2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Toward Abstracting the Communication Intent in Applications to Improve Portability and Productivity

Tiffany M. Mintz, Oscar Hernandez, Christos Kartsaklis,
David E. Bernholdt, Markus Eisenbach
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: {mintztm, oscar, kartsaklisc, bernholdtde, eisenbachm}@ornl.gov

Swaroop Pophale
University of Houston
Houston, TX, USA
Email: spophale@cs.uh.edu

***Abstract*—Programming with communication libraries such as the Message Passing Interface (MPI) obscures the high-level intent of the communication in an application and makes static communication analysis difficult to do. Compilers are unaware of communication libraries' specifics, leading to the exclusion of communication patterns from any automated analysis and optimizations. To overcome this, communication patterns can be expressed at higher-levels of abstraction and incrementally added to existing MPI applications. In this paper, we propose the use of directives to clearly express the communication intent of an application in a way that is not specific to a given communication library. Our communication directives allow programmers to express communication among processes in a portable way, giving hints to the compiler on regions of computations that can be overlapped with communication and relaxing communication constraints on the ordering, completion and synchronization of the communication imposed by specific libraries such as MPI. The directives can then be translated by the compiler into message passing calls that efficiently implement the intended pattern and be targeted to multiple communication libraries. Thus far, we have used the directives to express point-to-point communication patterns in C, C++ and Fortran applications, and have translated them to MPI and SHMEM.**

***Keywords*-message passing; directives; communication; remote data transfer; global address space; parallel computing; scientific computing**

I. Introduction

In order for applications to move to Exascale, it is essential for existing message passing applications to have a road map to explore other programming models that exploit different types of communication, such as one-sided or explore a hybrid approach. We also need to deal with challenges such as reducing the data motion in the application, hiding communication latencies and reducing the memory footprint of the code. Any transformation that overlaps computation with communication, relaxes synchronizations, and improves the data layout of communication data structures will help.

Traditionally, communication within parallel programs are implemented by programmers using libraries such as MPI, SHMEM, and PVM. While this allows programmers to construct an extensive variety of communication patterns, and manage their communication explicitly, the drawback to this approach is that the application becomes difficult to maintain and optimize, and locks an application to a given communication library or paradigm. In addition, few compiler tools are available to analyze existing MPI applications because it is difficult to describe the SPMD data flow and control flow of the application because the communication intent is difficult to derive and compilers are not aware of how to match sends and receives and the semantics of the MPI library. For these cases, over-conservative assumptions may be needed statically, that prevents optimizations. The compiler also has to understand the correct ordering of the calls and their matching synchronizations.

A step forward to improve this is to incrementally rewrite the communication portions of the application using directives to raise the level of abstraction of the communication intent where the communication control flow structure is well understood. The directives can then more easily be tuned by compilers (or users) and translated to multiple communication libraries and paradigms including one-sided libraries such as OpenSHMEM, ARMCI, MPI-2, etc. Raising the level of abstraction in the communication helps to define its intent, and compilers can benefit from this information to make sense of the communication that affects structured control flow regions of code. The directives can be translated to communication calls where all source and destination information can be incorporated into an analysis framework for automated analysis and optimization. The directives also enable opportunities for reusing structured communication patterns on different code regions. In addition, the incremental, re-targetability properties of having communication directives make them easier to use and incorporate into existing applications, as opposed to incrementally introducing a new language such as Chapel, UPC and CAF, where interoperability may be an issue. Our directives can target one communication library associated with a region of a code or target multiple communication libraries. Where some regions may use MPI and others SHMEM depending on the properties and types of communication (i.e. message sizes, communication data structures layouts, etc).

In this paper, we present our first efforts to define a set of

978-0-7695-4979-8 2013

DOI 10.1109/IPDPSW.2013.66

communicating directives, and demonstrate that the directives can be used in the context of point-to-point communication. There are a variety of point-to-point communication patterns that are recurring in scientific applications[1], [2], [3]. These patterns were used as the basis for determining a sufficient interface for the communication directives. The initial directives can successfully manage the point-to-point communication in scientific codes by encoding the pattern of communication, automatically generating data type information, as well as reusing buffers when needed and targeting the directives to MPI or SHMEM where it makes more sense.

