Increasing the Parallelism of Irregular Loops with Dependences*

David E. Singh[1], María J. Martín[2], and Francisco F. Rivera[1]

[1] Dept. of Electronics and Computer Science, Univ. Santiago de Compostela, SPAIN,
{david,fran}@dec.usc.es
[2] Dept. of Electronics and Systems, Univ. A Coruña, SPAIN
mariam@udc.es

Abstract. In this work, we present new proposals based on the owner-compute rule for the parallelization of irregular loops with dependences. The parallel code increases the available parallelism through the distribution of the statements inside each iteration instead of the whole iterations of the loop. Additionally, our proposal presents as main features the reordering of the layout of the indirection entries, optimizing data locality, and the efficient load balancing. Inspector and executor phases are fully parallel, without synchronizations and uncoupled, allowing the reuse of the information of the inspector. Experimental results on a SGI O2000 system prove that our approach exhibits a high performance, even when compared to well-known parallelization strategies.

1 Introduction

In general, we broadly classify irregular loops into two different families: loops without data dependences among iterations, and loops that present dependences, *doacross* loops. In this work, we focus on the parallelization of irregular *doacross* loops in CC-NUMA shared memory machines. In particular, we restrict ourselves to a generic structure of loops like the one shown in Figure 1(a) where S indirection arrays x_i of size N_x $(1 \leq i \leq S)$ perform arbitrary read or write operations in the accessed array a. In this figure, $\odot$ represents a generic operator that can be different for each indirection. According to the value of the indirection arrays, any kind of dependence can arise. For instance, assuming that $N_x = 3$, $S = 2$, $x_1 = \{4, 1, 4\}$ for read accesses to a and $x_2 = \{1, 2, 2\}$ for write accesses to a, there is a true dependence between the second and the first iterations, and an output dependence between the second and third iterations.

In order to parallelize these kind of loops, different strategies can be applied. One of the most popular approaches is the inspector-executor strategy. The goal of the inspector-executor algorithms is to maximize parallelism and minimize the overhead of the analysis stage. One of the first inspector-executor approaches was proposed by Zhu and Yew [1]. In this proposal loop iterations are divided into subsets called wavefronts, which contain iterations that can be executed in

* This work was funded by CICYT under project TIC 2001-3694.

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 287–296, 2003.

```
DO j = 1, Nx
   ... = a[x1[j]] ⊙ ...
   a[x2[j]] = ...
            ...
   ... = a[xs[j]] ⊙ ...
END DO
```

(a)

```
DO j = 1, Nx
   tmp1 = f(j)
   a[x1[j]] = tmp1
   tmp2 = a[x2[j]]
END DO
```

(b)

```
DO j = 1, Nx
   a[x1[j]] = b[j]
   a[x2[j]] = a[x2[j]] ⊕ b[j]
END DO
```

(c)

Fig. 1. Irregular loops: (a) template, (b) statement grouping, (c) multiple reordering.

parallel. This approach has two limitations: first, the inspector and the executor are tightly coupled and the inspector is not reused across invocations of the same loop. Second, the execution of consecutive reads to the same array entry is serialized. Midkiff and Padua [2] improve this strategy by allowing concurrent reads of the same entry. Saltz, et al [3] propose an alternative solution but restricted to loops with no output dependences. Their inspector and executor are uncoupled. Leung and Zahorjan [4] extend this work to consider output dependences and propose parallel versions for the inspector.

All these proposals exploit iteration-level parallelism, that is, loop iterations are distributed among the available processors and iteration-level synchronizations are performed to guarantee a correct execution order. A different approach can be found in the CYT method [5]. This method is also based on the distribution of the iterations among the processors but it checks for dependences in the operation-level instead of in the iterations level. Operation-level synchronizations are included to fulfill dependences. The main advantage of this algorithm is the extraction of a higher degree of parallelism. A comparison between strategies based on iteration-level and operation-level parallelism is presented in [6]. This work shows in an empiric way that operation-level methods outperform iteration-level methods. In [7], we presented two operation-level strategies for parallelizing *doacross* loops and improving data locality.

