spring 1996, pp. 57–73.
22. C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing* (Vol. II Software), The Pennsylvania State University Press, pp. 39–48.
23. G. D. Riley, J. M. Bull, and J. R. Gurd. Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics. In *Proceedings of the 1997 International Conference on Supercomputing*, ACM Press.
24. R. Sakellariou. *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations.* PhD Thesis, Department of Computer Science, University of Manchester, 1996.
25. R. Sakellariou and J. R. Gurd. Compile-Time Minimisation of Load Imbalance in Loop Nests. In *Proceedings of the 1997 International Conference on Supercomputing*, ACM Press.
26. E. A. Stöhr and M. F. P. O'Boyle. A Graph Based Approach to Barrier Synchronisation Minimisation. In *Proceedings of the 1997 International Conference on Supercomputing*, ACM Press.
27. L. Vuurpijl, T. Schouten, and J. Vytopil. Performance Prediction of Large MIMD Systems for Parallel Neural Network Simulations. *Future Generation Computer Systems*, **11**(2), Mar. 1995, pp. 221–232.
28. M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.
29. X. Zhang, R. Castañeda, and E. W. Chan. Spin-Lock Synchronization on the Butterfly and KSR1. *IEEE Parallel & Distributed Technology*, **2**(1), Spring 1994, pp. 51–63.
30. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press & Addison-Wesley, 1990.