

A Symbolic Analysis Method of Communication Set Generation for Irregular Array Reference

Zhen Liu^{1,*}

Minyi Guo^{2,†}

¹Graduate School of Engineering, Nagasaki Institute of Applied Science,
536 Aba-machi, Nagasaki 851-0193, Japan

²Department of Computer Software, The University of Aizu,
Aizu-Wakamatsu City, Fukushima 965-8580, Japan

Abstract Communication set generation significantly influences the performance of parallel programs. However, seldom works gives attention to the communication generation problem for irregular applications. In this paper, we will explain how support to generate communication set for irregular array references in loops. We propose a compile-time algorithm by introducing some symbolic analysis techniques. In our symbolic analysis system, a set of symbolic solutions of a symbolic expression system is solved by limiting some restrictions. For this proposal, we introduce some symbolic analysis algorithms to fix solutions in a system of equalities and inequalities. Finally, we show experimental results on a parallel computer CM-5 that validate our approach.

Keywords parallel compilers; irregular array references; Communication optimization

1 Background

Parallel compilers are necessary to allow programs written in standard sequential languages to run efficiently on parallel machine. Parallel compilers that generate code for each processor have to compute the sequence of local memory address accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. The distribution of computation in most compilers follows the owner-computer rules. That is, a processor performs only those computations (or assignments) for which it owns the left hand side variable. Access to non-local right hand side variables is achieved by inserting sends and receives.

Communication overhead influences the performance of parallel programs significantly. According to Hockney's representation, communication overhead can be measured by a liner function of the message length m — $T_{comm} = T_s + mT_d$, where T_s is the start-up time and T_d is the per-byte messaging time. Therefore, to achieve good performance, we must optimize communication in following three aspects:

*E-mail: liuzhen@cc.nias.ac.jp

†E-mail: minyi@u-aizu.ac.jp

Example 1

```

L1 : DO i1 = 1, N
      L2 : DO i2 = 1, i1
            LA = i1 * (i1 - 1) / 2 + i2
            IB = i2 * (i2 - 1) / 2 + i1
      S : A(LA) = F(B(IB))
      ENDDO
    ENDDO

```

Figure 1: Sample loop nest with irregular array reference

- (1) to exploit local computation as much as possible,
- (2) to vectorize and aggregate communication to reduce the number of communications,
- (3) to reduce the message length in a communication step.

In order to compile a loop into parallel code efficiently, one must generate the communication set for each processor at compiler-time. If the loop bounds are constants and array subscripts are represented as liner (affine) functions of loop index variables, the problem is similar to compile a typical HPF-style ^[1], assignment statement $A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2)$, where s_1 and s_2 are the access stride of A and B respectively. Given an array statement with HPF-style data mappings, there has been much research to generate the code including the communication for each processor ^{[2][3][4][5]}. Also, the methods to decide data distribution schemes for regular loop nests are discussed by many researchers ^{[2][6]}.

However, if the array subscript expressions are not of the liner form – called non-linear which appears some irregular applications – the above mentioned techniques cannot be applied in this situation.

Consider a loop nest with non-linear array referencing which is very similar to a code excerpt where induction variables are replaced, as found in the Perfect benchmarks ^[10] shown in Figure 1.

In the loop two array reference functions are $f = i_1 * (i_1 - 1) / 2 + i_2$ and $g = i_2 * (i_2 - 1) / 2 + i_1$ respectively. The general affine communication set generation techniques cannot be applied in these kinds of irregular applications, because there is no affine relationship between the array global addresses of LHS and RHS. Communication generation for this kind of issues has not received much attention.

Many researchers have focused on the problem of communication set generation under regular array reference in parallel loop nest, or array statements such as $A(l_1 : u_1 : s_1) = B(l_2 : u_2 : s_2)$ in some data-parallel languages such as HPF and Fortran D ^{[1][5]}, with block-cyclic distribution. For instance, Gupta *et al.* proposed closed forms for representing communication sets. These closed forms are then used with a virtual processor approach to obtain a solution for arrays with block-cyclic distribution ^[2]. Chatterjee *et al.* enumerated the local memory access sequence based on a