In this work, we propose new parallelization techniques that reduces synchronizations and improves even more data locality. Our techniques are based on the distribution of the statements instead of the iterations which increases the available parallelism. They are applicable to *doacross* loops where data dependences between statements are exclusively due to accesses to one array, named a in our example (Figure 1(a)). We define statement $stmt_i^j$ of a loop, as the i^{th} line of the body executed in the j^{th} iteration. In the previous example, dependences arise from statement $stmt_2^1$ to $stmt_1^2$ and from statement $stmt_2^2$ to $stmt_2^3$. Often real codes have statements that do not present indirect accesses. For instance, let's consider the loop of Figure 1(b). For each iteration j, statement $stmt_1^j$ does not have accesses to a but it presents a true dependence with $stmt_2^j$. In these cases, a fusion of statements can be performed, assuming that there are statements with more than one line of code. In the example, statement $stmt_1^j$ can include, for each iteration j, the first two lines, whereas $stmt_2^j$ refer to the third line of the body of the loop. In this way, we have a loop with two statements per iteration.

```
DOALL p = 1, N_p
 DO j = 1, N_x
   IF(x_1[j] ∈ my_block[p]) exec: stmt_1^j
   IF(x_2[j] ∈ my_block[p]) exec: stmt_2^j
     ...
   IF(x_s[j] ∈ my_block[p]) exec: stmt_s^j
 END DO
END DOALL
```

(a)

```
DOALL p = 1, N_p
 DO l = 1, N_slc − 1
   DO i = 1, S
     DO j = ρ_i^slc[l][p], ρ_i^slc[l + 1][p] − 1
        exec: stmt_i^j
     END DO
   END DO
 END DO
END DOALL
```

(b)

Fig. 2. Parallel executors: (a) parallel loop applying the owner compute rule, (b) parallel loop applying the slice sort strategy.

Our final proposal is based on the inspector-executor paradigm. Initially, on run-time, the access patterns are analyzed and the dependence information is collected. Then, in the executor stage, the $S \times N_x$ statements are distributed among the processors. Additionally, the information provided by the inspector is also applied to improve data locality and load balance.

2 Parallelization Strategy

We will use the loop of Figure 1(a) as case of study, where some of the statements inside of the loop could be irregular reduction operations. For this kind of irregular loops we can formulate the following property.

Property 1. Given two statements $stmt_i^j$ and $stmt_{i'}^{j'}$ that perform generic accesses to the same entry of a. If there is not an intermediate statement that accesses to this entry, and $j' > j$, then $stmt_{i'}^{j'}$ can be executed anytime after $stmt_i^j$. □

Then, according to the previous property, the order of the statements can be changed as long as the dependences are maintained. The following sections describe different approaches that exploit this property.

2.1 Owner-Compute Rule Strategy

As starting point, we propose a parallel executor based on the owner-compute rule. The parallel code corresponding to Figure 1(a) is shown in Figure 2(a), where N_p is the number of available processors. Each one has assigned an interval of entries of array a which is denoted as $my\_block[p]$. In the loop execution, each processor traverses all the iterations of the loop checking for local accesses to a in each statement. Given that the statements are accessed in the same order than the original loop, the dependences are preserved. The main advantage of this proposal is the locality in the accesses to a, and its main drawback is the

overhead introduced by the additional checks. In the next section, we propose a new strategy called *slice sort* based on the owner-compute rule, but focused on several additional objectives. First, our technique obtains a better exploitation of the memory hierarchy. Our proposal performs a reordering of the indirection arrays to obtain good locality in the indirection accesses, and to reduce false sharing. Several previous papers [8,9] proof the importance of this factor in the performance of the parallel execution, specially on CC-NUMA shared memory machines. Second, in order to obtain a good performance of the parallel code, a correct load balance must be guaranteed. Our proposal optimizes dynamically the load balance. And third, the checks introduced by the owner-compute rule strategy are eliminated.

2.2 Slice Sort Strategy

The slice sort strategy consists of an inspector that performs the analysis and scheduling of the loop, and an executor that properly runs the parallel code. The purpose of the inspector is the determination, in run-time, of the optimal arrangement of the statements of an irregular loop. Our proposal is based on the owner-compute rule, each processor p is assigned to an interval of a. Only statements accessing to entries of a inside the considered block will be executed. Additionally, the indirection arrays are reordered so that, for each one, each processor accesses consecutive entries, improving further more data locality.

Inspector stage. The inspector stores the statements to be executed for each processor, as well as the execution order needed to fulfill the dependences. This information is stored into a structure of N_{slc} sets called slices. Each slice contains only the statements that can be executed in the considered block without data dependences. The concept of slice is similar to the definition of wavefronts [1] but now the slices are defined for individual statements instead of iterations. The basic parallel inspector is shown in Figure 3.

