Efficient Communications in Parallel Loop Distribution

Marc Le Fur and Yves Mahéo

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, FRANCE
Email: (mlefur|maheo)@irisa.fr

1 Introduction

In the framework of the compilation of HPF-like languages on distributed memory parallel computers, the distribution of regular parallel loops is studied extensively [9, 4, 1, 5, 6, 10]. Indeed, this kind of loops composes most computation-intensive parts of scientific applications and contains a great amount of potential parallelism. The techniques embedded in data-parallel compilers are often based on the *owner-computes rule*: each processor modifies only the variables assigned to it by the user-specified distribution. For regular parallel loops, the compiler usually produces a SPMD target code composed of a communication code, during which distant data are received from other processors, and a computation code.

In this paper, we focus on the generation and on the efficient execution of the communication code and address the problem regarding both its compile-time and run-time aspects. Indeed, communication optimizations (vectorization, aggregation, etc.) are often described at a high level in the literature through send or receive sets. However, in order to obtain performances, the gap between sets and communication buffers has to be filled in a non naive way. Run-time implementation strategies have a great importance therein.

Our approach applies to parallel loop nests with one statement, affine loop bounds and array subscripts. Regarding data distribution, it is assumed that each distributed array is partitioned into rectangular blocks of equal dimensions (known at compile-time) but any mapping for these blocks is supported. In HPF, this hypothesis encompasses for instance arrays distributed onto an abstract processor structure using the `DISTRIBUTE` directive. Thus each dimension of the array can be distributed with `CYCLIC(k)`, `CYCLIC`, `BLOCK` or `BLOCK(k)`.

The techniques we propose here have been implemented in the PANDORE environment [3], which is dedicated to the compilation, the execution and the observation of programs written in C-PANDORE or in a subset of HPF. One of the originalities of PANDORE lies in the separation between the compilation scheme and the management of distributed arrays. Both the compile-time and the run-time techniques benefit from this array management for communication optimization.

The paper is organized as follows. Section 2 recalls the distributed array management used in PANDORE. The basic principles of our communication code generation for parallel loops is explained in section 3. An enhancement of this basic method is then presented in section 4; it reduces the complexity of the description of communication sets by taking advantage of the layout of distributed arrays.

2 Distributed Array Management

In the PANDORE environment, distributed arrays are managed by a *software* paging system. The run-time uses the addressing scheme of standard paging systems but is not a virtual shared memory: the compiler always generates communication when distant data are needed, so we do not need to handle page faults.

The array management is based on the paging of arrays: the multi-dimensional index space of *each* array is linearized and then broken into pages. Pages are used to store local blocks and distant data received. If data have to be shared by two processors, each processor stores a copy of the page (or a part of the page) in its local memory. Array elements are accessed through a table of pages allocated on each processor.

2.1 Principle

To access an element referred to by an index vector $(i_0, \ldots, i_{n-1})$ in the source program, a page number and an offset (PG and OF) are computed from the index vector with the linearization function $\mathcal{L}$ and the page size S: $PG = \mathcal{L}(i_0, \ldots, i_{n-1})\ div\ S$, $OF = \mathcal{L}(i_0, \ldots, i_{n-1})\ mod\ S$. For a given distributed array, the page size S and the linearization function $\mathcal{L}$ are computed by the compiler so that the evaluation of PG and OF is efficient. Time consuming operations are avoided by using powers of two, turning integer division, modulo and multiplication into simple logical operations (shifts and masks).

For this, the compiler first choose the dimension δ in which the size of the blocks is the largest. Function $\mathcal{L}$ is the C linearization function applied to a permutation of the access vector that puts index number δ in last position. The page size S is then defined by the following (s_δ is the block size in dimension δ): if s_δ is a power of two or dimension δ is not distributed, S is the smaller power of two greater than s_δ; otherwise S is the largest power of two less than s_δ. Moreover simplifications in the expression of PG and OF are performed when the is a non-distributed dimension. Figure 1 illustrates this paging in the 2D case.