finite-state machine (FSM). Their run-time algorithm involves a solution of b_1 linear Diophantine equations to determine the pattern of accessed address, followed by sorting of these address to derive the accesses in a linear order^[3]. Kennedy *et al.* adopted an integer lattice method to generate the memory access sequence^[4]. Recently, Tseng and Gaudlot^[13] derived an algebraic solution for an integer lattice that models the communication set using the Smith-Normal-Form analysis, thus generating the enumeration of the communication set. In this approach, the authors claimed that the SPMD program can be constructed without any inspector-like run-time codes. Paek *et al.*^[14] presented a compiler framework for communication generation that has the potential to reduce the time—consuming hand-tuning that would otherwise be necessary to achieve good performance for non-cache-coherent multiprocessors.

However, seldom work gives attention to the problem of generating communication for irregular access in loop nest. Antonio Lain *et al.*^[6] implemented a library, called PILAR, for exploiting regularity in irregular application. They presented methods for detecting irregularity in array references, as well as the presence of locality in such references, and finally, placement of inspectors and inter processor communication schedules^[6]. The CHAOS/PARTI library^[9], and in particular, the original PARTI library, had a significant impact in the design of PILAR. Similarly, LPARX^[11] is a C++ library that provides run-time support for dynamic, block-structured, irregular problems in a variety of platforms. Our recent work tried to reduce the communication cost of irregular loop partitioning. We partitioned a loop iteration to a processor for which the minimal communication costs is ensured when executing that iteration. Then, after all iterations are partitioned into various processors, we gave global vs. local data transformation rules, indirection array remapping, and communication optimization methods^[15].

We propose some communication optimization techniques for the situation of irregular array references in nest loops. In our methods, the local array distribution schemes are determined such that the total amount of communication messages is minimum. Then, we explain how to support communication set generation at compile-time by introducing some symbolic analysis techniques. In our symbolic analysis system, a set of symbolic solutions of a symbolic expression system is solved by limiting some restrictions. Experiments will be shown that demonstrate the effectiveness of our approach to the parallel compilers.

The rest of this paper is organized as follows: Section 2 motivates the need of communication optimization for irregular array references, and introduces some background knowledge used in the following sections. Section 3 describes how to determine array distribution schemes for irregular loops. Section 4 proposes a symbolic analysis method for communication set generation. Section 5 shows the experimental evaluations. Finally, Section 6 presents the conclusions.

2 Problem Description and Preliminaries

Given a perfectly nested loop L as shown in Figure 2.

For the sake of simplicity, we will assume that the referenced array A and B

```

L1: DO i1 = X1, Y1, Z1
      .....
      Ln: DO i1 = Xn, Yn, Zn
S:      A(f(i1, i2, ..., in)) =
          F(B(g(i1, i2, ..., in)))
          ENDDO
      .....
      ENDDO

```

Figure 2: A perfectly nested loop

have only one dimension. The array access functions (f and g), the loop's lower and upper bounds (X_i, Y_i), and stride (Z_i) may be arbitrary symbolic expressions made up of loop-invariant variables and loop indices of enclosing loops. We will also assume that all loop strides are positive. It is not difficult to extend our method to handle imperfectly nested loops, negative strides, multidimensional arrays, and loop-variant variables. Furthermore, let the array A and B be distributed in a block-cyclic fashion with block sizes of β_1 and β_2 respectively across P processors. This is also known as *cyclic*(β_1) and *cyclic*(β_2) distributions.

We assume that the array access functions f and g are non-linear functions. Non-linear subscript functions are commonly caused by induction variable substitution, linearizing arrays, parameterizing parallel programs with symbolic number of processors and problem sizes, and so forth.

Compilers currently parallelize irregular references using inspector and executor approach. The inspector partitions loop iterations, allocates local memory for each unique non-local array element accessed by a loop, and build a communication schedule to prefetch required non-local data. In the executor phase, the actual communication and computation are carried out. This approach was adopted by CHAOS run-time library [9]. Inspector-executor incurs significant overhead caused by the inspector and mapping non-local indices into local buffers. The inspector must be re-executed each time the access pattern changes.

