A Communication Backend for Parallel Language Compilers

James M. Stichnoth and Thomas Gross

Carnegie Mellon University

Abstract. Generating good communication code is an important issue for all compilers targeting parallel or distributed systems. However, different compilers for the same parallel system usually implement the communication generation routines (e.g., message buffer packing) independently and from scratch. As a result, these compilers either pursue a simple approach (calling a standard runtime library), which does not do justice to the capabilities of the system, or they incur high development costs. This paper describes a way to separate the communication issues from other compilation aspects (e.g., determining the distribution of data and computation). This organization places the responsibility for communication issues with the *communication backend*, and this backend can be shared by different compilers. It produces code that is customized for each communication step, based on the exact data distribution and the characteristics of the target parallel system. This approach has several advantages: (1) The communication backend can be shared by multiple compilers, e.g., for different parallel languages. (2) The communication backend provides a way to integrate regular and irregular communication, e.g., as required to deal with irregular codes. (3) Retargeting of a parallel compiler is simplified, since the communication backend deals with the interface to communication (and the single-node compiler). (4) The communication backend can optimize the code, e.g., by constant folding and constant propagation. Code produced by the communication backend is always at least as fast as library code, but the customization has the potential to significantly improve performance depending on what information is known at compile time.

1 Introduction

The standard model for compiling a program written in a parallel language like HPF for a specific parallel system involves two distinct phases. The first phase deals with all parallel aspects of mapping the application onto a parallel system. This phase starts with a program without explicit communication, and produces a program with explicit communication operations. Depending on the implementation strategy of the compiler, this phase may consist of multiple sub-phases, rely on user directives or hints, or use trace information. The second phase of the compiler deals with all single-node aspects of a program, including producing object code.

When mapping an application onto a parallel or distributed system, there exists a wide choice of mappings, and one of the research topics in compiling for parallel systems is how to find good data and computation mappings. Ideally, a compiler finds a mapping

This research was sponsored by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152 and by an Intel Graduate Fellowship.

so that there is no communication, but in practice, realistic applications include some amount of communication. These communication operations determine the scalability of a program (i.e., its efficiency when using more processors).

In this paper, we discuss an approach to deal with whatever communication a compiler decides to perform. That is, we structure the first phase of the parallelizing compiler so that generating communication operations is done by the *communication backend*. The communication backend starts with data distributions that are decided upon elsewhere. It generates code for whatever communication is required to adhere to the chosen distributions. Deciding on the distribution for an object, mapping computations onto the various processors of the system, and making tradeoffs between different distributions are all issues that are dealt with by other parts of the compilation tool chain.

The motivation for the development of this communication backend is provided by two tools developed at Carnegie Mellon. The Fx compiler [3, 11] translates a dialect of High Performance Fortran (HPF) for a variety of parallel systems and communication libraries (iWarp, PVM, Paragon/OSF, Paragon/SUNMOS, T3D). Archimedes [7] is a domain-specific compiler to support the solution of PDEs on parallel systems; its first major users are our collaborators on a Grand Challenge application in the Department of Civil Engineering at CMU. Archimedes targets iWarp and the Paragon, and a retarget to other platforms is under development.

During the development of these compilers, and during our efforts to retarget them as new parallel systems became available, we observed that producing good communication code is far from easy, even if the distributions of the data are known at compile time. Retargeting a compiler can be accomplished easily if performance is no objective, but this approach is not acceptable to our sophisticated users (who tend to point out any inefficiencies). Furthermore, we noticed that the requirements for Fx (dealing with regular computations) and Archimedes (dealing with irregular computations) are not vastly different; we discuss this point later in this paper. Although the communication requirements of these compilers differ in many details, there exists a common structure, provided that we can separate the issue of deciding on the mappings from the issue of generating communication operations. Although this decision to use a separate communication backend required revising part of the Fx compiler (which had been retargeted already to a few systems), this new structure results in a significant long-term savings. At this time, a first prototype of the communication backend is operational.

2 Problem Statement

We use the term *parallel language compiler* (or *language compiler* for short) to refer to the compiler that decides on the distribution of data and computation. Then, the communication backend processes the output of this compiler. For example, the parallel language compiler system may select different data distributions for an array A. When it is necessary to switch from one distribution to another, a specific *communication step* is required to move the data from the old distribution to conform with the new distribution. Usually, it is impossible to move all the data of A at once. Instead, a number of *communication operations* are needed. Each communication operation moves a block of A; a block may be as small as a single item. There may exist many choices to implement the communication operations, depending on the properties of target machine, like the memory system architecture [10], the interconnect topology, or the support for collective operations.

