



ELSEVIER

Parallel Computing 26 (2000) 1709–1738

PARALLEL
COMPUTING

www.elsevier.com/locate/parco

Automatic parallelization of irregular applications [☆]

E. Gutiérrez, R. Asenjo, O. Plata ^{*}, E.L. Zapata

Department of Computer Architecture, University of Málaga, P.O. Box 4114, E-29080 Málaga, Spain

Received 6 December 1999; received in revised form 9 May 2000

Abstract

Parallel computers are present in a variety of fields, having reached a high degree of architectural maturity. However, there is still a lack of convenient software support for implementing efficient parallel applications. This is specially true for the class of irregular applications, whose computational constructs hardly fit current parallel architectures. In fact, contemporary automatic parallelizers produce, in general, poor parallel code from these applications. This paper discusses techniques and methods to help improve the quality of automatic parallel programs. We focus on two issues: parallelism detection and parallelism implementation. The first issue refers to the detection of specific irregular computation constructs or data access patterns. The second issue considers the case that some frequent construct has been detected but has been sub-optimally parallelized. Both issues are dealt with in depth and in the context of sparse computations (for the first issue) and irregular histogram reductions (for the second issue). © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Automatic parallelization; Irregular problems; Parallelism detection; Sparse matrix computations; Irregular reductions; Distributed shared-memory architectures

[☆] This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03 and the Esprit IV Working Group of the European Union under contract no. 29488 (Automatic Performance Analysis: Resources and Tools, APART).

^{*} Corresponding author.

E-mail addresses: eladio@ac.uma.es (E. Gutiérrez), asenjo@ac.uma.es (R. Asenjo), oscar@ac.uma.es (O. Plata), ezapata@ac.uma.es (E.L. Zapata).

1. Introduction and background

This decade is seeing the popularization of parallel computers. Nowadays, these machines can be found in many different fields, in industry, research, academic and commercial places. The architecture of parallel computers has reached a high degree of maturity, and many people agree that parallel computing is an effective tool for solving a large variety of difficult problems. However, developing efficient parallel applications for these machines is still a difficult task.

One part of the problem is that parallel computers are architecturally complex. Contemporary general-purpose multiprocessors may be classified into two large classes: private memory and shared-memory machines [12]. Private memory multiprocessors present a high-bandwidth, high/medium-latency communication network, which is efficient for large and infrequent messages. Hence, exploiting private memory locality is important in order to minimize network communications. In addition, as these machines lack from a global or shared memory, parallel tasks should be provided with a simple and efficient mechanism to locate data distributed across processors' local memories. Shared-memory multiprocessors, on the other hand, provide a global physical memory address space, which facilitates the location of data. However, most of these machines implement a cache coherence protocol in hardware, that takes charge of data communications among processors at the cache block level. Exploiting cache locality, thus, is important in order to minimize cache interventions and invalidations, and get efficiency from these machines.

The second part of the problem is that there is some lack of convenient software support for implementing applications efficiently, that is, that produces parallel software able to exploit the above complex architectural features. This is a general situation that deserves to be analyzed in more detail. There is a large class of numerical applications, called *regular problems*, that exhibit a regular structure. Computationally, these problems are characterized by the following property. Considering the usual case in which data are organized as arrays, if two different array elements are data dependent then there is typically some simple relationship between the corresponding array indices, often a linear function analyzable by the compiler. These applications are usually easy to parallelize, either manually or automatically, making efficient use of the processor cycles and the memory hierarchy of the current multiprocessor architectures.

Many important scientific/engineering applications, however, show an irregular structure, and are known as *irregular problems*. These problems arise in sparse matrix computations, computational fluid dynamics, image processing, molecular dynamics simulations, galaxy simulations, climate modeling, optimization problems, etc. [28] The dependence graph for these applications depends on the input data, and so they exhibit an irregular and unpredictable run-time behavior, that does not fit directly to the architectural features of current multiprocessors. This makes that writing an efficient parallel program becomes a very difficult task.

Much research effort has been (and is being) devoted during this decade (and the previous one) to develop suitable programming tools for parallel computers [7,44]. Two important focuses of this research are in language and in compiler technologies.

Advances in parallel language technology for numerical problems are mainly aimed to enable users to program parallel computers using similar methods to those used in conventional computers. Leaving message passing libraries (like MPI [40]) aside, two standards have been established in recent years. On one hand, high-performance Fortran (HPF) [23,27], that extends the Fortran language with a set of language constructs following the data parallel programming model. This paradigm is based on a single thread of control and a globally shared address space. Parallelism is specified through data distributions, which drive the generation of parallel tasks following the owner compute rule. On the other hand, OpenMP [29], that extends C and Fortran languages with task-parallel shared-memory language constructs. In this model, parallelism is specified by partitioning computations instead of data.

Research in compiler technology is associated, in a first instance, to advances in parallel language technology, as powerful translators are necessary to produce effective parallel machine codes from programs explicitly parallelized using, for instance, the above mentioned standards. However, a step forward is given if the compiler is capable of a full parallelization effort. It is clear that automatic parallelizing compilers lead to smaller development times for writing a parallel program. Basically, these parallelizers are source-to-source translators which are fed with a sequential code which is subsequently restructured and extended with the necessary directives, sentences and communication operations, to produce the parallel version.

There are a variety of automatic parallelizers available today, most of them being developed as academic research projects, like Polaris [11,10], SUIF [21] and PROMIS [39] (which takes much of the technology from the past Parafrase-2 project [30]). There are also some commercial products [37,38].

Nowadays, most parallelizing compilers are able to generate efficient parallel codes from regular applications. However, the same cannot be said for irregular codes or in presence of dynamic data structures. In general cases, these compilers usually produce fully or partially parallel codes using techniques based on the *inspector-executor* model [32]. This model consists of the introduction of code to analyze each data access at run-time and decide if the access is local or remote (*inspector*). In the case of remote accesses, current location of remote data is determined (*localize*) and stored locally by communication routines before the actual computations are performed (*executor*).

Run-time support libraries were developed in order to simplify the implementation of inspectors and executors, like CHAOS/PARTI [31]. PILAR [24], a library developed to provide basic support for the PARADIGM compiler [6], is an improved implementation of the inspector/executor paradigm, as it can exploit the certain regularities that many irregular applications exhibit.

Other run-time techniques were proposed recently, as that based on the speculative execution of irregular loops in parallel [36,43]. Parallel execution of the loop proceeds until a dependence violation is detected. In such case, the execution is interrupted, the state is rolled back to the most recent safe state, the correct access order is enforced and parallel execution is resumed.

Techniques like the above are general enough to be applied to virtually any irregular application. However, due in part to their generality, the efficiency obtained

from the automatically parallelized codes is, in general, poor. Better performance could be obtained if techniques are developed and optimized for special cases of frequent computation structures and/or data access patterns [42,3,25]. The problem is two-fold. First (*parallelism detection*), the parallelizer must be able to detect such computation/data constructs and, this way, take advantage of some important code and problem properties. Second (*parallelism implementation*), some of the techniques used to parallelize currently detected constructs are too general or sub-optimal, and thus must be optimized. The final objective consists of generating automatically competitive (high-quality) parallel code, regarding equivalent manually parallelized versions.

In this paper, we take Polaris as the base parallelizer, due to the excellent results it achieves for a great deal of codes from both the SPEC and Perfect Club benchmarks. However, as many other compilers, Polaris can hardly parallelize any irregular code efficiently. In this paper, we present some of the weakest points of Polaris regarding this issue and propose some techniques to alleviate these deficiencies. As it is commonly done in these cases, we have compared our hand-parallelized versions of some irregular codes with the corresponding Polaris version to find out what should be improved.

The next section discusses the first issue, parallelism detection, in the context of sparse computations. A direct method for solving sparse linear systems has been chosen as a case study, as it presents most of the complexities associated to sparse codes, that is, the presence of dynamic compressed data structures. This example code allows us to determine the weakness of contemporary parallelizers like Polaris to produce effective parallel code from sparse programs. However, we will see that a small number of techniques to detect specific computational constructs is enough to generate a high-quality parallel output.

Section 3 is devoted to the second issue, parallelism implementation. In this case, we have selected an irregular (histogram) reduction as the case study. This kind of computation constructs is detected and parallelized by contemporary parallelizers, like Polaris. The problem here, in contrast, lies in the parallel output. Polaris uses array expansion to implement this parallelization, a simple and general technique. However, array expansion was designed for small scale shared-memory multiprocessors. The memory overhead it exhibits is too expensive for large machines. We will discuss a different implementation for such reductions that is free of such scalability problem.

2. Parallelism detection: Sparse matrix computations

When using a direct method for solving a large sparse system of linear equations [15,18], the coefficient sparse matrix is transformed, or factorized, an operation that may change the fill of the matrix. The compact representation of the matrix must take into consideration this fact. Also, row and/or column permutations of the coefficient matrix (pivoting) are usually accomplished in order to assure numerical

stability and reduce fill-in (preserve sparsity rate). All these features make direct methods hard to parallelize efficiently.

In this section we take the sparse LU factorization as a representative transformation which is used in many sparse direct methods. The next section briefly introduces the LU factorization and presents a general Fortran code for the sparse LU. In Section 2.2 we compare the hand-parallelized and the automatic-parallelized (Polaris) versions of the above serial code. An analysis of the resulting parallel codes allows us to determine the weak points of Polaris when dealing with this class of codes, and to propose some techniques (Section 2.3) in order to increase the efficiency of the Polaris output code. The same techniques can be used for other sparse irregular codes, as we discuss in Section 2.4.