Actually, an optimized computation of (PG, OF) is achieved by avoiding the explicit computation of the linear address $\mathcal{L}(i_0, \ldots, i_{n-1})$: we express PG and OF directly as a function of the index vector, thus, when dimension δ is not distributed, *mod* and *div* operations are removed. A more detailed description of this array management can be found in [8].

2.2 Benefits

With this software paging, access times remain very close to those without index conversion. The memory overhead induced does not exceed a few percents for most distributions; it is almost entirely due to the tables of pages: when a page contains elements that have no equivalent in the original sequential space, or when just a part of a distant page is accessed in a loop, only a portion of the page is actually allocated.

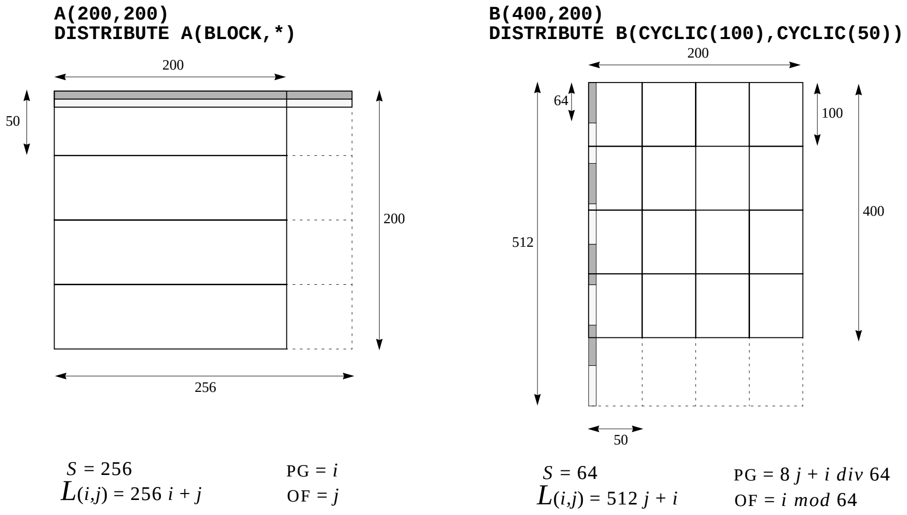


Figure 1: Paging of 2D arrays

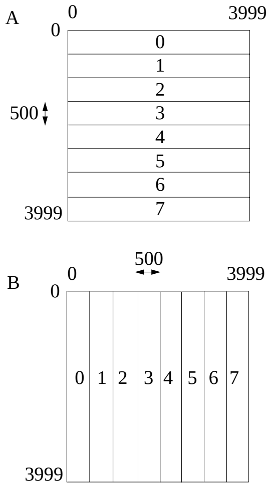
Moreover, paging distributed arrays offers several worthwhile characteristics. First, the scheme is always applicable as it is independent of the analysis of the code: it only depends on distribution parameters. The scheme is uniform: as far as accesses are concerned, no difference is made between local elements and distant elements previously received. Finally, the memory contiguity is preserved in the direction of the pages: contiguous elements of the original array are still contiguous in the local representation. This facilitates the exploitation of caches and vector processors and helps to optimize communications as it will be shown later.

3 Basic Communication Code for Parallel Loops

Let us briefly explain the principle of our communication code generation through the example given in figure 2. Loop bounds and array subscripts but also the distribution of arrays A and B are analyzed by the compiler. The generated code comprises two parts: a communication part —in charge of pre-fetching non-local data from other processors— followed by a computation part. The communication code is itself divided into a send code and a dual receive code. The basis of each code consists in the scanning of a polyhedron [7] that characterizes the set of data associated with $B[j, i+j-2]$ that must be exchanged between processors.