A number of applications can be expressed in a language like HPF that is based on regular array statements. A regular array statement is one like $A = B + C$ where all arrays have regular (i.e., block-cyclic) distributions. HPF allows only regular distributions (i.e., block, cyclic, and block-cyclic), but although there exists a compact (regular) representation of the communication required for such a statement, the communication operations can be quite complicated (see [8] for a discussion of this topic). A straightforward approach yields poor communication code, which directly impacts performance. To optimize the communication code, it may be possible to exploit that the number of processors is known at compile time, or that the dimensions of an array are known at compile time, etc.; in each case, it is possible to customize the communication operations.

Regular array statements are not sufficient to describe all applications of interest. Two important examples of irregularity are use of an indexing vector (e.g., $A[IA[1 : n]] = B[1 : n]$), or array operands with irregular distributions (e.g., a *mapped* distribution). Irregular distributions are not part of HPF, but are necessary for tools like Archimedes. Many other problems involve computations with indexed accesses to arrays. The communication required to deal with such problems is *irregular*, in the sense that there is no concise, regular description of how data has to be moved. As discussed later, executing such an irregular array statement requires multiple communication operations.

In summary, then, the objectives for this communication backend are (1) unify dealing with regular and irregular communication; (2) allow retargeting to different parallel or distributed systems, and allow use by multiple compilers; and (3) produce efficient communication operations.

3 Array Statements as a Communication Abstraction

When we decouple the communication generation from the source language, we must make the important decision of the method of abstracting a communication step. The set of choices forms a spectrum. At one end of the spectrum is simplicity, and at the other end is expressibility. We must find a point in this spectrum that represents a reasonable tradeoff between the two extremes.

On the "simple" end of the spectrum, communication might be represented as a simple reference of an array element. In this abstraction, the language compiler takes sequential code, extracts the individual array references, and passes the names of the individual arrays, along with the subscripts, to the communication backend. While this interface is extremely simple, the expressibility is quite lacking. There is no way to represent the movement of aggregate data structures, since data movement is expressed essentially at the element level.

On the "expressible" end of the spectrum, we could imagine passing an entire loop nest, or perhaps some sort of dataflow graph, to the communication backend. The backend would perform analysis on this loop nest or graph and try to produce optimal parallel communication code that obeyed correctness constraints imposed by, e.g., data dependences. There are two problems with this approach. First, it would be difficult to come up with a representation that could adequately express the operations allowed by the parallel language, and it might also be cumbersome for the parallel language compiler to analyze its input code and convert it to the necessary format. Second, the communication backend would be burdened with performing a great deal of analysis on the input code passed on by the parallel language compiler. This analysis would duplicate work in this compiler and is therefore beyond the scope of our effort.

As an engineering compromise, we choose the *array assignment statement* as the level of communication abstraction. An array assignment statement is a Fortran-90 style assignment involving whole arrays or array sections. An example of such a statement is $A[1:n] = B[1:2n:2]$. Such array statements allow the concise description of the communication needs of the parallel language compiler. Note that we use the format of array assignment statements, but go beyond the restrictions on distributions embraced by data-parallel languages. To support the irregular computations of a compiler like Archimedes, we must widen the class of distributions that we allow.

There are three types of array assignment statements, serving three general purposes. The first type performs local computations or initializations, as in $A[2:n-1] = 0$. Another example is $A = B$, where A and B are known to have the same distribution. Although this kind of statement may not involve actual communication, it still needs to be handled by the communication backend, as discussed below. The second type of assignment statement is a redistribution, such as $A = B$, where arrays A and B have different distributions. The third type, and the most general, is one that performs actual computation as well as communication, such as $A = B + C$. For this type of statement, we use the owner-computes rule, which means that we temporarily redistribute B and C to have the same distribution as A, and then perform the local computation. All three statement types are handled by the communication backend, regardless of whether communication is actually involved.