Initially, in the *analyzing stage*, each processor p traverses all the entries of each indirection, and analyze which of them perform accesses in its assigned interval of a. The private arrays ρ_i^{row} store the number of accesses that are performed in each entry of a by the statements $stmt_i^j$ $(1 \leq j \leq N_x)$. For a given statement that performs an access to a, the minimum index of slice in which it can be assigned is obtained by function $slice()$. Let's assume that ρ^{row} is the set of all ρ_i^{row} $\forall i$, and since the accesses are analyzed in the proper order of execution, we have:

$$slice(\rho^{row}, x_i[j]) = \sum_{k=1}^{S} \rho_k^{row}[x_i[j]] \tag{1}$$

With this values, the shared array ρ_i^{slc} is properly updated. For each processor, the array counts the number of statements $stmt_i^j$ assigned to each slice. Note that any kind of dependence is maintained, even Read after Read dependences.

Subsequently, in the *shifting stage*, the ρ^{slc} arrays are realigned. This operation allows to write, in the next stage, the reordered array in different and

```
DOALL p = 1, N_p
   DO j = 1, N_x                                        % Analyzing stage
     DO i = 1, S
        IF (x_i[j] ∈ my_block[p])
           ρ_i^row[x_i[j]] + +
           ρ_i^slc[slice(ρ^row, x_i[j]), p] + +
        END IF
     END DO
   END DO
   DO k = N_slc, 1                                      % Shifting stage
     DO i = 1, S
        ρ_i^slc[k, p] = accum(x_i, p) + Σ_{j=1}^{k-1} ρ_i^slc[j, p]
     END DO
   END DO
   DO j = N_x, 1                                        % Sorting stage
     DO i = S, 1
        IF (x_i[j] ∈ my_block[p])
           ρ_i^slc[slice(ρ^row, x_i[j]), p]] − −
           x_i^out[ρ_i^slc[slice(ρ^row, x_i[j]), p]]] = x_i[j]
           ρ_i^row[x_i[j]] − −
        END IF
     END DO
   END DO
END DOALL
```

Fig. 3. Parallel inspector algorithm.

contiguous memory regions. Using function $accum()$, the number of entries of each indirection computed by each processor is obtained. More formally, defining $\| \cdot \|$ as the cardinality operator, for a given indirection x_i and processor p:

$$acum(x_i, p) = \| \{x_i[j] \ / \ x_i[j] \in my\_block[p']\} \quad \forall p' < p \quad \forall j \in [1, N_x] \| \quad (2)$$

Once this value is computed, the entries of ρ^{slc} arrays are updated adding the offset of previous processors and previous entries in the same slice. Finally, in the *sorting stage*, the reordered indirection arrays x_i^{out} are generated. Once more, all the accesses are analyzed, but this time in reverse order. Therefore arrays ρ_i^{slc} point to the last element of each slice, and they are decremented as they are being processed. For each entry of x_i, the position in the new array x_i^{out} is given by the current value of ρ_i^{slc}. Note that this procedure allows us to reuse the information of the analyzing stage.

Executor stage. Figure 2(b) shows the parallel executor. As each processor only computes the accesses to the region of a assigned to it, therefore no synchronizations are required. Since all the statements within a slice do not present dependences, they can be executed in parallel. Note that using slices the dependences for each processor are preserved.

The workload can be dynamically balanced changing the size of the block of a assigned to each processor. The workload associated to the p^{th} processor is:

$$load[p] = \sum_{i=1}^{S} (C_i \parallel x_i[j] \ / \ x_i[j] \ \in \ my\_block[p], \ \forall \ j \in [1, N_x] \parallel) \qquad (3)$$

Where C_i represents the computational cost of each statement. Our algorithm obtains, with this equation, the whole cost of the loop. The ideal fraction of workload associated to each processor is obtained dividing the total workload by the number of processors. Therefore, contiguous blocks of entries of a that have the same workload are computed using equation 3. The whole process is done in parallel.

3 Performance Evaluation

We used various tests in order to evaluate the efficiency of our proposal. First, we evaluated the complexity and memory requirements of the inspector. Then, we compared the performance of our proposal with other state-of-the-art approaches, using synthetic and real benchmarks.