We want, first, to decide the distribution schemes β_1 and β_2 such that the total communication steps and the amount of message sizes are minimum, because the communication patterns are different for the same loop nest and arrays A and B , according to different data distribution schemes. Then, we will compute the necessary communication sets in each processor due to execution of the above loop. That is, we must be able to obtain an array subscript set for a processor pair (p, q) , named *Send*(p, q): which represents elements of array B sent from q to p . We only generate send communication set in this paper because the receive set generation is very similar to the algorithm for send pattern generation.

In our previous work [7][8], array elements distributed on a specific processor can be represented as a 4-tuple $\delta = (o, b, s, m)$, where o is the starting subscript of the

global array elements distributed on that processor; b the length of the block; s the stride between two consecutive blocks; and m is the number of blocks distributed onto the processor. For instance, if an array with the size 180 is distributed across 4 processors with distribution scheme *cyclic*(3), the data owned by processor p_0 can be expressed as $\delta_0 = (0, 3, 12, 15)$. A δ corresponds to a set of the global array subscript defined as

$$S_\delta = \{i | o + s * k \leq i < o + b + s * k, 0 \leq k < m\}.$$

Intuitively, δ can represent the set of elements of A owned by a processor under any regular distribution. Furthermore, if $\delta(A, p)$ represents 4-tuple of A distributed onto processor p , then $index(A, p)$ can be defined as

$$index(A, p) = \{(i_1, \dots, i_n) | f(i_1, \dots, i_n) \in S_{\delta(A, p)}\}$$

Thus,

$$Send(q, p) = \{g(i_1, \dots, i_n) | (i_1, \dots, i_n) \in Index(A, p) \wedge g(i_1, \dots, i_n) \in S_{\delta(B, q)}\}.$$

These formulas will be used in our symbolic communication generation algorithm. For example, let $N = 15$, $P = 4$, $\beta_1 = 3$, $\beta_2 = 2$. According to Algorithm 1, we can compute all $Send(q, p)$, $0 \leq p, q < 3$. For instance, the elements must be sent from p_3 to p_1 is $Send(3, 1) = \{6, 7, 15, 22, 31, 38\}$.

Although we can get the send communication set through the above algorithm, the rigorous problems are that it can only be invoked at run-time execution when the bounds of nested loops including non-constant; and the communication complexity of the algorithm is $O(y^n)$ (assuming that the average iteration of each loop is y). This is very high cost for the parallel program execution.

3 Decision of Array Distribution Schemes for Irregular Loops

As mentioned in the introduction section, a good data distribution strategy can reduce the number of communication and message length. Assuming that, in an iteration (i_1, \dots, i_n) , if $A[f(i_1, \dots, i_n)]$ and $B[g(i_1, \dots, i_n)]$ are distributed onto the same processor, the statement S in the loop L is executed with local array access.

For the regular loops, based on analysis of affine subscripts, some data distribution techniques are proposed to maximize the local accesses and minimize the remote accesses (inter-node communication), such as constraint-based method^[2], linear algebraic frameworks^[12].

With respect to an irregular reference loop, if it is interlined between two regular loops, the arrays in the irregular loop must be distributed the same as the previous scheme in order to avoid redistribution overhead, because the reduced cost of communication may be larger than the redistribution cost. However, if the irregular loop is absolute or the first loop nest, we must determine the distribution schemes for arrays in the loop to optimize communications.

Given the global address of an array element, we can easily determine the processor that owns this elements and the local address of the element on that processor using the expressions

$$\begin{aligned} Eq_{proc}(p, i) &: p = (i \operatorname{div} \beta) \operatorname{mod} P, \\ Eq_{loc}(L, i) &: L = \beta * (i \operatorname{div} P\beta) + i \operatorname{mod} \beta, \end{aligned}$$

where i is a global address, p the processor to which i belongs, P the number of processors executing the parallel program, L the local address of i on processor p , and β is the distribution scheme of the array. For the purpose of local execution of the statement S in the loop L , we must decide β_1 and β_2 so that the following formula is needed to became true:

$$\exists \beta_1, \beta_2, \max \left(\left| \left\{ (i_1, \dots, i_n) \mid (f \operatorname{div} \beta_1) \operatorname{mod} P = (g \operatorname{div} \beta_2) \operatorname{mod} P, \right. \right. \right. \\ \left. \left. \left. 1 \leq \beta_1, \beta_2 \leq \frac{N_1}{P}, \frac{N_2}{P}, \left(\bar{X} \leq \bar{I} \leq \bar{Y} \right) \right\} \right| \right).$$