There are several reasons that motivate our choice of array statements as a communication abstraction. First, these three types of array assignment statements are powerful enough to express parallelism for some important classes of problems, and to express the communication requirements for a wide variety of computations. Array assignment statements are succinct yet highly expressive when considered in conjunction with the wide spectrum of distribution possibilities. Second, array assignment statements are far easier for a compiler to analyze than loop nests. Third, the calling interface for passing the communication requirements from the parallel language compiler to the communication backend is far simpler than other options.

Array assignment statements have another important property: they can express both regular and irregular communication. A simple array assignment statement like $A = B$, where all arrays have regular distributions, results in a regular communication pattern. When either array has an irregular distribution, irregular communication results. In addition, when there is an indexing vector present, as in $A[IA[1:n]] = B[1:n]$, the same kind of irregular communication results, as the communication pattern is dependent on the contents of the indexing vector IA. This kind of array statement is powerful enough to express the communication requirements of many irregular programs, such as those written for Archimedes. Our communication backend uses many of the same techniques of regular communication generation to produce the communication for irregular array statements.

If the array statement involves computation, then some part of the tool chain must generate the appropriate code for this computation. At first thought, one might argue that it should be the parallel language compiler's responsibility to deal with the computation phase of an array assignment statement. This compiler should allocate the temporaries, invoke the communication backend to move the arrays into the temporaries, and then generate the code for the local computation on each processor. However, to perform the local computation requires knowledge of the global-to-local array index mapping function, and knowledge of other features managed by the communication backend, such as the format of the runtime distribution descriptors. For this reason, the communication backend must additionally generate code to perform the computation phase. This same argument holds for the first type of array assignment statement described above (e.g., $A = B$ where both arrays have the same distribution). Even though there is no communication, only the communication backend knows the global-to-local mapping function.

The communication backend cannot do all the computation, though. Imagine for instance an HPF "independent" loop. Although the communication backend can be used to temporarily redistribute the data arrays so that they are all aligned, the loop body contains references to array elements. The loop body may include any type of computation, and it is simply infeasible to try to incorporate all computational possibilities into the communication backend. Thus the parallel language compiler must compile the loop body, including applying the local memory mapping function to the array references. The communication backend provides an interface for performing the mapping, as well as methods for attempting to minimize the number of explicit mappings that need to be performed. For example, when advancing to the next local loop iteration, the local memory indices usually just increment by one; thus the parallel language compiler need not perform explicit local memory mappings for every loop iteration.

3.1 Compiling Array Statements

Here we give an overview of the process of generating the communication for both regular and irregular array assignment statements. First, some terminology: an *array section* is a set of array elements specified by a subscript triplet $(f : l : s)$, specifying the array elements beginning with index f, going no higher than l, and with access stride s. An *ownership set* is the set of array elements owned by a particular processor, as defined by the array distribution.

Regular The canonical regular array assignment statement is $A[f : l : s] = B[f : l : s]$, where A and B have block-cyclic distributions. (Block and cyclic are special cases of block-cyclic.) We reduce the problem to determining the portion of B that a particular source processor, s, sends to a particular destination processor, d. Let OA be the ownership set for d of array A, and let OB be the ownership set for s of array B. These block-cyclic ownership sets can be expressed compactly as a disjoint union of regular sections.

The set of elements of B that are sent is the intersection of three sets: $OA \cap OB \cap (f : l : s)$. In other words, s can only send elements that it owns, that are owned by d, and that are included in the array section in the assignment statement. (If the left-hand side section is different from right-hand side section, a simple linear mapping is applied to part of the intersection.)

We have developed algorithms [9] for efficient computation of these set intersections. These algorithms form the basis of the communication generation in Fx. Significant performance benefits are seen when one or both arrays have either block or cyclic distributions, because block and cyclic ownership sets are just regular sets, rather than unions of regular sets.

Irregular Array assignment statements can also be used to express irregular communication. There are two canonical irregular array statements. The first is $A[IA[f : l : s]] = B[f : l : s]$, where A, B, and IA all have regular distributions. The second is $A = B$, where one of the arrays has an irregular distribution. The index-to-processor mapping of the irregular distribution is typically given in a map array. Because the length of the map array is the same as that of the data array, the map array must itself be distributed, usually with a regular (most often block) distribution. Analysis of these two example statements is similar, interchanging only the index array and the map array.

In this paper we will not describe details of the communication generation for irregular array assignment statements. We note, however, that the regular section intersection techniques described above form an essential part of the irregular statement analysis. The important issue is that the array assignment statement can form a communication abstraction for both regular and irregular communication.

