Run-time Data Distribution for Block-Structured Applications on Distributed Memory Computers

Stephen J. Fink & Scott B. Baden

CSE Technical Report Number CS94-386

September 1994

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093-0114

To appear in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing

Run-time Data Distribution for Block-Structured Applications on Distributed Memory Computers*

Stephen J. Fink† Scott B. Baden‡

Abstract

In many scientific applications running on parallel computers, efficient data decompositions exhibit a block structure which can be regular or irregular. We present a run-time strategy for both regular and irregular block-structured applications which provides a three-stage mapping including alignment, first-class decomposition objects, and block data movement. We have implemented this strategy using a small set of primitive operations, and present these along with performance results. In contrast to data parallel Fortran dialects, our system executes completely at run-time and supports coarse-grained user-defined data decompositions.

1 Introduction

It is well-known that on distributed memory parallel computers, the distribution of data across processors can have a substantial impact on a program's performance. Under the message-passing programming model, the programmer must manage all aspects of data distribution by hand. Programming languages and run-time libraries that manage the low-level details of data distribution can alleviate the programmer's burden [10, 9, 8, 12, 1, 4, 11].

Perhaps the simplest strategy for managing distributed data is the mapping array, a many-to-one mapping of data elements to processors[9]. Mapping arrays provide the most flexibility of any data distribution scheme, but incur a large space overhead since one mapping element must be stored for each data element. Many scientific applications do not require the full generality of mapping arrays because the distribution of data across processors has a block structure. The block structure may be *regular*, in that the size of blocks is described by a simple rule, or *irregular*, consisting of blocks of various shapes and sizes. For block-structured applications, we may exploit the decomposition structure to realize a more efficient data management strategy than mapping arrays.

Data decompositions that are not known at compile-time or that may change during execution demand run-time support. We present a small set of efficient geometric run-time primitives to manage dynamic regular and irregular block-structured data distributions. With these primitives, we show how to provide

- A generalization of Fortran D's[9] three-stage distribution scheme for both regular and irregular block distributions,
- first-class dynamic distribution objects, and
- efficient support for block data movement.

*This work was supported by ONR contract N00014-93-1-0152. Computer time on the Intel Paragon at the San Diego Supercomputer Center was provided by a UCSD Division of Engineering Block Grant.

†Graduate student, University of California, San Diego, sfink@cs.ucsd.edu

‡Assistant professor, University of California, San Diego, baden@cs.ucsd.edu

2 Previous work

HPF[10] manages block-structured distributions with the 3-stage distribution scheme pioneered in Fortran D[9].

First, the programmer defines the shape of a *logical processor array*, which the implementation maps to the physical processor set. Then, a rectangular abstract index space called a *template* is distributed across the logical processor array. Finally, a distributed array is *aligned* to a template, setting up a correspondence between array indices and template indices. An array element is stored on the same processor that owns the corresponding template element. Using templates and alignments, the programmer can control the distribution of arrays relative to one another, and thus layout arrays across processors in order to minimize communication.

The BLOCK distribution directive produces a regular block distribution across the logical processor array. If there are N data elements along a dimension of the template, and P processors along the corresponding dimension of the logical processor array, then the BLOCK directive indicates that each logical processor is assigned a contiguous block of $\lceil \frac{N}{P} \rceil$ indices in that dimension.

In HPF, templates are compile-time directives only and not first-class objects. This limits the programmer since templates cannot be defined at run-time nor passed across procedure boundaries [7]. To avoid these limitations, Vienna Fortran omits templates but includes a more general form of the block distribution directive[6]. The Multiblock PARTI library [1] and pC++ [4] implement distribution schemes similar to the HPF 3-stage model, but implement distribution and alignment with first-class, dynamic objects.

The BLOCK distribution directive does not describe the irregular block distributions that arise in applications such as adaptive mesh refinement[3] and partitions produced by load-balancing with irregular decompositions such as recursive bisection[2]. The LPARX library provides mechanisms for managing these irregular block distributions [11]. However, LPARX does not provide automatic regular BLOCK decompositions, nor a three-stage mapping strategy for aligning distributed arrays relative to each other.

3 Run-time primitives for data distribution

Geometric primitives were introduced as first-class objects in FIDIL [13] as a means for managing index sets of arrays, and are employed by the ZPL language as compile-time iteration masks[12]. The LPARX [11] class library implements geometric domain abstractions as first-class run-time objects. We shall use the following subset of LPARX primitives to implement dynamic data distribution.