2.1. Sparse LU factorization

The LU factorization is used for the conversion of a general system of linear equations to triangular form via Gauss transformations [18]. The factorization of a coefficient n -by- n matrix A results in two n -by- n matrices, L (lower triangular) and U (upper triangular), and two permutation vectors π and ρ such that $A_{\pi_i \rho_j} = (LU)_{ij}$.

There are different strategies to compute the LU factorization [13]. The approach followed in this section corresponds to the right-looking (or sub-matrix based) generic method. We discarded the left-looking approach due to its lack of parallelism [1]. The algorithmic structure of this method is sketched in Fig. 1 (in-place factorization of the matrix A). In the outer iteration k , a pivot is chosen, column and row permutations may be performed so that the pivot occupies the (k, k) matrix position, and, finally, the sub-matrix defined by the pivot is updated (that is, entries $(k+1 : n, k : n)$ of A).

In the case of a sparse coefficient matrix, it is usual to represent it by a compressed format. These formats do not store zero entries of the matrix, with the aim of saving

```

do k = 1, n
  Find pivot = A(i,j)
  if (i .ne. k) then
    swap (A(k,1:n), A(i,1:n))
  endif
  if (j .ne. k) then
    swap (A(1:n,k), A(1:n,j))
  endif
  A(k+1:n,k) = A(k+1:n,k) / A(k,k)
  do j = k+1, n
    do i = k+1, n
      A(i,j) = A(i,j) - A(i,k)A(k,j)
    enddo
  enddo
enddo

```

Fig. 1. Right-looking LU algorithm.

both memory and computation overhead. One of the most used formats of this kind is the compressed column storage (CCS) [8]. This format represents a sparse matrix A as a set of three vectors. The first vector stores the non-zero values of the matrix (fill-ins), the second one stores the row indices of the entries in the first vector, and the third one stores the locations in the first vector that starts a column of A . Fig. 2 shows the CCS representation of a sample matrix A . Array A stores the non-zero matrix entries, R stores the row indices and $CPTR1$, the pointers to the beginning of each column. An additional pointer array, $CPTR2$, used during computation to point to the pivot row, is also present.

Taking again the algorithmic structure presented in Fig. 1, the fill-in which may take place during the update of the reduced sub-matrix ($A(k+1:n, k+1:n)$) can be managed by moving columns in the CCS data structure. In order to minimize the memory traffic due to this column movement, a previous analysis stage may be carried out. This analysis phase reorders the matrix A , selecting the pivots which ensure the numerical stability and preserve the sparsity. It also selects the outer iteration k at which it is worthwhile to switch from a sparse code to a dense one (when the reduced sub-matrix is, for instance, 15% dense [14,5]). After this analysis stage, the factorize stage comprises a sparse phase followed by the switch to a dense factorization stage, based on level 2 (or 3 for block-cyclic distribution) BLAS. To further reduce the column movement, the sparse LU code is designed to be not in-place. At the end of the factorization, hence, the output matrix LU appears stored in a CCS data structure, as depicted in Fig. 3. Here, $FPTR2$ points to the pivots of matrix LU .

We can, now, briefly describe the sparse LU algorithm. The main problem we need to solve is how to manage the new entries which appear during factorization

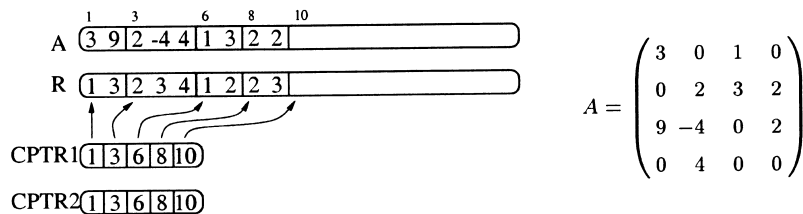


Fig. 2. CCS representation of matrix A .

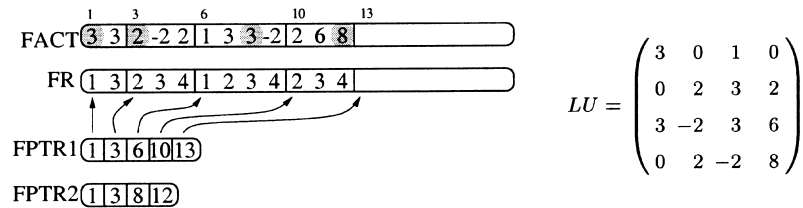


Fig. 3. CCS representation of matrix LU .

(fill-in). Due to the compressed data structure, during the update we have to move each column of the reduced sub-matrix to other memory zone in which the column can fit. To make a long story short, for each outer iteration k , we copy the pivot column, k , to the CCS data structure representing LU (arrays `FACT`, `FR` and `FPTR1`), after dividing it by the pivot. Fig. 4 shows this piece of code. The remaining columns are updated and moved from one half of the arrays `A` and `R` to the other half, properly updating the `CPTR1` and `CPTR2` pointer arrays, as shown in the code of Fig. 5. After the first *IterSwitch* iterations, the code switches to a dense factorization code for the remaining $n - \text{IterSwitch}$ iterations.

2.2. Hand-coded and automatic parallelization

The previously described code is easy to parallelize by hand. All loops but the outermost one, k , are parallel. Summarizing, to run over a $P \times Q$ processor mesh, the hand-parallel algorithm just needs [1]:

- An efficient and uniform distribution of the sparse coefficient matrix, like the BCS distribution scheme [4,2,42]. This scheme is a cyclic distribution of the compressed representation of the sparse matrix. First, the compressed data are expanded to the full matrix form. Then, the full matrix is cyclically distributed on the processors, as if it were a dense matrix. Finally, the sparse local matrices are stored using the CCS compact format (independently one for another). Both row index and column pointers are local. When changing the data structure (from CCS to a dense array) prior to the dense factorization, the dense array remains automatically

```

      DO 20 I=1,N                                !Initialization
        CPTR2(I)=CPTR1(I)
20    CONTINUE
      EYE=1
      DO 100 K=1,IterSwitch                      ! Outermost loop
        FPTR1(K)=EYE                             ! Start of U
        DO 30 I=CPTR1(K),CPTR2(K)-1
          FACT(EYE)=A(I)
          FR(EYE)=R(I)
          EYE=EYE+1
        30    CONTINUE
        FPTR2(K)=EYE                             ! End of U
      C Copying the pivot
        AMUL=1/A(CPTR2(K))
        FACT(EYE)=AMUL
        FR(EYE)=R(CPTR2(K))
        EYE=EYE+1
        F1=EYE                                   ! Start of vector x in SAXPY (y=y-ax)
      C Division of the rest of the column by the pivot
        DO 40 I=CPTR2(K)+1,CPTR1(K+1)-1
          FACT(EYE)=A(I)*AMUL
          FR(EYE)=R(I)
          EYE=EYE+1
        40    CONTINUE
        F2=EYE-1                                ! End of x vector
      .
      .
      .

```

Fig. 4. Sparse F77 LU factorization (normalization of the pivot column).

```

      •
      •
      •
C Reduced submatrix update
  SHIFT=MOD(K,2)*LFACT+1
  DO 50 J=K+1,N
    C1=SHIFT
    DO 60 I=CPTR1(J),CPTR2(J) ! At least one iteration.
      A(SHIFT)=A(I)
      R(SHIFT)=R(I)
      SHIFT=SHIFT+1
60    CONTINUE

    C2=SHIFT-1
C Copy the entry (k,j)
    IF (R(C2).EQ.K) THEN
      AMUL=A(C2)
      C2=C2+1
C Scatter of the rest of the column
      DO 70 I=CPTR2(J)+1,CPTR1(J+1)-1
        W(R(I))=A(I)
70      CONTINUE
C Update
      DO 80 I=F1,F2
        W(FR(I))=W(FR(I))-(AMUL*FACT(I))
80      CONTINUE
C Gather
      DO 90 I=1,N
        IF (W(I).EQ.0) GOTO 90
        A(SHIFT)=W(I)
        R(SHIFT)=I
        SHIFT=SHIFT+1
        W(I)=0
90      CONTINUE
      ELSE
        DO 95 I=CPTR2(J)+1,CPTR1(J+1)-1
          A(SHIFT)=A(I)
          R(SHIFT)=R(I)
          SHIFT=SHIFT+1
95      CONTINUE
      END IF

      CPTR1(J)=C1
      CPTR2(J)=C2
50    CONTINUE
      CPTR1(N+1)=SHIFT
      IF (SHIFT.GT.((MOD(K,2)+1)*LFACT)) STOP !Error: need more space
100  CONTINUE

```

Fig. 5. Sparse F77 LU factorization (update of the reduced sub-matrix).

distributed in a dense cyclic fashion. That way, the switch wastes a negligible amount of time.

- Two communication stages. The first one broadcasts the pivot row by columns of the mesh. The second one broadcasts the pivot column by rows of the mesh.
- The cutting down of the iteration space of the parallel loops to just traverse the local coefficients of the local matrices.

Table 1 presents the experimental performance of the described hand-parallel LU code obtained on a Cray T3D, for some of the Harwell–Boeing [16] sparse matrices. The message passing interface is provided by the SHMEM library (specifically, the function `shmem_put`) [9].