3.2 Example

A critical part of any communication step is identifying (and collecting) the local data that must be moved to another processor. This *buffer packing* code may involve a number of costly runtime operations, e.g., modulo, division, and floor/ceiling. Fig. 1 shows the code produced for the array assignment $A[1 : n : sa] = B[1 : n : sb]$, where A has a block distribution and B has a cyclic distribution, both over "numproc" processors. In the absence of information about the parameters, we obtain code as shown.

Fig. 2 depicts the customized code for filling the buffers that can be produced by the communication backend if all the parameters are known. The value of "n" has been instantiated to 1024×1024, "numproc" is 64, and "sa" and "sb" are both 1. (The actual transfer of data after buffer packing is the same in this case, and is therefore not shown.) In this case, the buffer packing code has been significantly optimized.

3.3 Alternatives

Compilers (and users writing programs) for a parallel system have a number of options to deal with communication. We distinguish between three different approaches: libraries, the inspector model, and use of a separate backend for communication. In this section, we briefly describe the approaches and discuss their advantages and disadvantages.

Libraries A popular approach for communication generation is to design a single large library function (or perhaps a small number of functions) that generates communication for any array assignment statement. The language compiler then only needs to parse

```
/* Find: g2 = gcd(sb,numproc) = sb*x2 + numproc*y2 */
euclid(&x2, &y2, &g2, sb, numproc);
stride = (sb * numproc) / g2;
if ((cellid < numproc) && (proc < numproc)) {
  bufptr = 0;
  last = MIN(n - 1,
    sb * floor(MIN(n - 1,
               ((dest * (((n + numproc) - 1) / numproc)) +
                (((n + numproc) - 1) / numproc)) - 1) / sa));
  lmlast = floor((last - cellid) / numproc);
  lower = MAX(sb * ceil((dest * blksize) / fx_sa), cellid);
  if ((cellid % g2) == 0) {
    r = ((cellid * (x2 * sb)) / g2);
    first = lower + mod(r - lower, stride);
    lmfirst = (first - cellid) / numproc;
    for (i=lmfirst; i<=lmlast; i += sb / g2)
      buf[bufptr++] = b[i];
    }
  }
```

Fig. 1. Non-customized buffer packing code.

```
bufptr = 0;
last = (dest * 16384) + 16383;
lmlast = (last - cellid) / 64;
lower = MAX(dest * 16384, cellid);
first = lower + mod(cellid - lower, 64);
lmfirst = (first - cellid) / 64;
for (i=lmfirst; i<=lmlast; i++)
  buf[bufptr++] = b[i];
```

Fig. 2. Optimized buffer packing code.

the array assignment statement and generate a parameter list to be passed to this library routine at run time. This approach suffers for a number of reasons:

- The arrays in the assignment statement could have an arbitrary number of dimensions. When generating code to pack/unpack communication buffers, we need a loop nest whose depth is proportional to the number of dimensions. This kind of variable-depth loop nest is notoriously difficult to write in C, often leading the function writer to impose arbitrary restrictions such as limiting the number of dimensions handled.
- It is extremely difficult to write a general-purpose function that efficiently handles all cases. The communication code can be significantly improved when we know certain properties of the assignment statement; e.g., the arrays all have block or cyclic distributions, the access stride of a regular section is 1, or the dimension size

is an even multiple of the number of processors over which it is distributed. In these cases and more, a general-purpose buffer-packing routine will probably generate highly suboptimal code.
- Another complication arises when the array reference mixes scalar subscripts and array sections. For example, $A[i, 1:n]$ must be treated differently from $A[1:n, i]$. Having to deal with this complication can be another large headache for the function writer.
- As a result of the above points, the single runtime function usually ends up a collection of many special-purpose routines. These specialized routines are usually added in an ad-hoc manner, as the need arises. As such, there exists no practical way of optimizing the communication routines. Furthermore, extra effort is spent at run time deciding which of the many special cases this statement might be.

For these reasons, we have decided against the library approach for communication generation.

Inspector Model Another approach for communication generation is based on the *inspector-executor* model, which forms the basis of the communication interface in the CHAOS runtime libraries [12]. This model is motivated by the observation that for many irregular computations, the elements of an array that participate in a communication step are often known only at runtime. So instead of attempting to compute the array elements involved in a communication step a priori, the compiler generates code that is executed before a communication step takes place. This inspector code finds out, at runtime, which array elements have to be moved to another processor. This information is then cached to save execution time in future invocations; if there has been no change in the data layout, the inspector phase can be skipped. Ruehl and Annaratone [6] extend this model to include symbolic analysis.