The β_1 and β_2 to satisfy the above formula can be easily solved. However, even if at the compile-phase, the solving time may be very long. In order to efficiently decide the distribution schemes, we should pay attention to the fact that because the random access of array elements for irregular applications, the optimal distribution schemes for a small range should also be suitable for a large range. This conjecture is verified by our experiment. For the example1, when N is selected as 500, 1000, and 5000, the β_1, β_2 all show 7 and 3 respectively. Thus, the optimal distribution schemes can be determined at compile-time with a short excerpt of array.

4 Symbolic Analysis Methods for Generating Communication Sets

In order to compute communication sets when array subscripts and loop bounds are symbolic and nonlinear expressions, a symbolic analysis method is proposed. A symbolic expression may consist of arbitrary arithmetic operators and operands that can be array subscripts, loop indices, loop bounds and strides, integer constants and infinity symbol $(-\infty, \infty)$.

A restriction is a set of symbolic qualities and inequalities defined over loop variables and parameters (loop invariant) which are commonly derived from loop bounds, array subscript expressions, conditional statements, data declarations, and data and computation distribution scheme of a program. In this paper, the symbolic variables are designed as loop indices. An integer solution to a set of restriction is a set of loop indices satisfying all of the constraints. There are three sub-restrictions. They are,

- (1) loop bounds and control-flow restriction C_1 ,

(2) array A reference and distribution restriction C_2 ,

(3) array B reference and distribution restriction C_3 .

To generate the communication sets, our goal is to solve the symbolic solution vector \vec{I} to satisfy restriction $C = C_1 \cup C_2 \cup C_3$.

The first sub-restriction is derived from the loop bounds and control-flow (ex. condition statements). That is, $C_1 = \{X_j \leq i_j \leq Y_j, (i_j - X_j) \bmod Z_j = 0, 1 \leq j \leq n\}$.

Before computation of expressions, in order to solve the equations as easily as possible, we must simplify symbolic expressions. The simplification can be handled based on a set of rules of simplification. After simplifying this restriction, the initial lower and upper bounds, $low(i)$ and $up(i)$, for each symbolic variable i can be deduced. Also, we use $eva(i)$ to represent the evaluated symbolic value under the restrictions. Thus we can obtain $low(i) \leq eva(i) \leq up(i)$. Furthermore, we define

$$eva(f(i_1, \dots, i_n)) = f(eva(i_1), \dots, eva(i_n)).$$

For the k -th block of the array A distributed onto processor p , we have

$$C_2 = \left\{ p = \left(f(\vec{I}) \operatorname{div} \beta_1 \right) \bmod P \right\}, \quad \text{or} \quad (1)$$

$$C_2 = \left\{ LB_k \leq f(\vec{I}) \leq UB_k, 0 \leq k \leq m_p \right\}, \quad (2)$$

and for the k -th block of the array B distributed onto processor q , we have

$$C_3 = \left\{ q = \left(g(\vec{I}) \operatorname{div} \beta_2 \right) \bmod P \right\}, \quad \text{or} \quad (3)$$

$$C_3 = \left\{ LB'_k \leq g(\vec{I}) \leq UB'_k, 0 \leq k \leq m_q \right\}, \quad (4)$$

where $LB_k = o_p + s_p * k$, $UB_k = o_p + b_p + s_p * k$, $LB'_k = o_q + s_q * k$, $UB'_k = o_q + b_q + s_q * k$.

Our evaluation method is, from the restriction C_2 , to limit the maximum lower and minimum upper bounds for a symbolic variable, to replace the old, wide bounds accordingly. The most intuitive method to replace with its bounds is to physically substitute each occurrence of the variable in the given expression with the variable's bounds, then simplify the resulting expression until the minimum range can be obtained. The replacement is progressively applying rewrite rules at each point where the variable is replaced by its bounds. All these rewrite rules are as a form of range of inequalities. If the derived new range is wider than the old one, the replacement does not occur. The similar way is applied to restriction C_3 .