Table 1

Time (s), speed-up and efficiency of the hand-parallel F77 LU code for different Cray T3D mesh sizes

Matrix	Time (s)				Speed-up			Efficiency (%)		
	Seq.	2 × 2	4 × 4	8 × 8	2 × 2	4 × 4	8 × 8	2 × 2	4 × 4	8 × 8
STEAM2	2.33	0.77	0.34	0.21	3.02	6.85	11.09	75.64	42.83	17.33
JPWH 991	3.73	1.30	0.55	0.34	2.86	6.78	10.97	71.73	42.38	17.14
SHERMAN1	2.18	0.88	0.42	0.26	2.47	5.19	8.38	61.93	32.44	13.10
SHERMAN2	36.22	8.84	2.58	1.02	3.99	14.38	35.50	99.75	87.74	55.48
LNS 3937	157.48	44.20	12.17	4.24	3.56	12.94	37.14	89.07	80.87	58.03

The same sequential sparse LU code (see Figs. 4 and 5) was parallelized using the Polaris automatic parallelizer, with no intervention of the user. In this case, as the Polaris output code contained shared-memory directives, the experiments were conducted on a 16-processor SGI power challenge. Results for a Harwell–Boeing matrix are presented in Table 2. Note first the overhead introduced by the compiler in the parallel execution time (comparing the sequential code with the parallel one executed in a single node), and second that the execution time increases (instead of decreasing) with the number of processors.

There are a number of reasons that justify the above disappointing result. For this irregular code, Polaris only exploits the trivial parallelism of loops 20, 30 and 40 (Fig. 4), by avoiding the induction variable `EYE` and transforming the loop in its closed form. In Fig. 6 we show, as an example, the Polaris output for the loop 40.

It is not difficult to imagine that the bookkeeping introduced in this parallel version of the loop 40 is greater than the speed-up achieved just traversing one sparse column. Actually the most time consuming loop in the code is loop 50, which cannot be parallelized by Polaris. Despite this, inside loop 50, Polaris detects a histogram irregular reduction [33], loop 80 that writes array `W()`. Due to the existence of a subscripted subscript in this reduction, Polaris decides to use array expansion to safely parallelize it. However, the subscript array `FR(I)`, in the range `F1: F2` of the index `I`, contains the row indices of a column of the coefficient matrix, and thus is a permutation array (there are no repeated entries in this section of the array). So, considering this information, loop 80 is a fully parallel loop, and should not suffer from the array expansion overhead. We will return to this implementation in the section about irregular reductions.

In any case, even if we manually parallelize loop 80 and the other loops conservatively serialized by Polaris (loops 60, 70, 90 and 95), it is very difficult to obtain good efficiency. Basically these loops traverse column sections, which do not present enough work to justify their parallelization. The alternative is to parallelize the most time consuming loop, loop 50, which takes care of the reduced sub-matrix update.

Table 2

Time (s) of the Polaris parallel LU code for different processor counts in a SGI power challenge

Matrix	Sequential	1	2	4	8	16
[2pt] LNS 3937	77.14	179.41	350.66	425.49	492.75	710.01

```

      eye1 = eye
C$DOACROSS IF((-1)+cptr1(1+k)+(-1)*cptr2(k).GT.5.OR.29.GT.300).AND.29+
C$& 5*((-1)+cptr1(1+k)+(-1)*cptr2(k)).GT.150),LOCAL(EYE3,I),SHARE(EYE,
C$& CPTR1,K,CPTR2,A,AMUL,EYE1,FACT,R,FR)
CSR$ LOOPLABEL 'DPFAC_do40'
B40-- DO i = 1, (-1)+cptr1(1+k)+(-cptr2(k)), 1
| CSR$ INDUCTION eye3, 'DPFAC_do40', 0, eye1, (-1)+cptr1(1+k)+eye1+(-1)*cptr2(k)
| IF ((-1)+cptr1(1+k)+(-cptr2(k)).NE.i) THEN
|   fact((-1)+eye1+i) = a(cptr2(k)+i)*amul
|   fr((-1)+eye1+i) = r(cptr2(k)+i)
|   eye3 = eye1+i
|   CONTINUE
|   40
|   ELSE
|     fact((-1)+eye1+i) = a(cptr2(k)+i)*amul
|     fr((-1)+eye1+i) = r(cptr2(k)+i)
|     eye = eye1+i
|     1002
|     CONTINUE
|     ENDIF
B40-- ENDDO

```

Fig. 6. Polaris parallel version of the loop 40 of Fig. 4.

All columns in this reduced sub-matrix can be processed independently, but Polaris considers that variables A, R, CPTR1, W and SHIFT may present dependences. Conservatively, Polaris marks loop 50 as sequential. The problem is complex because arrays A and R are accessed by the SHIFT induction variable, which is conditionally incremented (depending on the fill-in) in loop 90.

We will discuss next what should be done to automatically determine that this loop can be executed in parallel.

2.3. Sparse-code parallelization techniques

In order to automatically parallelize loop 50 we need to complete two tasks. First we have to detect that the loop iterations are independent (that is, free of loop-carried true dependences), and second, we need to generate the parallel code by breaking the remaining false dependencies with the proper privatization techniques. Let us discuss in detail both issues.

2.3.1. Parallelism detection: Dependence test

Clearly, inside loop 50 (with index J), the reads in variables A and R are indexed by I, which traverses a column from CPTR1(J) to CPTR1(J+1)-1. On the other hand, writes in these variables are indexed with the induction variable SHIFT. Hence, two conditions must be fulfilled to prove that this loop is parallel:

1. The range of SHIFT must not overlap in different J iterations (no output dependences).
2. The range of SHIFT must not overlap the range of I, (CPTR1(J)-CPTR1(J+1)-1), in any J iteration (no flow or anti-dependences).

The first condition can be proved with the following symbolic analysis: always A and R are written, variable SHIFT is incremented, and therefore it is impossible to write twice in the same position. A more complex analysis can be done by proving that

```

        if(max_i .ge. min_shift .and.
&    max_shift .ge. min_i) then
            parallel = .true.
        else
            parallel = .false.
        end if

C$DOACROSS if(parallel), local(...), share(...)
DO 50 J=K+1,N
    ...
50 CONTINUE

```

Fig. 7. Run-time test before execution of loop 50 in Fig. 5.

SHIFT is monotonically increasing (loop 60 executes at least one iteration due to initial values set in loop 20).

To prove the second condition, we need a run-time test since SHIFT and CPTRL are not known at compile time. This run-time test, shown in Fig. 7, should prove that

- the lowest index for reading array A, min_i, is greater than the greater index for writing A, max_shift, and,
- the greater index for reading array A, max_i, is smaller than the lowest index for writing A, min_shift.

As a first approach, the minimum and maximum values of I can be obtained by traversing the $n - k$ last components of array CPTRL. A wiser option consists of noticing that CPTRL is monotonically increasing, since it is updated with SHIFT, which fulfills this condition. In such a case, we get $\text{min_i} = \text{CPTRL}(K+1)$ and $\text{max_i} = \text{CPTRL}(N+1) - 1$. On the other hand, it is clear that $\text{min_shift} = \text{SHIFT}$, but max_shift should be estimated for the worst case, that is, $\text{max_shift} = \text{SHIFT} + (N-K) * (F2 - F1 + 1) + \text{max_i} - \text{min_i}$.

2.3.2. Parallelization: Automatic privatization

Once we know there are no dependencies, we have to face the generation of the parallel code, that is, to allow the parallel processing of separate sets of matrix columns. However, there are scalar (C1, C2, AMUL, SHIFT) and array (W, A, R) variables which should be privatized in order to process in parallel each set of columns, by breaking remaining loop-carried output and anti-dependences.

Scalar variables C1, C2, and AMUL are easily detected as privatizable. A more complex situation arises for array W. The access pattern to W is schematically presented in Fig. 8. From this pattern, W is privatizable if its values are the same at the beginning of each iteration. Actually, W is at the beginning a zero array, and remains so after each j iteration if values in arrays R and FR are between 1 and n . In order to check the range of values in these arrays we have to test before entering in the outermost loop that values in R are between 1 and n . Then, the following symbolic analysis suffices. FR is copied from R before entering loop 50, and $R(\text{SHIFT})$ is copied from $R(i)$ in this loop, except in loop 90, where $R(\text{SHIFT}) = I$ with I in $1:N$. Therefore the range in R and FR is $(1:N)$.

Regarding arrays A and R, we notice that they are written indexed with SHIFT, which is conditionally incremented in loop 90. In such a case, we cannot transform

```

do i=1,n
  W(i)=0
end do

do j=...
  ...
  do i=...
    W(R(i))=...
  end do
  do i=...
    W(FR(i))=...
  end do

  do i=1,n
    if(W(i).ne.0) then
      ...=W(i)
      ...
      W(i)=0
    end if
  end do
end do

```

Fig. 8. Access pattern to W in loop 50 (Fig. 5).

the induction variable in its closed form to determine at which position each column has to be written before writing the previous one. Here again, we have to privatize the writings in local arrays pA and pR, indexed by the private variable pSHIFT. After the local update process, we need a subsequent copy out step to download parallel local arrays into global ones.