In the analysis of array distributions, only the partitioning into blocks is considered by the compiler. In the example, array A is divided into 8 blocks of size 500×4000 whereas array B is decomposed into 8 blocks of size 4000×500. The mapping of the blocks (`CYCLIC` in the example) is taken into account at run-time through guards depending on the processor identity. The send code generated by the compiler is given in figure 3.

In this code, the (i, j)-loop describes the set $Block\_send\_set(A, k_A, B, k_B)$: the set of elements of block number k_B of B that must be sent to the owner of block number k_A of



```
      REAL, DIMENSION(0:3999,0:3999) :: A,B
!HPF$ PROCESSORS PROCS(P)
!HPF$ DISTRIBUTE (CYCLIC(500),*) ONTO PROCS :: A
!HPF$ DISTRIBUTE (*, CYCLIC(500)) ONTO PROCS :: B

      DO I=1, 1000
         DO J=I, 2*I+1
            A(I,J-I) = B(J,I+J-2)
         END DO
      END DO
```

Figure 2: HPF Source code

```
for $k_A = 0$ , 2
    if $myself \neq$ owner of block $k_A$ of $A$ then
        for $k_B = max(0, 2*k_A-1)$ , $min(5, 3*k_A+2)$
            if $myself =$ owner of block $k_B$ of $B$ then
                for $i = max(500*k_A, div(500*k_B+3, 3))$ ,
                        $min(500*k_A+499, 1000, 250*k_B+250)$
                    for $j = max(i, 500*k_B-i+2)$ , $min(2*i+1, 500*k_B-i+501)$
                        elt_send($B[j, i+j-2]$, owner of block $k_A$ of $A$)
```

Figure 3: Basic send code

A. So $Block\_send\_set(A, k_A, B, k_B)$ is a subset of $Send\_set(B, p, p')$: the set of data of B that must be sent from p (the owner of block k_B of B) to p' (the owner of block k_A of A).

A straightforward implementation of the run-time primitive **elt_send** consists in a simple send of the element $B[j, i+j-2]$. This is not a satisfactory solution because it leads to a great number of small messages, and so to a prohibitive latency cost.

One can think optimizing this implementation by aggregating all the elements to be sent from a processor to another. In this case, the primitive **elt_send** adds a couple (address, value) to a buffer that can be sent at the end of the (i, j)-loop. This reduces the number of messages but several drawbacks remain. First, the number of data transferred is not optimal since an address has to be attached to each element. Second, it necessitates memory copies between local representations of arrays and communication buffers (packing/unpacking). The fact that intermediate communication buffers are allocated constitutes also a memory overhead.

Moreover, in these two solutions, the time spent in the description of the send set is high since the scanning is performed element-wise.

To summarize, an efficient communication code needs to reduce memory overhead (communication buffers, etc.) and the time passed in:

- the description of the communication sets;
- the packing/unpacking of the communication buffers;
- the effective data transfer (*i.e.* the number of messages and their size must be minimized).

Minimizing all these parameters may be contradictory: for instance, a coarse description of send sets (*e.g.* surrounding rectangular sections) can be rapidly performed but is likely to bring about transfers of useless elements. Therefore, it is clear that a compromise must be reached that allows for the layout of arrays and that involves both the compiler and the run-time system. Next section presents such a compromise that is implemented in the PANDORE environment.

4 Efficient Communication Code for Parallel Loops

A solution for optimizing communications consists in sending supersets of send sets, while exploiting the memory contiguity in the layout of distributed arrays. The choice adopted in PANDORE consists in transferring the convex-hull of each send set associated with a pair of blocks *i.e.*, for the example of figure 2, the convex-hull of each $Block\_send\_set(A, k_A, B, k_B)$. Thus, in the general case, $Send\_set(B, p, p')$ is a non necessarily disjoint union of convex-hulls.

4.1 Enhancing the Compilation Technique

Allowing for the memory contiguity (*i.e.* the direction of pages) and representing send sets by their convex-hulls find their expression in the definition of new polyhedrons at compile-time. See [3] for more details about the static analysis performed to construct these polyhedrons.