If absolutely no information about the communication requirements is known to the compiler (or communication backend), such an inspector-based approach may be the only option. However, in cases where such information exists, a communication backend is able to optimize at compile time, whereas an inspector-executor loop discovers this information only at runtime.

Customized Code Generation The third approach, customized code generation, involves compile-time generation of the communication code. This method can be viewed as a compile-time instantiation of the right set of communication library functions. Here, for every array assignment statement, the communication backend takes all compile-time information into account and generates a customized routine to perform the communication. Useful compile-time information includes:

- Whether the distribution of a particular array dimension is block, cyclic, or block-cyclic. Although block and cyclic distributions are just special cases of the general block-cyclic distribution, substantial performance gains are possible when a distribution is known to be either block or cyclic.
- Constant regular section (i.e., a Fortran-90 *subscript triplet*) bounds. This knowledge especially pays off when the access stride is known to be 1.

– Other distribution parameters, such as the number of processors or the block size of a block-cyclic distribution.

The main disadvantage of the customized approach over the library approach is that the customized backend is much harder to build. While it is relatively straightforward to convert an array assignment statement into a single procedure call, it is far from easy to generate customized code. This approach requires a substantial compiler framework, such as expression manipulation routines and constant folding. Nevertheless, customized code is guaranteed to be at least as good as library code, since in the worst case, one could simply generate the library routine blindly. Our communication backend takes the approach of generating customized code.

4 Details of the Communication Backend

We now discuss aspects of the design of our communication backend, which uses the customized code generation approach. The aspects discussed include details of retargeting to new parallel systems and language compilers, caching and reuse of communication patterns, and extensibility to add support for new data types and operators.

4.1 Optimization

The most critical portion of the regular communication generation routines, and the portion whose performance is most strongly affected by the code generation techniques of the communication backend, is the step of packing data into buffers. Other details, such as the actual low-level transfer of data across the communication network, are not nearly as susceptible to optimization by the communication backend and are not discussed here. The ultimate goal of the buffer packing optimization routines is to produce the simplest code to be executed at run time, thus minimizing the runtime overhead, especially overhead that occurs inside loops.

The communication generation routines for regular array assignment statements are based on algorithms derived for the most general regular array statements. To optimize these routines, the backend needs to instantiate compile-time constants and simplify the resulting expressions. We begin the optimization process by selecting the general purpose buffer-packing code segment. This first step involves making a decision regarding the code template that should serve as the basis for further optimizations. Unfortunately, there is no single master template that can be used in all cases; the choice depends on whether the distributions of the LHS and RHS arrays are block, cyclic, or the general block-cyclic. We have derived more specialized and efficient routines for when a distribution is known to be the simpler block or cyclic type; the general block-cyclic routines cannot be readily simplified to these specialized versions. Because each of the two arrays can have one of three types of distributions, the communication backend has nine possible code segments to choose from. These buffer-packing code segments are written in a completely general form, assuming that no information (e.g., array sizes, distribution block sizes, number of processors) is known at compile time.

After selecting the distribution-specific code segment, the rest of the optimizations are applied to the code: constant folding, constant/copy propagation, dead code elimination, and a new technique called interval folding.

Constant folding is integrated into the basic expression manipulation routines, which use a simple rule-based approach to find expressions susceptible to folding. To perform dead code elimination and constant/copy propagation, we form a flow graph from the buffer-packing code template, and we apply standard dataflow algorithms to the flow graph.

From a performance perspective, it could be argued that dead code elimination, constant folding, and constant/copy propagation are unnecessary, since these functions would also be performed by the optimizing C compiler that compiles the communication backend's output. However, from a software engineering point of view, these optimizations are necessary, as they help the communication backend developer identify bugs and evaluate the quality of the code being generated.

The final optimization, interval folding, is a new and simple but extremely effective optimization in the context of regular communication generation. To describe this optimization, we begin by observing that there are some variables and expressions that we know can take on only a limited range of values. For example, if the parallel system has 64 processors, we know that the *procid* variables is in the range $[0 : 63]$. As another example, even if we don't know the precise value of the distribution block size, we know it must be positive. Many expressions can be simplified based on these intervals.