In some cases, using replacement shown in the above can not determine the exact lower and upper bounds for an expression. Determining these bounds need to observe whether $f(i_1, \dots, i_n)$ is monotonic for i_k . Determining whether $f(i)$ is monotonically non-decreasing or monotonically non-increasing is not difficult. One can prove $f(x)$ is monotonically non-decreasing for x by proving that $f(x+1) - f(x) \geq 0$.

The range determination of a loop index i_k is more easily if the access function f and g are monotonically non-decreasing (non-increasing) for i_k , because we can replace $eva(i_k)$ simply with its known lower or upper bound in formula (2). That is, if assuming f is monotonically non-decreasing for i_k ,

$$\begin{aligned} LB_k &\leq eva(f(i_1, \dots, i_j, \dots, i_n)) \leq UB_k \Rightarrow \\ LB_k &\leq f(eva(i_1), \dots, eva(i_j), \dots, eva(i_n)) \leq f(eva(i_1), \dots, up(i_j), \dots, eva(i_n)) \wedge \\ &UB_k \geq eva(f(i_1, \dots, i_j, \dots, i_n)) \geq f(eva(i_1), \dots, low(i_j), \dots, eva(i_n)) \quad (5) \end{aligned}$$

5 Experiments

We evaluated our symbolic analysis algorithm on a 32-node distributed memory parallel computer CM-5, using MPI communication library and *gettimeofday()* system call to measure execution time. We select a subroutine OLDA from the code TRFD, appearing in Perfect benchmark ^[10]. A simplified version of this loop nest is shown in the left side of Figure 5. After using induction variable substitution to replace the induction variable $mrsij$ at statement S_1 the optimized version is shown in the right side of Figure 5. There is nonlinear array subscript for $xrsij$ at S_2 . To parallelize this loop nest, the communication set generation and address translation routine must be used.

The best distribution cases *cyclic*(2) for array $xrsij$, and *cyclic*(4) for xij are selected when $N = 16$ (with global array size 18632) and *cyclic*(3), and *cyclic*(7) are selected when $N = 20$ (with global array size 44310). Figure 3 and Figure 4 show the total loop execution time when $N = 16$ and $N = 20$ respectively. *Runtime algo* and *symbolic algo* respectively represent that we use runtime algorithm and symbolic algorithm in the communication set generation routine. We observed as the number of nodes increases, the execution time is not so much improvement because each processor has to communicate with increasing number of nodes. Although the communication set generation only involves the computation overhead, the performance can also be improved by using symbolic analysis algorithm.

6 Conclusions

Communication set generation influences the performance of parallel programs significantly. In this paper, we have proposed a symbolic analysis method to generate communication set for irregular array references. The technique overcomes the existed library's draw back which incurs significant overhead caused by the inspector and mapping non-local indices into local buffers. It completes the computation for generating communication at compile-time as much as possible. Thus, the total performance of the parallel programs including loop nest with nonlinear array references can be upgraded.

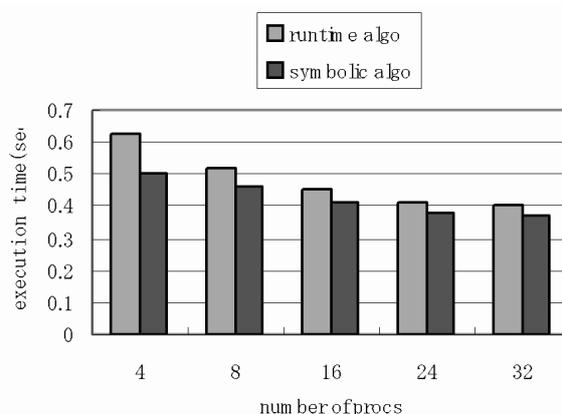


Figure 3: Results of TRFD loop nest OLDA when $N = 16$, $\beta_1 = 2$, $\beta_2 = 4$ on CM-5