The communication code obtained for the example given above is shown in figure 4. It comprises three parts:

- The first one computes, on a given processor p, the set of processors p' that must receive data from p and for each p', a description of $Send\_set(B, p, p')$.
- In the second part, each processor p determines the set of processors that will send distant data to p. There is no computation of any receive set here since the description of the send sets will be included in the messages received by p.
- The third part is only a call to a run-time routine that is in charge of the interprocessor communications according to the different sets computed in the previous parts.

As it can be seen on the code, the scanning of the convex-hull of a $Block\_send\_set(A, k_A, B, k_B)$ is performed efficiently —no longer element-wise— by enumerating its extremal points in a given direction. The (i, j)-loop of figure 3 has been replaced by a single loop scanning the columns of array B, since the pages of B are column-wise oriented. For a

```
——— Part 1 ———
for $k_A = 0,\ 2$
  $pA$ := owner_block($A, k_A$)
  if myself $\neq pA$ then
    for $k_B = max(0, 2*k_A - 1)\ ,\ min(5, 3*k_A + 2)$
      if myself = owner_block($B, k_B$) then
        add_recver($B, pA$)
        for $v = max(500*k_B, 1000*k_A - 2)\ ,\ min(500*k_B + 499, 1500*k_A + 1496)$
          $u\_inf$ := $max(div(v+3, 2), v - 998, -500*k_A + v - 497)$
          $u\_sup$ := $min(div(2*v+5, 3), -500*k_A + v + 2)$
          portion_pack($B, v, u\_inf, u\_sup, pA$)

——— Part 2 ———
for $k_A = 0,\ 2$
  if myself = owner_block($A, k_A$) then
    for $k_B = max(0, 2*k_A - 1)\ ,\ min(5, 3*k_A + 2)$
      $pB$ := owner_block($B, k_B$)
      if myself $\neq pB$ then
        add_sender($Y, pB$)

——— Part 3 ———
exchange($B$)
```

Figure 4: Optimized communication code

given column v, the routine portion_pack adds a portion of column v ($B[u\_inf..u\_sup, v]$) to the current $Send\_set(B,$ myself$, pA)$.

Moreover, in this example, the convex-hull is exact since the linear part of the access function of reference $B[j, i+j-2]$ is unimodular, which is generally the case in regular scientific applications.

4.2 Enhancing the Effective Data Transfers

The notion of segments is used to reduce the amount of memory needed for the storage of the description of send sets. It also permits data transfers to be and optimized. A segment is a contiguous set of elements within a page. It is represented by a triplet (*pg*, *ofs*, *ofe*) where *pg* is the page number and *ofs* (resp. *ofe*) is the offset of the beginning (resp. the end) of the segment.

In figure 4, the call portion_pack(B, v, u_inf, u_sup, pA) adds the segments intersected by the portion of column $B[u\_inf..u\_sup, v]$ to the list of segments to be sent to pA. Only one segment per page is memorized in this list; this segment is defined by the convex union of the segments within the page. This mechanism ensures that an array element is recorded only once and prevents from redundant transfers, notably in the case of multiple right hand side reference to the same array.

The routine exchange(B) performs the sends and the receives of all the segments related to B. Segments are communicated differently according to their size. Small segments are aggregated in a unique message whereas direct communication is used for big segments. In the latter case, the segment is transferred in a single message directly from the page on the sender side to the page on the receiver side, without any packing/-unpacking. The threshold between small and big segments is determined from platform-specific parameters such as the message latency and the memory copy bandwidth. In the current implementation of `exchange`, all the sends are performed prior to all the receives. A more asynchronous solution may be envisaged in which communication overlaps the construction of buffers for aggregation.

This approach leads to a good compromise between the number of messages, the total amount of data transferred, the memory overhead and the time required for packing/-unpacking buffers.