We define our intervals in terms of two parameters: a lower bound and an upper bound, both of which are integer constants. A constant c has the interval $[c : c]$. We define a simple algebra for manipulating intervals across the expression operators. For example, if a has the interval $[a_l : a_h]$ and b has the interval $[b_l : b_h]$,

– $a + b$ has the interval $[a_l + b_l : a_h + b_h]$
– $a - b$ has the interval $[a_l - b_h : a_h - b_l]$
– $\max(a, b)$ has the interval $[\max(a_l, b_l) : \max(a_h, b_h)]$
– When $b_l > 0$, $\lfloor a/b \rfloor$ has the interval $[\lfloor a_l/b_h \rfloor : \lfloor a_h/b_l \rfloor]$
– Multiplication and division are slightly more complex in general, as the results depend on the sign of the input bounds.

There are two ways in which expressions can be simplified using interval information. The first involves the min and max operations. Continuing the examples above, if $a_h \le b_l$, then $\min(a, b) = a$. Likewise, if $a_l \ge b_h$, then $\max(a, b) = a$. The second simplification occurs when the lower bound and the upper bound are equal, in which case the expression just folds to the value of the upper/lower bound. As a somewhat typical example of this case, when there are 64 processors, $\lfloor (1024 - procid)/64 \rfloor$ folds to the value 15. This particular example arises naturally as part of the computation involving a 1024-element array block-distributed over 64 processors.

Our representation of intervals (i.e., a constant lower and upper bound) could be extended in a few ways. One extension is to include a stride value as well. For example, if p has the interval $[0 : 63]$, then $2p + 1$ has the interval $[1 : 127 : 2]$. We have found at least one example, involving block-cyclic distributions, in which this extension would lead to improved output. Another extension is to allow non-constant bounds on expressions. This extension would allow more optimizations when the number of processors is not known at compile time.

4.2 Reuse

Use of the communication backend is not limited to parallelizing Fortran compilers. Any parallel language that uses arrays for bulk storage (as opposed to, say, pointer-based structures) can use the communication compiler as its communication backend. There are two aspects of reuse: targeting a different language compiler to the communication backend, and retargeting the communication backend to a different parallel or distributed system.

Retargeting to Different Architectures The communication backend is designed to produce code for a wide variety of target architectures. When we talk about different architectures, we also include architecture-independent communication interfaces such as PVM or MPI. The backend defines a set of low-level communication routines (e.g., sending/receiving a message, broadcasting a value, barrier synchronization), so that only this small set of routines needs to be rewritten when moving to a new architecture.

There is another architecture dependent aspect of the communication backend that may need to rewritten for each target system, due to efficiency concerns. This aspect is the buffer packing/unpacking routines. Although the communication backend generates portable C code for buffer packing, the single-node optimizing C compiler might not be good enough to make the most effective use of the processor. In such a system, it would be desirable to rewrite these routines as hand-coded assembly loops.

Retargeting to Different Parallel Languages One of the fundamental design goals of the communication backend is portability, i.e., the ability to support multiple parallel compilers. This is one of the novel aspects of the communication backend. We do not want to lock our work into our own (or any other) HPF-like compiler. In fact, we do not even want to limit ourselves to Fortran as a base language. Thus we have abstracted the communication generation routines into a form that, with a little effort, can be integrated into any array-based language compiler.

Because the language compiler and the communication backend need to interact at compile time, we need a set of "interface routines" that convert internal expressions back and forth between the two compilers. Expressions are represented as a binary tree whose leaves consist of constants, symbols, function calls, and array references. Other information attached to symbols includes type information, array dimension sizes, and distribution information. These interface routines, which must be rewritten for every parallel language compiler, have the job of deconstructing the expression tree of one compiler and producing a tree for the other compiler.

One low-level aspect of the interface routines deserves to be mentioned. At least one language, Fortran, is incompatible with C (the language that the communication backend generates code in), with respect to its notion of row-major versus column-major memory layout of arrays. This incompatibility is easily reconciled by the interface routines. When an array is created, the dimension size list must be reversed, and the subscript lists of array references must also be reversed.