3.1 LPARX primitives

A *Point* is an n-tuple representing a point in Z^n. Addition and multiplication are defined over Points as addition and multiplication of corresponding elements.

Two points in Z^n define a rectangular subset of Z^n. A *Region* is a first-class object representing this subset, and will be denoted here as $[P_1, P_2]$, where P_1 and P_2 are Points. For a Region R, $lwb(R)$ returns P_1 and $upb(R)$ returns P_2. For a Region R and a Point P, $shift(R, P)$ returns the Region translated by the vector P; i.e., $[lwb(R) + P, upb(R) + P]$. Intersection is defined over Regions in the usual geometric way. Note that this intersection can be computed in essentially constant time[1].

[1]Intersection is computed in time that is linear in the number of spatial dimensions. However, since the

A *Grid* is a set of objects all of the same type whose index space is a *Region*. A Grid G may not be distributed across multiple processor memories; rather, it exists only in the address space of one processor, $owner(G)$. For a Grid G, $region(G)$ returns the Region that is the index space of G.

An *XArray* is an array of Grids, each of which may be owned by an arbitrary processor. An m-element XArray incurs an $O(m)$ space overhead on each processor since the index domain for each Grid is replicated. However, by replicating the distribution information we can avoid communication during index domain calculations. Typically, $m = P$ and since an n-dimensional Region is represented by $2n$ integers, the space overhead is small in practice.

3.2 Additional primitives

We augment the LPARX primitives with the *transpose* operation and the *sRegion*.

Let Q be a Point whose coordinates are a permutation of the integers $\{1, \ldots, n\}$. Then for a Point P, $transpose(P, Q)$ returns the Point with the coordinates of P in the order given by the permutation Q. For a permutation vector Q, $t^{-1}(Q)$ is the permutation vector such that for any point P, $transpose(transpose(P, Q), t^{-1}(Q)) = P$. The *transpose* operation is defined over Regions so that for a Region R, $transpose(R, Q)$ returns $[transpose(lwb(R), Q), transpose(upb(R), Q)]$.

The *sRegion* is a Region with a stride in each dimension. This object is similar to the Fortran 90 array section specifier and the regular section employed in the Multiblock Parti run-time library [1].

The stride values for an n-dimensional sRegion can be represented by an n-dimensional Point. The *expand* and *collapse* functions relate Regions and sRegions. Given a Region R and a point S, $expand(R, S)$ returns the sRegion whose Region is $[S * lwb(R), S * upb(R)]$ and whose stride is S. *Collapse* is the inverse of expand, so $collapse(expand(R, S)) = R$. We shall use intersection between a Region and an sRegion, defined in the obvious way. Note that this intersection can be computed in essentially constant time [1].

3.3 Distribution and Alignment

With these primitives and basic data structures, we can implement logical processor arrays, templates, and alignment with first-class dynamic distribution objects.

A logical processor array can be implemented as an array of integers. The shape of the array is the shape of the logical processor array, and the integers are the physical processor assignments for the logical processors.

A first-class template is just a Region. When a template is distributed across a logical processor array according to a BLOCK or irregular block distribution, each processor's partition is a Region. Thus a distributed template is implemented by an array of Regions, each with a processor assignment $owner(R)$. A standard routine generates the array of Regions for a regular block distribution, or a user-written routine can generate the array of Regions to represent an irregular block distribution. In either case, the representation is the same and can be used in the alignment stage.

Let A be an array that is to be distributed, and let $region(A)$ be its index set. Let T be a distributed template; i.e., an array of Regions with processor assignments. We do not assume that T has a regular BLOCK distribution. We now give an algorithm to instantiate A as an XArray of Grids according to the directive "ALIGN A($i_1, i_2, \ldots, i_n$)

number of spatial dimensions is effectively bounded by 3 in most scientific computations, we can in practice assume a constant time.

with T($d_{j_1} i_{j_1} + c j_1, d_{j_2} i_{j_2} + c_{j_2}, \ldots, d_{j_n} i_{j_n} + c_{j_n}$)," where $\{j_1, \ldots, j_n\}$ is a permutation of $\{1, \ldots, n\}$.

Let P_c be the Point $(c_1, c_2, \ldots, c_n)$, let P_d be the Point $(d_1, d_2, \ldots, d_n)$, and let P_j be the Point $(j_1, j_2, \ldots, j_n)$. It is easy to verify that steps 1-3 of algorithm 3.1 produce the sRegion R_s that is the image of $region(A)$ in the template index space according to the alignment directive above. The intersection in step 4 of this image with the index set of R gives the indices to instantiate on $owner(R)$. Finally, steps 5-7 map these indices back to the index space of A, giving the index space of the Grid instantiated on $owner(R)$.