The complexity of our inspector is proportional to the number of entries of the indirection arrays. Specifically, it is proportional to the sum of the complexity of its three stages: $\mathcal{O}(N_xS + N_{slc}S + N_xS)$. Typically, $N_x \gg N_{slc}$, so we have $\mathcal{O}(2N_xS)$. If the irregular code presents poor load balance, then a variable block distribution is advisable, so we have to take into account the cost of the load balancer that is $\mathcal{O}(N_xS)$.

In terms of the memory overhead of the executor, we have distinguished two situations. The first one, shown in Figure 1(a), deals only with the indirection arrays that depend on the loop index. In this case, the overhead of the executor is due to ρ^{slc} arrays and it is $N_{slc}N_pS$. The second situation differs in that if more than one array depend of the loop index, it will have to be replicated. Figure 1(c) shows an example where array b is acceded by two statements. As

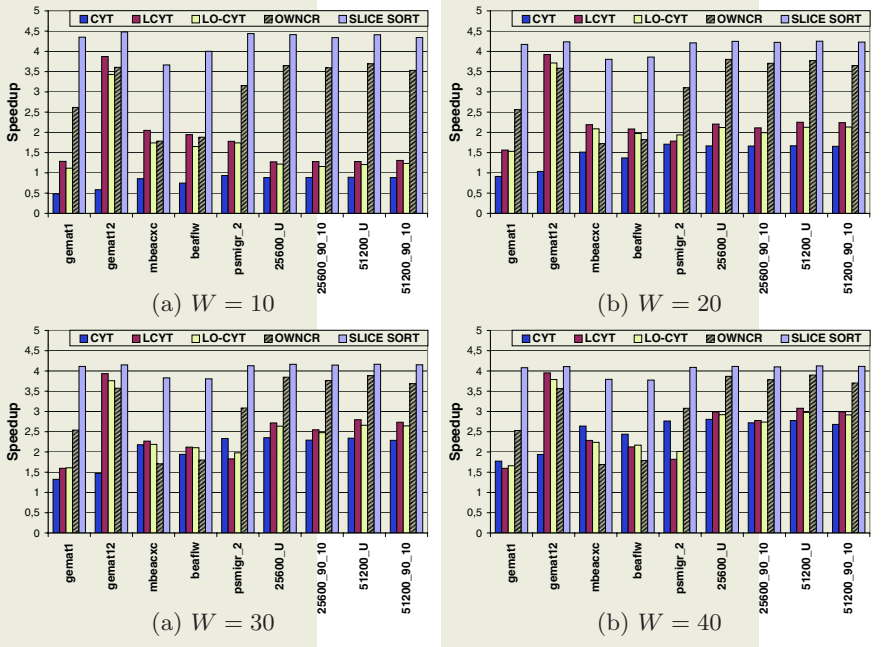
```
DO j = 1, Nx
   DO k = 1, W
      dummy work for stmt1
   END DO
   tmp1 = tmp1 + a[x1[j]]
   DO k = 1, W
      dummy work for stmt2
   END DO
   a[x2[j]] = tmp2
END DO
```

(a) Synthetic code

Matrix	N_x	N_a	CP	N_{slc}
$gemat1$	23684	4929	4938	4928
$gemat12$	16555	4929	49	44
$mbeacxc$	24960	496	487	485
$beaflw$	26701	507	500	495
$psmigr\_2$	270011	3140	2626	2294
25600_U	12800	25600	9	7
25600_90_10	12800	25600	45	35
51200_U	25600	51200	11	9
51200_90_10	25600	51200	46	30

(b) Benchmark matrices

Fig. 4. Synthetic benchmark.



(a) $W = 10$ (b) $W = 20$

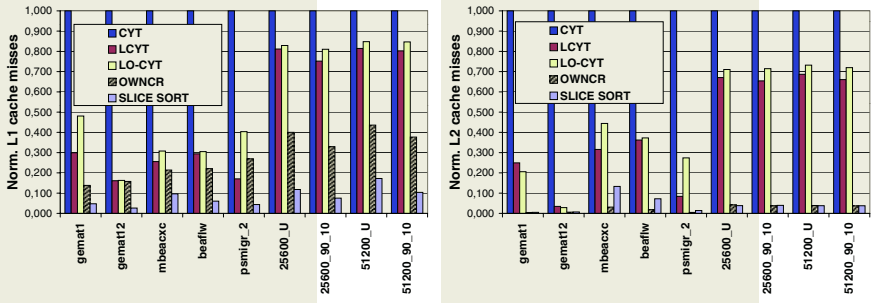
(a) $W = 30$ (b) $W = 40$

Fig. 5. Speedups for different workloads on 8 processors for $W = 10$.

a rule, we have that when one array is acceded by several statements, private copies for each one must be created and each copy must be sorted in the same way that the corresponding indirection array. This can be done in the inspector without any further computational cost. In this way, and for the example, we ensure that when entry $a[x_i^{out}[j]]$ is computed, the correct value of $b_i^{out}[j]$ will be used.