4.3 Caching of Communication Patterns

In general, irregular communication patterns are data-dependent; thus we cannot determine until run time what they will be. The pattern is dependent on either the contents of an indexing vector or the contents of a distribution array, which describes an irregular mapping of array elements to processors. The fact that indexing vectors and distribution arrays are large and therefore distributed means that extra global communication steps are required to perform the final communication analysis. In general, each indexing vector and each irregularly distributed array incurs a need for one additional global communication step. If possible, we would like to find ways to eliminate these extra global communications.

Frequently, these indexing vectors and irregular distributions reflect the physical aspects of the problem being simulated. The physical aspects often remain unchanged over many iterations of a loop, and thus the associated distributions and indexing vector contents also remain unchanged. Therefore, repeating the communication analysis will yield the same final communication pattern. This means that the communication backend can save a significant amount of work by caching and reusing the communication pattern.

The problem for the communication backend is that it only sees one communication statement at a time, and thus has no knowledge about how indexing vectors and distribution arrays might be changing between different loop iterations. The communication backend must be certain that the data describing the communication pattern has not changed since it was last cached. One possible solution is for the communication backend to maintain copies of all relevant variables, and then check these variables to determine whether any have changed since the last execution. The problem with this solution is that there is simply too much state to be saved. For example, if the array assignment statement includes an indexing vector, the communication backend must internally save the entire contents of the vector. Furthermore, because the vector is probably distributed, each processor must check to make sure its portion of the vector has not changed, and a global reduction over these values must be performed so that all processors know whether any processor's values have changed. This is probably more work than we want to do, especially given that better alternatives exist.

To address this problem, we look at the execution of the program at a higher level than individual statements. Only the parallel language compiler has this kind of global picture, so it must help out with the caching decisions. For some programs, the parallel language compiler can determine that the physical structure of the problem, and hence the values of the communication parameters such as indexing vectors, never changes throughout the computation. For this type of communication, the parallel language compiler can direct the communication backend always to use the cached pattern (except of course during the first iteration, before the communication cache is warmed up). Other programs might be highly adaptive, where the communication pattern changes every iteration, in which case the parallel language compiler can direct the communication backend never to even bother trying to cache.

The more interesting cases fall between these two extremes. For example, slightly adaptive algorithms might refine the structure only every few iterations, or in a purely data-dependent fashion, such as when the error exceeds a certain threshold. Here, we can reuse a communication pattern most of the time, but not always. To address this issue, the parallel language compiler can add code that tracks changes to the parameters in question. For the above examples, it might simply maintain a flag telling whether or not a refinement occurred during the current iteration. This flag would be passed to the communication routine at run time to tell whether to reuse the previous communication pattern.

The language compiler has to decide at what level of granularity it will attempt to track changes to the parameters. If tests are inserted too far inside loop nests, the overhead of tracking changes might outweigh the caching benefits. On the other hand, if the language compiler is overly conservative and inserts checks too high in the loop nest, communication pattern reuse might happen less often than otherwise possible. Discussion of loop instrumentation techniques is beyond the scope of this paper.

Communication pattern reuse can also be useful for regular communication [2]. For example, while block and cyclic distributions are fairly simple to generate communication for, communication induced by the more general block-cyclic distribution can be considerably more expensive to determine. In this kind of communication pattern, caching can yield a significant performance improvement. For regular communication, it may turn out to be best for the communication backend to track all parameter changes, since there are far fewer such parameters than for irregular communication.

4.4 Extensibility

Building a standalone communication backend provides interesting challenges, primarily relating to the fact that we cannot anticipate or hope to accommodate all possible parallel language compilers. The first step in addressing this problem was to define the interface routines. Having interface routines addresses the fact that virtually all compilers use a different data structure for representing expressions than the communication backend uses.

But there is a more subtle problem. Many parallel languages will use operators (e.g., plus, minus, mod) that the communication backend does not support, or data types that the backend does not support. There is no way in general for the interface routines to handle this problem. Thus we plan to make the communication backend extensible so that code can be added to handle new operators and types.

To give a concrete example, suppose that the communication backend were written only to interface with parallel C-like languages, and that we wanted to integrate it with an HPF compiler. The backend would need to be extended to deal with Fortran's "complex" data type (disregarding the fact that the current prototype of the communication backend already treats complex as a first-class data type).