2.3.3. Evaluation of the automatic parallel code

The techniques described previously are not yet implemented in Polaris. However, we have generated what can be seen as the output of a parallelizing compiler implementing those techniques. Fig. 9 shows such output parallel code (loop 50 is the only shown). In this code, local variables are privatized by the `local()` clause in the `DOACROSS` directive. However, pSHIFT is privatized by expansion, depending on the number of processors (obtained through calling to `mp_numthreads()`), because pSHIFT value should live after the parallel loop. Notice that false sharing is avoided in the access to this privatized variable.

In [3], we reported a speed-up of 3.84 for this code in a SGI challenge multi-processor with four 150 MHz R4400 processors, taking the Harwell–Boeing matrix LNS3937 as input. More recent results are presented in Table 3 for the same input matrix and a SGI power challenge with 16 R10000 processors. The first column in this table contains the time of the sequential code with no modification. In the next columns we see the speed-up and efficiency for different processor counts. For 16 processors, efficiency decreases due to load imbalance. This is because columns are block distributed to simplify the copy out stage, and therefore the last processors are more heavily loaded due to the greater fill-in in the right-lower corner of the matrix. Also, notice the overhead introduced by the parallel code (comparing the sequential time with the parallel time in one processor), basically explained by the amount of time consumed in the copy out stage.

```

do i = 1,mp_numthreads()
  psi = lshift(i-1,6)+33
  pSHIFT(psi) = 1
enddo
C$DOACROSS if(parallel),local(pc1,pc2,psi,amul,j,tid,i,ii),
C$%
  share(pA,pW,pR,pCPTR1,pCPTR2)
  DO 50 J=K+1,N
    tid = mp_my_threadnum() + 1
    psi = pSHIFT(lshift(tid-1,6)+33)
    pC1=psi
    DO 60 I=CPTR1(J),CPTR2(J)
      pA(psi,tid)=A(I)
      pR(psi,tid)=R(I)
      psi=psi+1
    60    CONTINUE
    pC2=psi-1
C Copy the entry (k,j)
    IF (pR(pC2,tid).EQ.K) THEN
      AMUL=pA(pC2,tid)
      pC2=pC2+1
C Scatter
      DO 70 I=CPTR2(J)+1,CPTR1(J+1)-1
        pW(R(I),tid)=A(I)
      70    CONTINUE
C Update
      DO 80 I=F1,F2
        pW(FR(I),tid)=pW(FR(I),tid)-(AMUL*FACT(I))
      80    CONTINUE
C Gather
      DO 90 I=1,N
        IF (pW(I,tid).EQ.0) GOTO 90
        pA(psi,tid)=pW(I,tid)
        pR(psi,tid)=I
        psi=psi+1
        pW(I,tid)=0
      90    CONTINUE
      ELSE
        DO 95 I=CPTR2(J)+1,CPTR1(J+1)-1
          pA(psi,tid)=A(I)
          pR(psi,tid)=R(I)
          psi=psi+1
        95    CONTINUE
      END IF
      pCPTR1(J,tid)=pC1
      pCPTR2(J,tid)=pC2
      pSHIFT(lshift(tid-1,6)+33) = psi
    50    CONTINUE

```

Fig. 9. Automatic parallel code for loop 50.

Table 3
Performance of the automatic parallelized code in a SGI power challenge for the HB matrix LNS3937

	Sequential	1	2	4	8	16
Time (s)	77	138	67	31	18	15
speed-up	–	0.56	1.15	2.48	4.28	5.13
Efficiency	–	56%	57%	62%	53%	32%

2.4. Summary of parallelizing techniques

In general, codes that process sparse matrices present some degree of parallelism when traversing rows or columns of the matrix. For instance, we have analyzed

```

      DO 10 I= ...
          ...
          DO 20 J=PTR(I),PTR(I+1)-1
              ...
              [ARRAY(IND(J))] .... [VAL(J)] ....
          ...
      CONTINUE
20  CONTINUE
10  CONTINUE

```

Fig. 10. General sparse CCS/CRS pattern.

numerical codes like sparse matrix addition, multiplication, transposition, and sparse QR transformation, among others, finding that some of the previously described techniques for the LU are also applicable for them. Actually, the sparse LU factorization code is a good case study as it exhibits many of the paradigms we have to face when parallelizing dynamic sparse algorithms (in which the fill pattern changes during computation).

It is really tough to design a compiler able to detect that a sequential code is actually processing a sparse matrix. However, our approach is to teach the compiler to detect some simple code patterns, usually related to the processing of sparse matrices in CCS or CRS format. For instance, all the sparse codes mentioned in the previous paragraph contain a section with the pattern presented in Fig. 10.

In this data access pattern, bracket arrays are optional, and dots represent any valid F77 sentence. When this pattern is present in a code we can assume (with a low probability of error) that this code is dealing with a CRS/CCS compact data structure. In such a case, the compiler can trigger a special compiler pass in order to prove certain properties of these data structures: array PTR is monotonic non-decreasing; IND is a permutation array in the range PTR(I) : PTR(I+1) - 1; and values in IND vector are in the range (1:n), where $n \times n$ is the size of the sparse matrix. If such properties can be proved, it may be possible to exploit the parallelism inherent to traversing different rows or columns. The corresponding semi-automatic approach may consist of just instructing the compiler that it is dealing with a CRS/CCS data structure, via an HPF directive, for instance [42].

Summarizing, to prove these properties we next describe four techniques that can be valid for many of the sparse codes using CRS/CCS compact representations (and also for other codes with similar properties). In addition, and due to the generality of the pattern presented in Fig. 10, each technique will be triggered by a more specific pattern (described for each case).

2.4.1. Monotonicity of loop bound arrays

In Fig. 11, the computational pattern is shown which identifies when this technique should be used. It is clear that all writes in VAL are independent for each iteration I if array PTR is monotonically non-decreasing. More formally, the following expression should be fulfilled:

$$\bigcap_{i=1}^n [PTR(I) : PTR(I+1) - 1] = \emptyset. \quad (1)$$

```

DO 10 I= 1,n
DO 20 J=PTR(I),PTR(I+1)-1
  VAL(J)=....
20  CONTINUE
10  CONTINUE

```

Fig. 11. Computational pattern to trigger the *monotonicity* test.

To prove that a vector, PTR, is non-decreasing in the range (1:n) we distinguish two cases:

- If PTR is not modified during execution, we just need to place a loop after PTR initialization (or during initialization) to check that $PTR(I) \leq PTR(I+1)$ for $I = 1, n$.
- However, if vector PTR is modified during execution, a symbolic analysis is needed to check the variables from which PTR is written. In some cases (as we saw for the LU), the compiler can prove that the condition is true at the beginning of the program and that it is an invariant property through the program (Polaris has been recently extended with a property checker [26] to deal with similar cases).

2.4.2. Non-overlapping of induction variables

Fig. 12 presents two patterns in which an array is indexed via an induction variable inside a loop. To determine if loop I is parallel we consider two situations:

- If the loop can be transformed to a closed form, standard dependence tests are taken into account. Last value techniques should be applied if K lives after the loop [33].
- When this is not possible (may be because `positive_value` is not known at compile time) a range test is needed to prove that the range of K does not overlap with the range of I.

Due to the complexity of this second case, we focus on a particular case presented in loop J of Fig. 12, considering that it is possible to estimate an upper bound of `positive_value` for all iterations. In such a case, to execute loop J in parallel we have to:

1. Find maximum and minimum values of I for the J range. This can be done by traversing array PTR at run-time, or in a cheaper way, if we can prove the monotonicity of PTR by the previous technique.
2. Determine the minimum value of induction variable K and obtain the equation which estimates the upper bound of K.
3. Apply the run-time test which proves that $(\min_I > \max_K) \wedge (\max_I < \min_K)$

<pre> DO 10 I= VAL(K)=VAL(I) K=K + (positive_value) 10 CONTINUE </pre>	<pre> DO 20 J=M, N DO 30 I=PTR(J),PTR(J+1)-1 VAL(K)=VAL(I) ... K=K + (positive_value) 30 CONTINUE 20 CONTINUE </pre>
---	--

Fig. 12. Computational pattern to trigger the *non-overlapping* test.

```

DO 10 I= ...
  DO 20 J=PTR(I),PTR(I+1)-1
    VAL(IND(J))=...
  20 CONTINUE
10 CONTINUE

```

Fig. 13. Computational pattern to trigger the *permutation vector* test.

2.4.3. Permutation vector detection

This test should be applied in cases in which the compiler finds a pattern similar to the one presented in Fig. 10, and previous techniques were not successful (see Fig. 13). If loop *I* cannot be parallelized by previous techniques, we can try to parallelize loop *J*. This is possible if we prove that *IND* is a permutation array in the range $\text{PTR}(I) : \text{PTR}(I+1) - 1$. We cover to cases:

- If array *IND* is written just once during initialization, we only need to prove this condition, at run time, after the initialization.
- In other cases, when array *IND* is modified in other program points, a symbolic analysis is needed to prove that values written in *IND* are different in the range $\text{PTR}(I) : \text{PTR}(I+1) - 1$, for all *I*.

2.4.4. Privatization technique extension

As discussed in Section 2.3.2, privatization is a powerful technique to break loop-carried output and anti-dependences. At the same time, it is also a difficult method to use in a general case. We consider here the two cases discussed in the previous Section 2.3.2: privatization of work arrays and privatization of arrays accessed via induction variables.

Regarding the detection of a working array, as *W* in Fig. 8, the problem may be solved as done in Section 2.3.2. However, there are more complex situations like, for example, in sparse matrix addition and multiplication codes. General run-time tests using shadow arrays [35] solve these cases but they are expensive, so more specific and cheaper solutions should be developed.