ALGORITHM 3.1.
(1) $R_s \leftarrow shift(region(A), P_c)$
(2) $R_s \leftarrow expand(R_s, P_d)$
(3) $R_s \leftarrow transpose(R_s, P_j)$
For each $R \in T$
(4) $R_d \leftarrow R \cap R_s$
(5) $R_d \leftarrow transpose(R_d, t^{-1}(P_j))$
(6) $R_d \leftarrow collapse(R_d)$
(7) $R_d \leftarrow shift(R_d, -1 \cdot P_c)$
(8) *On* $owner(R)$ *instantiate a Grid with index domain* R_d

Note that the alignment algorithm is dimension-independent does not assume any structure on the block decomposition of T. Thus, it can be used for both regular and irregular block distributions.

4 Distributed Block Copy

To achieve good performance, it is imperative that data movement be performed efficiently once data is distributed across processors. For some common communication patterns, the primitives described above can be used to determine communication schedules dynamically at runtime. As an example, we show how to implement block copy between distributed arrays.

We build on top of the LPARX *Copy* primitive that performs block copies between individual Grids. Let A and B be Grids, and let S and D be Regions; Copy moves values from Grid B over Region S into Grid A over Region D. Since A and B may reside on different processors, Copy may cause communication between the Grids. There are two forms of Copy, *get* and *put*, which differ in the form of communication generated[2].

- *get* (denoted A on $D \Leftarrow B$ on S): the processor which owns Grid A determines D and S and sends a request message to the processor which owns Grid B. The reply from Grid B contains the requested data.

- *put* (denoted B on $S \Rightarrow A$ on D): the processor which owns Grid B determines D and S and sends the data directly to the processor that owns Grid A.

Note that a get incurs two messages but a put incurs only one.

Now let X and Y be distributed arrays. We shall use the primitives above to implement block copy between distributed arrays; i.e., to copy values from some Region S of Y into some Region D of X. In this distributed case, S and D may span multiple processor memories.

[2]The get and put forms of Copy are coarse-grain operations, similar to the `bulk_get` and `bulk_put` library extensions to Split-C[8].

Assuming no regular block structure in the distributions of X and Y, distributed block copy can be implemented as shown in algorithm 4.1. Since the domains of each Grid in X and Y are replicated on each processor, the Region operations incur no communication.

ALGORITHM 4.1.

Forall Grid $G \in Y$, *do on* $owner(G)$:
  For each Grid $G' \in X$
    $R_d \leftarrow region(G') \cap D$
    $R_s \leftarrow shift(R_d, lwb(S) - lwb(D))$
    $R_s \leftarrow R_s \cap region(G)$
    G *on* $R_s \Rightarrow G'$ *on* R_d

ALGORITHM 4.2.

Forall Grid $G \in Y$, *do on* $owner(G)$:
  $S' \leftarrow S \cap region(G)$
  $D' \leftarrow shift(S', lwb(D) - lwb(S))$
  $X_{low} \leftarrow XOwner(lwb(D'))$
  $X_{high} \leftarrow XOwner(upb(D'))$
  For each Point $P \in [X_{low}, X_{high}]$
    $R_d \leftarrow region(X(P)) \cap D'$
    $R_s \leftarrow shift(R_d, lwb(S) - lwb(D))$
    G *on* $R_s \Rightarrow X(P) on R_d$

Let A be an XArray of Grids implementing a distributed array whose global index space is a Region L. Then A is itself an array (whose elements are Grids) whose index set is a Region $region(A)$. Let P be some Point, $P \in L$. Since A is distributed, $P \in region(G)$ for exactly one Grid $G \in A$. We can identify G by its index in A; i.e.,, by a Point Q in $region(A)$ such that $G = A(Q)$. We define an operation $XOwner(P, A)$ which returns the Point $Q \in region(A)$ such that $P \in region(A(Q))$.

If A has a *regular* block decomposition, it is simple to determine $XOwner(P, A)$ in constant time as described in [5]. Furthermore, let R be a Region, and consider the Region $[XOwner(lwb(R), A), XOwner(upb(R), A)]$. This Region identifies the set of Grids in A that own some point $p \in R$.

Returning to algorithm 4.1, suppose X has a regular block decomposition. Using the previous argument, on each processor $owner(G)$ we can efficiently identify the Grids $G' \in X$ which require data to be sent from G. Thus, we can minimize the number of intersections performed, resulting in a more efficient algorithm, 4.2.