Next, our proposals are compared with the CYT, LCYT and LO-LCYT strategies [5,7]. The target machine is a SGI O2000 with up to eight R10K processors. The benchmarks are written in Fortran, and parallelized using OpenMP directives. As starting point, and following similar approaches taken by other authors [5,10], we have used the synthetic loop of Figure 4(a) in order to evaluate the efficiency of each strategy. We assume that each iteration of the loop has associated two statements; each one includes one irregular access and its associated dummy work. W determines the amount of workload associated to each one. To simplify the study, we assume the same W for both statements.

We have used sparse matrices extracted from real programs [11] as indirection arrays, as well as artificial access patterns that can be classified as uniform and non-uniform. For the uniform ones, denoted with the termination "_U", all the array elements have the same probability of being accessed, whereas in the non-uniform, 90% of references access to 10% of array elements. The table shown in Figure 4(b) contains the main features of the access patterns. N_x is the number of entries of each indirection, N_a is the size of a, and CP is the Critical Path. For the CYT, LCYT and LO-LCYT algorithms the N_x iterations are distributed



(a) L1 normalized cache misses (b) L2 normalized cache misses

Fig. 6. Cache behaviour on 8 processors for $W = 10$.

among the p processors. The CP ($1 \leq CP \leq N_x$) is the length of the largest dependence chain in the loop and gives an estimate of how parallel the loop is. If $CP = 1$ the loop is fully parallel whereas if $CP = N_x$ the loop sequential. For the slice sort strategy, synchronizations are only needed when all the iterations of each slice are processed, so N_S gives an estimate of how parallel the loop is.

Note that the Critical Path is slightly bigger than the number of slices N_{slc}. This is due to the fact that previous proposals take into account dependences between iterations instead of single statements. For instance, considering the loop of Figure 1(a) with $N_x = 3$, $S = 2$, $x_1 = \{1, 2, 3\}$ and $x_2 = \{2, 3, 4\}$, the associated critical path is 3 because each iteration depends on the previous one. However, considering single statements, only two slices are required to preserve the dependences, that is, a better exploitation of the available parallelism is achieved using the slice sort strategy.

Figure 5 shows the speedups for the executor on 8 processors and different workloads. A block distribution of the entries of a was used for the owner-compute rule approach. This technique presents good data locality in the write accesses to a. It has good performance with synthetic matrices because they are well balanced. However, this performance exhibits an important drop down for less balanced real patterns. In all the cases, the slice sort strategy obtains the best results due to an equilibrated load balance and a good data locality. Figure 6 shows the number of L1 and L2 cache misses, normalized with respect to the CYT strategy, for 8 processors. Note the important reduction obtained with our proposals. Additionally, synchronizations between processors and run-time checks are eliminated. The performance of our proposal decreases as W increases due to a smaller influence of the locality improvements.

Table 1 shows the overall performance of the slice sort strategy taking into account the overhead of the inspector. Specifically, it contains the number of times the inspector needs to be reused to be faster than the others. For example, for matrix *gemat1*, two iterations of our executor are faster than two iterations of the owner-compute rule approach, including the overhead of the inspector of our proposal. A zero entry means that our proposal is faster than the other proposal even from the first iteration and taking into account the overhead of

Table 1. Threshold in iterations for outperforming the rest of the proposals.

Matrix	W=10				W=30			
	OWNCR	CYT	LCYT	LO-LCYT	OWNCR	CYT	LCYT	LO-LCYT
*gemat*1	**2**	0	0	0	0	0	0	0
*gemat*12	**5**	0	**5**	**2**	**2**	0	**4**	**2**
mbeacxc	**1**	0	**1**	0	0	0	0	0
beaflw	**1**	0	0	0	0	0	0	0
psmigr_2	**5**	0	**1**	**1**	**2**	0	0	0
25600_*U*	**7**	0	0	0	**6**	0	0	0
25600_90_10	**7**	0	0	0	**5**	0	0	0
51200_*U*	**9**	0	0	0	**8**	0	0	0
51200_90_10	**7**	0	0	0	**4**	0	0	0

the inspector. Note that, in general, the threshold is not high, and decreases as W increases. This threshold is more important for the owner-compute rule approach because it does not use an inspector.