On the other hand, the privatization of arrays accessed by induction variables, which cannot be transformed to its closed form, is an easier problem. The automatic generation of the copy out operation does not present too much trouble either, as it can be seen as a particular reduction case [33].

3. Parallelism implementation: Irregular reductions

Irregular reduction operations are frequently found in the core of many large scientific and engineering applications. Fig. 14 shows simple examples of reduction loops (*histogram* reduction [34]), with a single reduction vector, *A*(), updated through single or multiple subscript arrays, *f1*(), *f2*(). Due to the loop-variant nature of the subscript array/s, loop-carried dependences may be present at run-time (if it is not a permutation array). It is usual that this reduction loop is executed many

<pre> real A(ADim) integer f(fDim) do i = 1, fDim r = function(i,f(i)) A(f(i)) = A(f(i)) + r enddo </pre>	<pre> real A(ADim) integer f1(fDim), f2(fDim) do i = 1, fDim r = function(i,f1(i),f2(i)) A(f1(i)) = A(f1(i)) + r A(f2(i)) = A(f2(i)) - r enddo </pre>
(a)	(b)

Fig. 14. A single (a) and a multiple (b) irregular histogram reduction.

times in an iterative process. The subscript array/s may be static (unmodified) during all the computation, or may change, usually slowly, through the iterative process.

In a shared memory context, academic parallelizers like Polaris and SUIF recognize irregular reductions and parallelize them using the replicated buffer or the array expansion techniques. The first method replicates the reduction array on all the processors. Each processor computes a portion of the reduction on its private buffer. Later a global reduction is obtained by combining all partial reductions, using synchronization to ensure mutual exclusion [20]. Array expansion, on the other hand, expands the reduction vector by the number of threads participating in the computation. This approach does not need any synchronization to obtain the global reduction, performing in general better than the replicated buffer method. However, both techniques have scalability problems due to the high-memory overhead they exhibit.

The next sections will discuss array expansion as one of the state-of-the-art automatic parallelizing techniques to deal with irregular reductions, showing one of its main drawbacks, scalability. As an alternative, we present a new method to parallelize irregular reductions on distributed shared-memory machines, whose efficiency and scalability overcome that of array expansion (and other current available techniques). The mapping of computations is based on the conflict-free write distribution of the reduction vector across the processors. The proposed method could replace array expansion in the implementation of parallel irregular reductions in compilers like Polaris.

3.1. Array expansion

Array expansion is a powerful and simple technique to parallelize irregular reductions. Fig. 15 shows the parallel code for the single reduction loop in Fig. 14, as obtained by the Polaris compiler (actually, a simplified and slightly optimized version is presented), that is, using the array expansion technique. A private copy of the full reduction vector ($A(\)$) is used for each processor. This is accomplished by expanding such vector by the number of threads participating in the computation. The parallel computation is organized around three phases. In phase 1, each processor initializes its own copy of the reduction array ($A_{tmp}(\)$). In phase

```

    real A(ADim), Atmp(ADim,NumThreads)
    integer f(fDim)
    !- Phase 1
    c$omp parallel do
        do p = 1, NumThreads
            do t = 1, ADim
                Atmp(t,p) = 0
            enddo
        enddo
    c$omp end parallel
    !- Phase 2
    c$omp parallel do
        do i = 1, fDim
            r = function(i,f(i))
            Atmp(f(i),MyThreadNum) =
                Atmp(f(i),MyThreadNum) + r
        enddo
    c$omp end parallel
    !- Phase 3
    c$omp parallel do
        do t = 1, ADim
            do p = 1, NumThreads
                A(t) = A(t) + Atmp(t,p)
            enddo
        enddo
    c$omp end parallel

```

Fig. 15. Parallelization of the single reduction of Fig. 14 using array expansion.

2, each processor works in parallel on the reduction loop, updating its private copy of the reduction array. Partial reductions are hence computed. Finally, in phase 3, global reduction values are calculated by combining the private copies $Atmp(\)$ on the global reduction array. With this arrangement, all loop-carried dependences that may exist in the loop due to possible replicated values in the subscript array $f(\)$ are fulfilled.

Array expansion introduces two sources of overhead. Memory overhead due to the private copies of the reduction vector, and computing overhead due to the initialization of such buffers and the final combining of them on the global reduction array (phases 1 and 3 in Fig. 15). These features introduce hard scalability problems in the technique. In fact, array expansion works very well for small shared-memory multiprocessors. However, the same cannot be said if the programmer is interested in solving a large problem on a medium-size or large-scale distributed shared-memory machine.

3.2. Data write affinity with loop-index prefetching

Array expansion is based on the domain decomposition of the histogram reduction loop (that is, the decomposition of the $[1 : fDim]$ domain). This way, and due to the irregular data access pattern to the reduction vector through $f(\)$, private

copies of such vector are needed. Such private buffers can be avoided (and the corresponding initialization and final combination) if the domain decomposition of the loop is substituted for a data decomposition of the reduction vector. The reduction vector may be, for instance, block distributed across the local memories of the distributed shared-memory multiprocessor. Afterwards, the computations of the histogram loop are arranged in such a way that each processor only computes those iterations that update reduction vector elements that the processor owns. Note that the data distribution of the reduction vector may be carried out at compile time (using some compiler or language directive), or at run-time, as a consequence of the arrangement of the loop iterations (that generates particular memory reference patterns).

A simple form to implement this computation arrangement is called *data affiliated loop* in [25]. Each processor traverses all the iterations in the reduction loop (that is, the $[1 : fDim]$ domain) and checks whether the reduction vector element referenced in the current iteration has been assigned to it. In such case, the iteration is executed; otherwise, the iteration is skipped. That is, the reduction in the histogram loop is guarded by a condition.

The above implementation is not efficient for large iteration domains. A better approach consists of exploiting *data write affinity* (DWA) on the reduction vector with the help of a *loop-index prefetching* (LIP) data structure (DWA–LIP technique). The LIP structure keeps track of the set of iterations that write each one of the blocks of the reduction vector. By using such prefetching structure, private buffers are avoided. A similar idea, *LOCALWRITE*, was proposed recently [22], which is a data affinity based compiler and run-time parallelization technique based on the owner compute rule. However, *LOCALWRITE* is not general, as it cannot deal in a uniform and effective way with general multiple reductions.

A simple implementation of a LIP link structure was discussed previously [19]. Here, however, we will explain a more general and efficient implementation. In order to present that, we need some notation. Let us consider a general multiple reduction loop, with many subscript arrays, $f1()$, $f2()$, $f3()$, ... Let b be the block number function for vector $A()$, $b(k) = \lfloor (k-1)T/ADim \rfloor + 1$, where k is an integer number in the range $[1 : ADim]$ and T is the total number of cooperating threads in the system. That is, a block of number r is owned by thread r . Given iteration i of the reduction loop, we define B_{\min} (B_{\max}) as the minimum (maximum) number of all blocks written in such iteration, $B_{\min}(i) = \min\{b(f1(i)), b(f2(i)), \dots\}$, $B_{\max}(i) = \max\{b(f1(i)), b(f2(i)), \dots\}$. The difference between both limits is then $\Delta B(i) = B_{\max}(i) - B_{\min}(i)$.

In order to exploit parallelism from the reduction loop, the iterations are sorted into sets characterized by the pair $(B_{\min}, \Delta B)$. Those sets of iterations of the form $(B_{\min}, 0)$ are data flow independent and thus can be executed in parallel. In general, two sets of iterations, (b_1, db_1) and (b_2, db_2) , are data flow independent if the writing areas in the reduction vector are non-overlapping, that is $b_1 + db_1 < b_2$. Fig. 16(a) depicts a graphical example, for a system with six threads running in parallel executing a reduction loop with two subscript arrays. Iteration i belongs to the set $(2, 1)$, while j is in the set $(4, 1)$. Hence, arrays $f1()$ and $f2()$ write in

blocks 2 or 3 of the reduction vector in iteration i , while write in blocks 4 or 5 in iteration j . As a consequence, both iterations can be safely executed in parallel by two different threads.

The classification of iterations into those sets really represents a reordering of their execution. Fig. 16(b) shows a data structure for storing the above sets of reduction iterations, that represents an implementation of the LIP structure. A three-array linked list structure is used, $\text{init}()$, $\text{count}()$ and $\text{next}()$, where the first two are triangular matrices. The entry $\text{init}(i, j)$ contains the first iteration in the set ($B_{\min} = i, \Delta B = j$), while $\text{count}(i, j)$ contains the total number of iterations in the same set. Array $\text{next}()$ contains links to the rest of iterations of the set, starting from $\text{next}(\text{init}(i, j))$. Note that in the case of a single reduction loop, all entries in both matrices $\text{init}()$ and $\text{count}()$ are null except for the first column.

Fig. 17(a) shows the parallel version of a multiple reduction loop (as shown in Fig. 14 but with multiple subscript arrays) using the DWA–LIP approach taking into account the sets ($B_{\min}, \Delta B$) of reduction iterations. The procedure is as follows. All sets ($B_{\min}, 0$), referenced by the entries in the first column of $\text{init}()$, are executed in parallel. Afterwards, all sets referenced by the even entries in the second column, which are all data flow independent, are executed in parallel, followed by the sets referenced by the odd entries. This scheme is continued in the third and following columns. Part (b) of this figure shows a simple code to compute the LIP linked list structure. This code contains a histogram reduction on the matrix $\text{count}()$. As the size of this matrix is given by the number of threads computing the code, array expansion may be chosen to parallelize this computation with no significant memory overhead.