Finally, suppose X does not have a regular block distribution, but Y does. Using similar reasoning, we can reduce the number of intersections by only considering relevant Grids in Y. However, since the processors that own Grids $G' \in X$ perform the computation, this strategy requires using the *get* form of Copy rather than the *put*. See algorithm 4.3.

ALGORITHM 4.3.

For all Grid $G' \in X$, *do on* $owner(G')$
  $D' \leftarrow region(G') \cap D$
  $S' \leftarrow shift(D', lwb(S) - lwb(D))$
  $Y_{low} \leftarrow XOwner(lwb(S'))$
  $Y_{high} \leftarrow XOwner(upb(S'))$
  For each Point $P \in [Y_{low}, Y_{high}]$
    $R_s \leftarrow S' \cap region(Y(P))$
    $R_d \leftarrow shift(S', lwb(D) - lwb(S))$
    G' *on* $R_d \Leftarrow Y(P)$ *on* R_s

Algorithm	time(ms)
4.1	16.8
4.2	11.6
4.3	11.9

TABLE 1

Distributed block copy times. Block size is 100 x 100 32-bit integers. Both the source and destination regions are distributed across 4 processors, while the entire 500 x 500 arrays are distributed across 64 processors.

The distribution and block copy algorithms presented here have been implemented as a dimension-independent C++ library with calls to LPARX. Table 1 gives performance results for distributed block copy using each of the three algorithms. Results were obtained using a regular (BLOCK,BLOCK) distribution on 64 processors of an Intel Paragon using the NX message-passing layer under OSF/1 R1.2, compiling with gcc v.2.5.7 and linking with LPARX v.1.0. The results show that although algorithm 4.1 is the most general and can handle irregular block distributions, better performance can be realized by algorithms 4.2 and 4.3 if one or both of the arrays is known to have a regular block distribution.

Note that these results were obtained from a prototype implementation, and we expect that tuning of the underlying software will result in substantial performance improvements for all three algorithms.

5 Conclusions

We have presented a small set of geometric run-time primitives and algorithms to manage distributed block-structured data. Unlike existing parallel languages and run-time libraries, our system provides a Fortran D-style three-stage mapping with first-class dynamic distribution objects for both regular and irregular block decompositions. All decisions are made at run-time, providing support for distributions that are unknown at compile-time or may change dynamically during execution.

The authors would like to thank Scott Kohn for many valuable discussions and suggestions for this paper.

References

[1] G. Agrawal, A. Sussman, and J. Saltz, *Efficient runtime support for parallelizeing block structured applications*, in Proceedings of the 1994 Scalable High Performance Computing Conference, Knoxville, Tenessee, May 1994.

[2] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers, C-36 (1987), pp. 570–580.

[3] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, Journal of Comp. Physics, 82 (1989), pp. 64–84.

[4] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B.Mohr, *Implementing a parallel C++ runtime system for scalable parallel systems*, in Proc. Supercomputing, 1993, pp. 588–597.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, *Compiling distribution directives in a fortran 90d compiler*, Tech. Rep. SCCS-388, Northeast Parallel Architectures Center, Syracuse, NY, July 1992.

[6] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima, *Dynamic data distributions in vienna fortran*, Report 93-92, ICASE, Decemeger 1993.

[7] B. Chapman, P. Mehrotra, and H. Zima, *High performance fortran without templates: an alternative model for distribution and alignment*, in Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, July 1993.

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, *Parallel programming in split-c*. Draft, May 1993.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, *Fortran D language specification*, Tech. Rep. TR90-141, Dept. of Computer Science, Rice University, Houston, TX, December 1989.

[10] High Performance Fortran Forum, *High performance fortran language specification*, Rice Univeristy, Houston, Texas, May 1993.

[11] S. R. Kohn and S. B. Baden, *A robust parallel programming model for dynamic, non-uniform scientific computation*, in Proceedings of the 1994 Scalable High Performance Computing Conference, Knoxville, Tenessee, May 1994.

[12] C. Lin and L. Snyder, *ZPL: An array sublanguage*, in Languages and Compilers for Parallel Computation, 6th International Workshop Proceedings, Springer-Verlag, 1994, pp. 96–114.

[13] L. Semenzato and P. Hilfinger, *Arrays in fidil*, in Proceedings of the First International Workshop on Arrays, Functional Programming, and Parallel Systems, July 1990.