Recall that in addition to handling communication, the backend produces code to do some amount of computation as well. Extending the communication portion is fairly simple: we need only inform the backend what the size of the data type is, so that it knows how many words to move around at run time. Extending the computation portion is somewhat trickier. The user needs to provide C code that will be emitted when computation on data arrays needs to happen at run time. In the example of the complex data type, the user must provide code for performing complex additions, complex multiplications, etc. When adding a new operator, code must be provided to perform the operation on each of the data types. The code can be provided in terms of a function call, or preferably a macro.

A more ambitious goal for communication backend extensibility is to provide a mechanism for giving programmers a well-defined interface for adding new types of distributions. We are currently exploring the possibilities of adding this functionality.

5 Related Work

To our knowledge, ours is the only system that generates customized communication code at compile time for array assignment statements. The other systems described below use either the runtime library approach or the inspector-executor model for communication generation.

In terms of the low-level buffer packing algorithm, our method is closest to that of Sadayappan et al. [4]. Our method of representing ownership sets corresponds to their *virtual cyclic* ownership representation. A substantially different approach was pioneered by Chatterjee et al. [1], and improved upon by Thirumalai and Ramanujam [13], as well as Kennedy, Nedeljkovic, and Sethi [5]. This approach uses a *finite state machine* to generate tables which are used for address generation. All of these methods are applicable only for regular array assignment statements.

Most systems for irregular communication are based on the inspector-executor method. CHAOS [12] is the predominant such system; parallel programs must be converted into an inspector-executor style, and the resulting code is linked against the CHAOS runtime libraries to do the communication and memory management. The Vienna Fortran Compilation System [14] allows the specification of irregular computations, and then automatically transforms the program into a version that invokes CHAOS.

6 Concluding Remarks

Optimizing communication is important, since the communication behavior of an application has a direct impact on performance and scalability. Communication operations are on the critical path for many regular problems. Sometimes it is possible to overlap communication and computations for these problems. However, for irregular problems, any overlap is difficult or impossible to realize, since the exact details of the communication may depend on the results of the computation steps.

A separate communication backend is a practical solution for parallel compilers that map regular and irregular computations. It provides a clean abstraction between the parallel language compiler, which deals with issues like deciding on a suitable data layout and distribution of the computations, and the communication system.

A communication backend allows us to reuse the generation of communication operations for different parallel compilers. The effort lowers the implementation cost of a parallel compiler; given such a backend, it is much easier to explore different approaches to data partitioning, automatic alignment, etc. At the same time, the communication backend provides one central place to apply communication optimizations, and all compilers can benefit from them.

Acknowledgements

We appreciate the contributions by our colleagues Peter Dinda, Susan Hinrichs, David O'Hallaron, Jonathan Shewchuk, Peter Steenkiste, Tom Stricker, Jaspal Subhlok, and Po-Jen Yang.

References

1. S. Chatterjee, J. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of 4th PPoPP*, pages 149–158, San Diego, CA, May 1993.
2. P. Dinda and D. O'Hallaron. The impact of address relation caching on the performance of deposit model communication. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1995. To appear.
3. T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.
4. S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *Proceedings of ICPP*, , St. Charles, IL, August 1993. IEEE.
5. K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of 5th PPoPP*, pages 102–111, Santa Barbara, CA, July 1995.
6. R. Ruehl and M. Annaratone. Parallelization of Fortran code on distributed-memory parallel processors. In *Proc. Intl. Conf. on Supercomputers*. ACM, Jne 1990.
7. E. J. Schwabe, G. E. Blelloch, A. Feldmann, O. Ghattas, J. R. Gilbert, G. L. Miller, D. R. O'Hallaron, J. R. Shewchuk, and S. Teng. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical solution of pdes. In *Proceedings of the 1992 DAGS/PC Symposium*, pages 48–62, June 1992.
8. J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.
9. James M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.
10. T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symp. on Computer Architecture*, page accepted, Portofino, Italy, June 1995. ACM/IEEE.
11. J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Programming task and data parallelism on a multicomputer. In *Proceedings of 4th PPoPP*, pages 13–22, May 1993.
12. A. Sussman, G. Agrawal, and J. Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Tech Report CS-TR-3070.1, University of Maryland, December 1993.
13. A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *Proceedings of 8th Workshop on Languages and Compilers for Parallel Computing*, page to appear, Columbus, Ohio, August 1995. Springer Verlag.
14. H. Zima, P. Brezany, B. Chapman, P. Mehrota, and A. Schwald. Vienna Fortran – a language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.