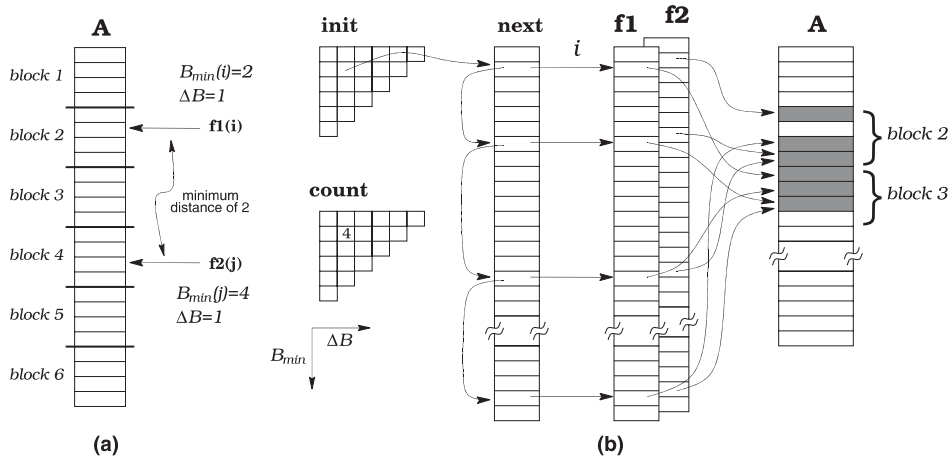


Fig. 16. Example graphical depiction of how two sets of iterations of a multiple reduction loop can be safely executed in parallel (a), and a data structure implementation for storing such sets (LIP) (b) (6 threads and 2 subscript arrays are assumed).

```

real A(ADim)
integer f1(fDim), f2(fDim), f3(fDim), ...
integer init(NumThreads,NumThreads)
integer count(NumThreads,NumThreads)
integer next(fDim)

do dB = 1, NumThreads
  do s = 1, dB
c$omp parallel do
    do B = s, NumThreads-dB+1, dB
      i = init(B,dB)
      cnt = count(B,dB)
      do k = 1, cnt
        r = function(i,f1(i),f2(i),f3(i),...)
        A(f1(i)) = A(f1(i)) + r
        A(f2(i)) = A(f2(i)) + r
        A(f3(i)) = A(f3(i)) + r
        ...
        i = next(i)
      enddo
    enddo
  enddo
c$omp end parallel
enddo
enddo

```

(a)

```

integer prev(NumThreads,NumThreads)

BlockSz = floor(ADim/NumThreads) + 1
do dB = 1, NumThreads
  do B = 1, NumThreads
    count(B,dB) = 0
  enddo
enddo
do i = 1, fDim
  block1 = (f1(i)-1)/BlockSz + 1
  block2 = (f2(i)-1)/BlockSz + 1
  block3 = (f3(i)-1)/BlockSz + 1
  ...
  Bmin = min(block1,block2,block3,...)
  Bmax = max(block1,block2,block3,...)
  dB = Bmax - Bmin
  if ( count(Bmin,dB) .eq. 0 ) then
    init(Bmin,dB) = i
  else
    next(prev(Bmin,dB)) = i
  endif
  prev(Bmin,dB) = i
  count(Bmin,dB) = count(Bmin,dB) + 1
enddo

```

(b)

Fig. 17. Parallel multiple histogram reduction using data write affinity on the reduction vector with LIP (a), and the computation of the prefetching arrays (b).

3.3. Analysis

The DWA with LIP is an efficient technique to extract and exploit parallelism from irregular histogram reductions. Consider the general case of a multiple irregular reduction (Fig. 14 shows particular cases with one and two subscript arrays). Considering a total of T threads cooperating in the parallel execution of the reduction loop, the parallel execution time may be written as

$$T_{\text{par}} = T_{\text{iter}} \left(\frac{Nit_0}{T} + \frac{Nit_1}{\lceil T/2 \rceil} + \frac{Nit_2}{\lceil T/3 \rceil} + \dots \right), \quad (2)$$

where T_{iter} is the (average) execution time of an iteration and Nit_k is the total number of iterations considering all the sets with $\Delta B = k$ (that is, the total sum of all entries in column k of array `count()` in the LIP structure). This simplified expression considers that the number of iterations is uniformly distributed along each column of array `init()`. Similarly, the sequential execution time of the same loop is

$$T_{\text{seq}} = T_{\text{iter}}(Nit_0 + Nit_1 + Nit_2 + \dots). \quad (3)$$

Real applications usually exhibit locality properties in the input domain, which is reflected in the fact that values Nit_k rapidly decrease as k increases. For such applications $T_{\text{par}} \approx T_{\text{iter}}Nit_0/T$ and thus speed-up of the parallel code is near optimum (unless there is a significant non-uniform distribution of iterations writing the different blocks of the reduction vector). A similar conclusion is obtained when only a single irregular reduction appears in the application code, as $Nit_k = 0$ for $k = 1, 2, \dots$

Regarding memory scalability, as no private buffering is needed (as in array expansion), the memory overhead is relatively small and not linearly depending on the number of threads. In general, the extra memory needed to store the LIP data structure has a complexity of $\mathcal{O}(fDim + 2T^2)$, where $fDim$ is the total number of iterations in the reduction loop ($fDim = \sum_{k=0}^T Nit_k$) (see Fig. 14). Frequently, $\mathcal{O}(fDim + 2T^2) \approx \mathcal{O}(fDim)$. Comparatively, the memory overhead complexity for array expansion is $\mathcal{O}(ADim * T)$, which increases linearly with the number of threads.

3.4. Performance evaluation

We have also experimentally evaluated the DWA–LIP parallelization technique on two benchmark codes. The first code is EULER, from the motivating applications suite of HPPF-2 [17], which solves the differential Euler equations on an irregular mesh. We have selected one of the loops computing an irregular reduction inside a time-step loop (see Fig. 18). The basic operation on this code is the computation of physical magnitudes (such as forces) corresponding to the nodes described by a mesh. The magnitudes are computed over the mesh edges, each one defined by two nodes. Therefore two subscript arrays are needed to compute the magnitudes of each edge [3,22]. This reduction loop is interesting from the parallelization point of view because it contains subscripted reads and writes. In order to avoid side effects different from the irregular reductions, all experiments presented in this section only consider one of the reduction loops included in the EULER code. Specifically, the

```

real vel_delta(3,numNodes)
integer edge(2,numEdges)
real edgeData(3,numEdges)
real velocity(3,numNodes)

do i = 1, numEdges
  n1 = edge(1,i)
  n2 = edge(2,i)
  a1 = funct(edgeData(1,i), edgeData(2,i), edgeData(3,i),
             velocity(1,n1), velocity(2,n1), velocity(3,n1))
  a2 = funct(edgeData(1,i), edgeData(2,i), edgeData(3,i),
             velocity(1,n2), velocity(2,n2), velocity(3,n2))
  r1 = a1*velocity(1,n1) + a2*velocity(1,n2) + edgeData(1,i)
  r2 = a1*velocity(2,n1) + a2*velocity(2,n2) + edgeData(2,i)
  r3 = a1*velocity(3,n1) + a2*velocity(3,n2) + edgeData(3,i)
  vel_delta(1,n1) = vel_delta(1,n1) + r1
  vel_delta(2,n1) = vel_delta(2,n1) + r2
  vel_delta(3,n1) = vel_delta(3,n1) + r3
  vel_delta(1,n2) = vel_delta(1,n2) - r1
  vel_delta(2,n2) = vel_delta(2,n2) - r2
  vel_delta(3,n2) = vel_delta(3,n2) - r3
enddo

```

Fig. 18. A loop with multiple irregular reductions from the EULER code.

loop shown in Fig. 18, which corresponds to a single static loop with reductions using two subscript arrays.

The second code we consider in this section is a simple 2D short-range molecular dynamics simulation [41]. This application simulates an ensemble of particles subject to a Lennard–Jones short-range potential. To integrate the equations of motion of the particles, a finite-difference leapfrog algorithm on a Nosé–Hoover thermostat dynamics is used. The core of the code, force contributions calculation, is sketched in Fig. 19. To speed-up such calculations an array of pairs of interactive particles is built, `neigh()`, every THop time steps. During this strip of time iterations, the neighbor list is reused, introducing a write indirection during force computation. At the end of the strip, the neighbor list is updated with the help of a link-cell. Hence, this piece of code represents a single loop with a single subscript array inside. The interesting point here is that the reduction loop is dynamic, as the subscript array is updated periodically.