5 Conclusion

In this paper, we have presented original techniques for the generation and the efficient execution of communication code for parallel loop nests. The problem has been studied through its two components: on one hand, the generation of a fast description of communication sets by the compiler and, on the other hand, the implementation of efficient transfers at run-time. Both take into account the characteristics of the distributed array management, notably the memory contiguity.

These optimizations have been integrated in the Pandore environment. They lead to good performances for a number of numerical applications [2]. Figure 5 shows the speedups obtained on the Jacobi kernel on the Intel iPSC/2 and on a network of workstations.

Although the approach we have presented here applies to Hpf direct distributions, we are currently investigating the adaptation of our array management and of our static analysis in order to allow for alignment.

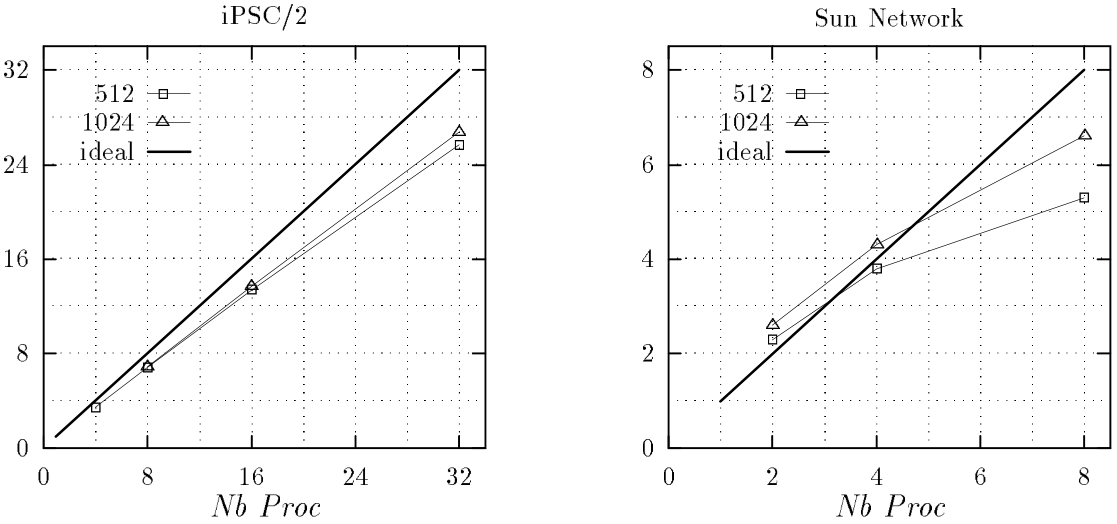


Figure 5: Speedups obtained on the Jacobi kernel

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. – A Linear Algebra Framework for Static HPF Code Distribution. – In *Fourth International Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[2] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. – Parallelization of a Wave Propagation Application using a Data Parallel Compiler. – In *9th International Parallel Processing Symposium*, Santa Barbara, California, April 1995.

[3] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. – The Pandore Data-Parallel Compiler and its Portable Runtime. – In *High-Performance Computing and Networking*, Milan, Italy, May 1995. LNCS 919, Springer Verlag.

[4] B.M. Chapman and H.P. Zima. – Compiling for Distributed-Memory Systems. – Research Report ACPC/TR 92-17, Austrian Center for Parallel Computation, University of Vienna, November 1992.

[5] 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. – In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

[6] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. – Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines. – Technical Report 19, The Ohio State University, 1994.

[7] M. Le Fur. – Scanning Parameterized Polyhedron using Fourier-Motzkin Elimination. – In *High Performance Computing Symposium*, Montréal, Canada, July 1995.

[8] Y. Mahéo and J.-L. Pazat. – Distributed Array Management for HPF Compilers. – In *High Performance Computing Symposium*, Montréal, Canada, July 1995.

[9] C.-W. Tseng. – *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* – PhD thesis, Rice University, Houston, Texas, January 1993.

[10] V. Van Dongen. – Compiling Distributed Loops onto SPMD Code. – *Parallel Processing Letter*, 4(3), 1994.