Table 2. Input data and speedups for the PLTMG benchmark.

Characteristics				Speedup on 4 processors			Speedup on 8 processors		
Matrix	N_x	N_a	N_{slc}	OWNCR	ARRAYEXP	SLCSRT	OWNCR	ARRAYEXP	SLCSRT
*test*1	7500	1300	9	0.65	0.65	1.88	0.75	0.72	2.32
*test*2	21090	3595	9	0.55	0.80	2.25	0.76	0.91	3.21
*test*3	90480	15240	9	0.79	1.10	2.91	1.93	1.46	2.52

Table 3. Input data and speedups for the BDNA benchmark.

Characteristics				Speedup on 4 processors		Speedup on 8 processors	
Matrix	N_x	N_a	N_{slc}	OWNCR	SLCSRT	OWNCR	SLCSRT
*test*1	1520	5830	1	1.1	2.1	1.5	1.7

We also have evaluated the efficiency of our proposal with two real applications. The first one was extracted from the *mtxmlt* subroutine of PLTMG Edition 7.1 [12]. This software solves elliptic partial differential equations in general regions of the plane. Although data dependences related to this loop can be overridden by the array expansion parallelization technique, we have found two reasons to include it in our benchmarks. First, it allows us to compare our proposal with other high-competitive approaches. Second, as in other applications, it is mandatory to execute the loop following the sequential order. In an out-of-order execution, like array expansion, the final result can be slightly different from the original. Our proposals assures the correct results. We made tests for three different problem sizes. Their main features are shown in Table 2. This table also shows the obtained speedups for the executor on 4 and 8 proces-

sors. Note that the slice sort approach is the only one that obtains significant speedups.

The second benchmark was extracted from the Perfect Club Benchmarks [13]. Specifically, it corresponds to the loop 711 of the *correc* routine of the BDNA application. Given that access pattern is unknown at compiler-time, dependence analysis have to be performed at run-time. In this stage, our inspector determines that for the input data considered the loop is fully parallel. In [5,10] this loop was tested for the CYT algorithm, and poor speedups were obtained. LCYT and LO-LCYT can not be applied as more than one write access is considered. Results obtained with our proposals for the executor stage are shown in Table 3. Due to the small size of the problem, the available parallelism is low, even though our proposal obtains significant speedups.

References

1. C-Q. Zhu and P-C. Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Trans. on Software Engineering*, 13(6):726–739, (1987).
2. S P Midkiff and D A Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, (1987).
3. J H Saltz, R Mirchandaney, and K Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, (1991).
4. S-T Leung and J Zahorjan. Restructuring Arrays for Efficient Parallel Loop Execution. Technical Report 94-02-01, Department of Computer Science and Engineering, University of Washington, (1994).
5. D-K. Chen, J. Torrellas, and P-C. Yew. An Efficient Algorithm for the Run-Time Parallelization of DOACROSS Loops. In *Supercomputing Conference*, pages 518–527, Washington DC, (1994).
6. C. Xu. Effects of Parallelism Degree on Run-Time Parallelization of Loops. In *31^{st} Hawaii Int'l Conference on System Sciences*, Kohala Coast, HI, (1998).
7. María J. Martín, David E. Singh, Juan Touriño, and Franciso F. Rivera. Exploiting Locality in the Run-Time Parallelization of Irregular Loops. In *Proceedings of the 31th International Conference on Parallel Processing.*, pages 17–22, (2002).
8. H. Han and C-W. Tseng. Improving Locality for Adaptive Irregular Scientific Codes. In *13^{th} Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 173–188, Yorktown Heights, NY, (2000).
9. J. M. Mellor-Crummey, D. B. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *ACM Int'l Conference on Supercomputing*, pages 425–433, Rhodes, Greece, (1999).
10. C. Xu and V. Chaudhary. Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):433–450, (2001).
11. Iain S. Duff, Roger G. Grimes, and John G. Lewis. *Users' Guide for the Harwell-Boeing Sparse Matrix Collection.* Boeing Computer Services, (1992).
12. R. E. Bank. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 7.0.* SIAM, Philadelphia, (1994).
13. M. Berry et al. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *Intl. Journal of Supercomputer Applications*, 3(3):5–40, (1989).