References

- [1] C. Koelbel, D. Loveman, R. Schreeiber, G. Steele and M. Zosel. *The High Performance Fortran Handbook*, The MIT Press, 1994.
- [2] S. K. S. Gupta, S. S. Kaushik, C. H. Huang and P. Sadayappan. Compiling array expressions for efficient execution on distributed memory machines. *Journal of Parallel and Distributed Computing*, 32, 155–172, 1996.
- [3] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber and S. H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26, 72–84, 1995.
- [4] K. Kennedy, N. Nedeljkovic and A. Sethi. Efficient address generation for block-cyclic distributions. *Proceeding of ACM International Conference on Supercomputing*, 180–184, Barcelona, Spain, July 1995.
- [5] A. Thirumalai, J. Ramanujam and A. Venkatachar. Communication generation for data-parallel programs using integer lattices. *Languages and Compilers for Parallel Computing*, P. Sadayappan *et al.* (eds.), Lecture Notes in Computer Science, Springer-Verlag, 1996.
- [6] A. Lain, D. R. Chakrabarti and P. Banerjee. Compiler and run-time support for exploiting regularity within irregular applications. *IEEE Transactions on Parallel and Distributed Systems*, 11(2), February, 2000.
- [7] M. Guo, I. Nakata and Y. Yamashita. Contention-free communication scheduling for array redistribution. *Parallel Computing*, 26, 1325–1343, 2000.
- [8] M. Guo, Y. Yamashita and I. Nakata. Efficient implementation of multi-dimensional array redistribution. *IEICE Transactions on Information and Systems*, E81-D(11), Nov. 1998.

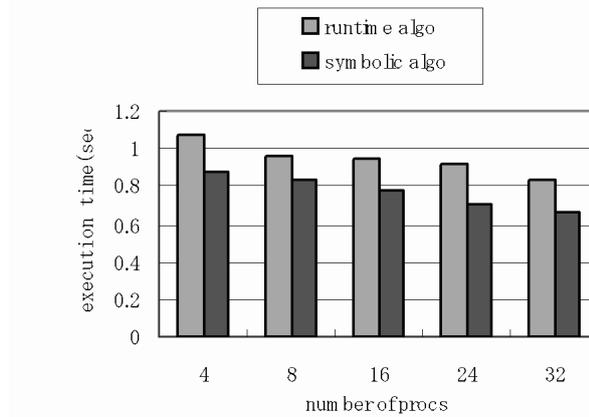


Figure 4: Results of TRFD loop nest OLDA when $N = 20$, $\beta_1 = 3$, $\beta_2 = 7$ on CM-5

<pre> mrsij0 = 0 DO mrs = 0, (N * N + N) / 2 - 1 mrsij = mrsij0 DO mi = 0, N - 1 DO mj = 0, mi - 1 S1: mrsij = mrsij + 1 S2: xrsij(mrsij) = xij(mj) ENDDO ENDDO mrsij0 = mrsij0 + (N * N + N) / 2 ENDDO </pre>		<pre> DO mrs = 0, (N * N + N) / 2 - 1 DO mi = 0, N - 1 DO mj = 0, mi - 1 S1: mrsij = (mi * mi + mi) & mrs * (N * N + N) / 2 + mj + 1 S2: xrsij(mrsji) = xij(mj) ENDDO ENDDO ENDDO </pre>
--	--	---

Figure 5: Sample loop nest with irregular array reference

- [9] R. Ponnusamy, J. Saltz and A. Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. *Proceeding of SuperComputing'93*, 361–370, Nov. 1993.
- [10] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3), 5–40, 1989.
- [11] Scott R. Kohn and Scott B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. *Proceeding of SHPCC*, 1994.
- [12] U. Kremer, J. Mellor-Crummey, K. Kennedy and A. Carlr. Automatic data lay-

- out for distribute-memory machines in the D programming environment. *Automatic Parallelization-New Approaches to Code Generation, Data Distribution, and Performance Prediction*, 136–147, C. W. Kessler, (ed.), Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993.
- [13] E. H. Y. Tseng and J.-L. Gaudlot. Communication generation for aligned and cyclic(k) distributions using integer lattice. *IEEE Transactions on Parallel and Distributed Systems*, 10(4), 136–146, 1999.
- [14] Y. Paek, A. Navarro, E. Zapata, J. Hoeflinger and D. Padua. An advanced compiler framework for non-cache-coherent multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 13(3), 241–259, 2002.
- [15] M. Guo, Z. Liu, C. Liu and L. Li. Reducing communication cost for parallelizing irregular scientific codes. *The 6th International Conference on Applied Parallel Computing*, 203–216, Lecture Notes in Computer Science 2367, Finland, June 2002.