The experiments have been conducted on a SGI Origin2000 multiprocessor, with 32 250 MHz R10000 processors, a main memory of 8192 MB, and a second-level cache of 4 MB for each processor. The OpenMP shared-memory directives have been used to carry out the parallelization of the loops. The array expansion parallel code used for comparison purposes was obtained using the Polaris compiler. All parallel codes (the DWA–LIP based loops and the Polaris output) were compiled using the SGI MIPSpro Fortran77 compiler (with optimization level 02). The maximum optimization level (03) was not applied because the MIPSpro compiler does not optimize at all the parallel codes when such level is specified.

```

real rx(numPart), ry(numPart)
real vx(numPart), vy(numPart)
real fx(numPart), fy(numPart)
integer neigh(2*numInteract)
do ts = TimeStep, TimeStep+THop-1
  do k = 2, NeighListLength, 2
    i = neigh(k-1)
    j = neigh(k)
    r = distance(rx(i),rx(j),ry(i),ry(j))
    if (r .lt. cutoff) then
      ff = force(r)
      fx(i) = fx(i) + ff
      fy(i) = fy(i) + ff
      fx(j) = fx(j) - ff
      fy(j) = fy(j) - ff
    endif
  enddo
  do i = 1, numPart
    vx(i) = K1 * vx(i) + K2 * fx(i)
    vy(i) = K1 * vy(i) + K2 * fy(i)
    rx(i) = rx(i) + TS * vx(i)
    ry(i) = ry(i) + TS * vy(i)
  enddo
enddo

```

Fig. 19. A loop with multiple irregular reductions from the MD code.

In the case of the EULER kernel, the parallelization performance has been tested on two irregular meshes of different sizes, one with 891 Knodes and the other with 1161 Knodes. Both meshes have a connectivity ($numEdges/numNodes$) of 8. Two versions of each mesh have been tested: colored and sorted. In the first version, an edge-coloring algorithm has been applied, and the edges of the same color are placed consecutively in the indirection array. In this case, a low locality in accesses to the reduction array would be expected. In the second version, the list of edges has been sorted, and therefore a higher locality would be found in the accesses to the reduction vector. The input data to the MD code were generated in a different way. Two sets of pairs positions–velocities, of size 40 and 640 K, were generated representing a uniform realistic ensemble of simple particles in a liquid state.

We show in Fig. 20 the experimental performance of the parallel EULER benchmark code for the colored and sorted versions of the input mesh of size 1161 Knodes. Part (a) of the figure shows the execution time (5 iterations of the time-step loop) of both methods, the array expansion and the proposed DWA–LIP. These times exclude the calculation of the LIP data structure, as this is done only once before entering into the reduction loop (static case). Part (b) shows speed-ups with respect to the sequential code, which was also compiled with optimization level 02 (sequential time is 103.5 and 15.3 s. for the colored and sorted meshes, respectively, for the same 5 iterations of the time-step loop). The DWA–LIP method obtains a significant performance improvement because it exploits efficiently locality when

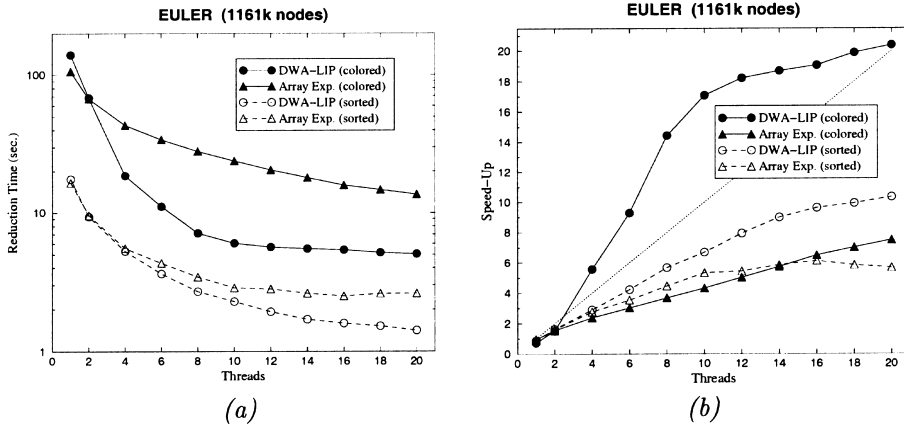


Fig. 20. Parallel execution times (a) and speed-ups (b) for the parallel EULER code using the DWA-LIP and array expansion methods.

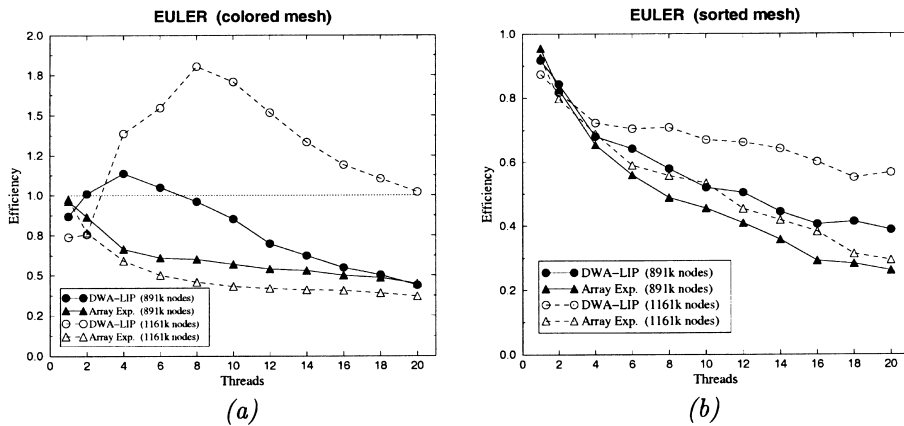


Fig. 21. Parallel efficiencies for the colored (a) and sorted (b) meshes using DWA-LIP and array expansion.

writing in the reduction array. This fact explains a superlinear and sustained speed-up.

Fig. 21 shows the parallel efficiencies for the colored (a) and sorted (b) meshes using the DWA-LIP and array expansion methods. For each class, we present results for two different mesh sizes. The sequential times for the small colored and sorted meshes of sizes 891 Knodes are 51.8 and 11.8 s, respectively. While our DWA-LIP method has good scalability for both orderings of the mesh, the array expansion technique exhibits an anomaly for the colored mesh. Due to the memory overhead of the expanded arrays as well as the low locality presented in the mesh, the efficiency decreases as the mesh size grows.

The sequential time costs of computing the LIP data structure for both mesh sizes are 2.3 (small mesh) and 3.0 s (large mesh). These times are a small fraction of the total reduction time, which can be further reduced by parallelizing the code. In the case of static meshes, the whole program suffers this computing overhead only once, at the beginning of the execution. In the dynamic case, the prefetching must be recomputed periodically. In many realistic situations, however, this updating process is not frequent. For instance, in the code MD it is common to recompute the neighbor list every 10 time steps or so. Then, the prefetching cost is a small fraction of the total time consumed in the reduction iterations executed between two consecutive updatings.

In the EULER code, array expansion has a significant memory overhead due to the replication of the reduction vector in all the processors ($\mathcal{O}(Q * numNodes * T)$, where Q is the number of reduction vectors). The DWA-LIP method also has memory overhead due to the prefetching array ($\mathcal{O}(numEdges + T^2) \approx \mathcal{O}(numEdges)$). In the EULER reduction loop, $Q = 3$ and $numEdges \approx 8 * numNodes$. Hence, the memory overhead of array expansion is larger than that of the DWA-LIP method when the number of threads is greater than 3. In the EULER code, there are reduction loops with $Q = 5$, where the situation is even worse for the array expansion.

The performance obtained in the case of the dynamic irregular reduction code MD is shown in Fig. 22. The experiments were conducted with two different input particle sets, one with 40 K particles and the other with 640 K particles. In addition, some input parameters of the simulations were also changed. For 40 K particles, the size of the link-cell was set to 13.0 units, while the cutoff distance was set to 5.5 units. With these numbers, the neighbor list was really big, having each particle a total of around 169 neighbors (an uniform particle distribution was considered). On the other hand, for 640 K particles, the above parameters were set to 2.9 and 2.5 units, respectively. Now the connectivity is much lower, giving a total of about 8 neighbors

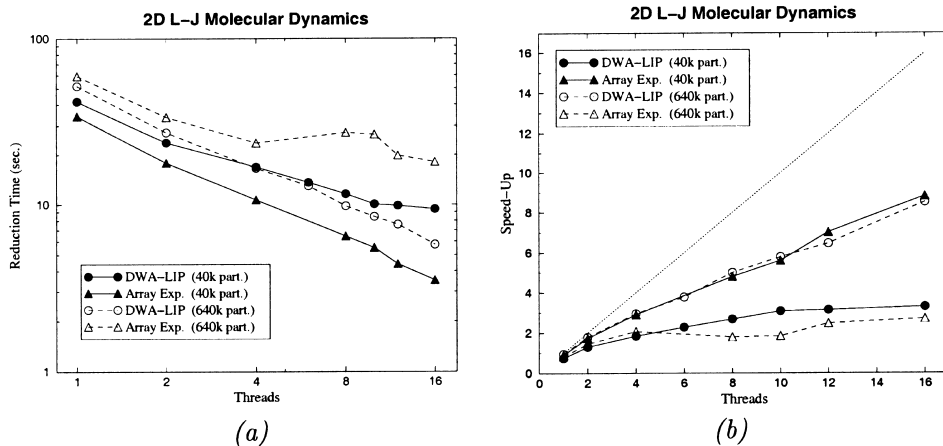


Fig. 22. Parallel execution times (a) and speed-ups (b) for the parallel MD code using the DWA-LIP and array expansion methods.

per each particle. In any case, the size of the simulation box was 250.0×250.0 units, the number of time steps executed was 40 and the neighbor list was updated every 10 time steps. Speed-ups were measured with respect to the sequential version of the code (sequential time was 31 and 49 s for the small and big problems, respectively). Note that DWA–LIP performs much better than array expansion when the size of the problem is big enough. This is an important property, taking into account the better scalability behavior of DWA–LIP. The sequential execution times for the calculation of the LIP data structure were 11 (40 K particles) and 8.5 s (640 K particles).

In general, array expansion has less memory overhead and, sometimes, performs better than DWA–LIP only in very small multiprocessors, and with small problem domains. However, the presented experimental results show that the DWA–LIP method has significant better performance and scalability than array expansion for many realistic situations. The good parallel behavior of DWA–LIP is justified by the fact that, in general, realistic problem domains exhibit short-range relations between data points.

4. Conclusions

In the last few years it has been shown that parallel computing is a powerful tool to solve large and complex computational problems. Automatic parallelization facilitates the development of parallel software by enabling programmers to use familiar programming languages, like C or Fortran, typically used in numerical applications. However, the current state of the technology is not able to obtain efficient parallel code for a large class of scientific/engineering problems, known as irregular applications.

In this paper we have identified and proposed solutions to some of the issues responsible for this poor performance. New compilation techniques have to be incorporated into parallelizers in order to detect (and further parallelize) specific computation constructs and data access patterns, that appear frequently in (and identify) classes of numerical codes, like, for example, sparse codes. And new optimized parallelization techniques must also be developed in order to obtain high-quality parallel code for specific and frequent computational constructs, like, for example, irregular reductions.

Acknowledgements

We gratefully thank David Padua, at the Department of Computer Science, University of Illinois at Urbana-Champaign, for providing us the Polaris compiler, and also Yuan Lin, for the kind help and collaboration, as well as Søren Toxvaerd, at the Department of Chemistry, University of Copenhagen, for providing us the example short-range molecular dynamics program. We also thank the CIEMAT

(Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas), Spain, for giving us access to their Cray T3E multiprocessor, and the Centro de Supercomputación Complutense at Madrid for providing us access to their SGI Origin2000 system.

References

- [1] R. Asenjo, LU factorization of sparse matrices on multiprocessors, Ph.D. Dissertation, University of Málaga, Department of Computer Architecture, December 1997).
- [2] R. Asenjo, G. Bandera, G.P. Trabado, O. Plata, E.L. Zapata, Iterative and direct sparse solvers on parallel computers, Euroconference: Supercomputation in Non-linear and Disordered Systems: Algorithms, Applications and Architectures, San Lorenzo de El Escorial, Madrid, Spain, September 1996, pp. 85–99.
- [3] R. Asenjo, E. Gutiérrez, Y. Lin, D. Padua, B. Pottenger, E.L. Zapata, On the Automatic Parallelization of Sparse and Irregular Fortran Codes, Technical Report 1512, University of Illinois at Urbana-Champaign, CSRD, December 1996.
- [4] R. Asenjo, L.F. Romero, M. Ujaldón, E.L. Zapata, Sparse block and cyclic data distributions for matrix computations, Adv. Workshop in High Performance Computing: Technology, Methods and Applications, Cetraro, Italy, June 1994, pp. 359–377.
- [5] R. Asenjo, E.L. Zapata, Parallel pivots LU algorithm on the Cray T3E, in: Proceedings of the Fourth International Conference of the ACPC, ACPC'99, Salzburg, Austria, February 1999, pp. 38–47.
- [6] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, E. Su, The PARADIGM compiler for distributed-memory multicomputers, IEEE Computer, October 1995.
- [7] U. Banerjee, R. Eigenmann, A. Nicolau, D.A. Padua, Automatic program parallelization, Proceedings of the IEEE 81 (2) (1993) 264–287.
- [8] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, Templates for the solution of linear systems: Building blocks for Iterative methods, SIAM Press, USA, 1994.
- [9] R. Barriuso, A. Knies, SHMEM User's Guide for Fortran, Rev. 2.2, Cray Research, Inc., August 1994.
- [10] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, P. Tu, Parallel programming with Polaris, IEEE Computer 29 (12) (1996) 78–82.
- [11] 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 and Distributed Technology 2 (3) (1994) 37–47.
- [12] D.E. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, Los Altos, CA, 1999.
- [13] J.J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Solving Linear Systems on Vector and Shared Memory Computers, SIAM Press, USA, 1991.
- [14] I.S. Duff, Sparse Numerical Linear Algebra: Direct Methods and Preconditioning, Technical Report RAL-TR-96-047, Rutherford Appleton Lab., Chilton Didcot Oxon OX11 0QX, April 1996.
- [15] I.S. Duff, A.M. Erisman, J.K. Reid, Direct Methods for Sparse Matrices, Oxford University Press, New York, 1986.
- [16] I.S. Duff, R.G. Grimes, J.G. Lewis, Users' Guide for the Harwell-Boeing Sparse Matrix Collection, Research and Technology Div., Boeing Computer Services, Seattle, WA, 1992.
- [17] I. Foster, R. Schreiber, P. Havlak, HPF-2, Scope of Activities and Motivating Applications, Technical Report CRPC-TR94492, Rice University, November 1994.
- [18] G.H. Golub, C.F. van Loan, Matrix Computations, The Johns Hopkins University Press, Baltimore, MD, USA, 1991.

- [19] E. Gutiérrez, O. Plata, E.L. Zapata, On automatic parallelization of irregular reductions on scalable shared memory systems, in: *Proceedings of the Fifth International Euro-Par Conference, Euro-Par'99*, Toulouse, France, August–September, 1999, pp. 422–429.
- [20] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, M. Lam, Detecting coarse-grain parallelism using an interprocedural parallelizing compiler, *IEEE Supercomputing '95*, San Diego, CA, December 1995.
- [21] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, E. Bugnion, M.S. Lam, Maximizing multiprocessor performance with the SUIF compiler, *IEEE Computer* 24 (12) (1996) 84–89.
- [22] H. Han, C.-W. Tseng, Improving compiler and run-time support for irregular reductions, in: *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, NC, August 1998.
- [23] High Performance Fortran Language Specification, Version 2.0, High Performance Fortran Forum, 1996.
- [24] A. Lain, Compiler and run-time support for irregular computations, Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1997.
- [25] Y. Lin, D. Padua, On the automatic parallelization of sparse and irregular fortran programs, in: *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Computers, LCR'98*, Pittsburgh, PA, May 1998.
- [26] Y. Lin, D. Padua, Demand-driven interprocedural array property analysis, in: *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC'99*, San Diego, CA, August 1999.
- [27] P. Mehrotra, J.V. Rosendale, H. Zima, High performance fortran: history, status and future, *Journal of Parallel Computing* 24 (3–4) (1998) 325–354.
- [28] P. Mehrotra, J. Saltz, R. Voigt, *Unstructured Scientific Computation on Scalable Multiprocessors*, MIT Press, Cambridge, MA, 1992.
- [29] OpenMP, A Proposed Industry Standard API for Shared Memory Programming, OpenMP Architecture Review Board, 1997.
- [30] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung, D.A. Schouten, The structure of parafrase-2: an advanced parallelizing compiler for C and Fortran, in: *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC '89*, Urbana, IL, August 1989, pp. 423–453.
- [31] R. Ponnusamy, J. Saltz, A Manual for the CHAOS Runtime Library, Technical Report, UMIACS, University of Maryland, May 1994.
- [32] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, G. Fox, Runtime support and compilation methods for user-specified data distributions, *IEEE Transactions on Parallel and Distributed Systems* 6 (8) (1995) 815–831.
- [33] B. Pottenger, Theory, techniques, and experiments in solving recurrences in computer programs, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, CSRD, May 1997.
- [34] B. Pottenger, R. Eigenmann, Idiom recognition in the Polaris parallelizing compiler, in: *Proceedings of the Ninth ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995, pp. 444–448.
- [35] L. Rauchwerger, Run-time parallelization: a framework for parallel computation, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, CSRD, August 1995.
- [36] L. Rauchwerger, D. Padua, The LRPD Test: Speculative run-time parallelization of loops with privatization and reduction parallelization, *SIGPLAN Conference on Programming Language Design and Implementation* (1995) 218–232.
- [37] MIPSpro Fortran77 Programmer's Guide, Silicon Graphics Inc., SGI Inc., 1994.
- [38] MIPSpro Automatic Parallelization, Silicon Graphics Inc., SGI Inc., 1998.
- [39] H. Saito, N. Stavrakos, S. Carroll, C. Polychronopoulos, A. Nicolau, The Design of the PROMIS Compiler, in: *Proceedings of the Eighth International Conference on Compiler Construction*, Amsterdam, The Netherlands, March 1999.
- [40] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996.

- [41] S. Toxvaerd, Algorithms for canonical molecular dynamics simulations, *Molecular Physics* 72 (1) (1991) 159–168.
- [42] E.L. Zapata, O. Plata, R. Asenjo, G.P. Trabado, Data-parallel support for numerical irregular problems, *Journal of Parallel Computing* 25 (13–14) (1999) 1971–1994.
- [43] Y. Zhang, L. Rauchwerger, J. Torrellas, Speculative parallel execution of loops with cross-iteration dependencies in DSM multiprocessors, in: *Proceedings of the Fifth IEEE International Symposium on High-Performance Computer Architecture*, January 1999.
- [44] H.P. Zima, B. Chapman, Compiling for distributed-memory systems, *Proceedings of the IEEE* 81 (2) (1993) 264–287.