The remainder of this paper is organized accordingly: Section II gives a summary of related work. Section III presents an overview of the structured communication that is enabled through the use of the directives. Section IV describes point-to-point communication intensive portions of a scientific application that has been modified to use the directives. Lastly, Section V concludes this paper with a summary of our current work and future efforts.

II. Related Work

There are many efforts dedicated to clarifying the purpose and use of communication. Many of these projects design their communications to fit common data structures such as vectors and matrices, and are often domain specific. For instance, Basic Linear algebra Communication Subprograms (BLACS)[4] is a linear algebra oriented message passing interface. The operations in this library are expressed in terms of two dimensional matrices, where vectors and scalars are treated as a subclass of matrices. It is intended to provide easy to use, and portable communication across distributed memory systems for linear algebra problems.

A similar library is Portable Extensible Toolkit for Scientific Computation (PETSc)[5]. PETSc is a suite of parallel linear, nonlinear equation solvers and time integrators. These routines and data structures can be used in Fortran, C, C++, Python and MATLAB codes. It is organized into several libraries that manage a specific family of objects and operations to be performed on these objects. Examples of the family of objects are index sets, vectors and matrices.

While these libraries provide some degree of portability, communication intent is expressed at the expense of general purpose use. The communication calls in these libraries target specific domains and data structures which make them less desirable for more general communication. These libraries also have some of the same drawbacks as general purpose communication libraries when automatic static analysis and optimizations are desired by the user. Where as our communication directives help to express communication intent for broader use with the added benefit of possible automatic static analysis and optimizations.

There are also directive-based language extensions for data parallelism that aim to increase productivity and reduce programmer effort to parallelize code. One commonly known implementation is High Performance Fortran (HPF)[6], [7]. HPF is designed to support data parallel programming. While HPF is noted for its directive notation for distributing data on parallel machines, the language extension also provides data parallel constructs, intrinsic functions and a standard library. In [8], which is an extension to HPF+ (similar to HPF), additional directives are offered for conveying hints to the compiler in the form of communication schedules regarding the behavior of irregular communication patterns, thus leading to more efficient communications code.

OpenMPD[9] is another approach that uses directives to support data parallelism. It includes features for array distribution and work sharing. The array distribution occurs on global arrays with every process being assigned a partition of the array for a given computation. Work sharing is expressed using a `for` directive with an optional `affinity` clause for explicit loop iteration mapping.

Yet another data parallel language extension is OpenMPI[10]. Not to be confused with the MPI library implementation of the same name. OpenMPI is a set of OpenMP-like directives for creating a data parallel version of an application. This research has developed a set of directives for distributing arrays, work sharing and performing data transfers commonly found in MPI programs, such as parameter broadcasting and scalar and vector reduction.

Though each of these directive implementations simplify the expression of parallelism, they only focus on data parallelism and do not address issues related to message passing communication. Our directive is not intended to exploit parallelism. The primary focus of this research is the expression of message passing communication.

With this in mind, there has been some recent research in the area of directive annotations for communication using MPI. In Bamboo[11] directives are used as an annotations mechanism that clarifies intent and dependence among blocks of communication. It is implemented on top of the ROSE compiler and intends to convert MPI programs into a form that is easier to manipulate using data flow techniques. To accommodate for the varying sparsity in communications and relative overheads, Bamboo supports communication layouts at the directive levels that help with scenarios such as data decomposition and nearest neighbors. While this research provides a mechanism for static analysis and optimization of MPI only, it does not address portability or provide a higher level expression of communication.

Our communication directives go beyond annotations and provide programmers with a complete expression for message passing communication. This expression not only encompasses the intent of the communication but also presents the opportunity for automatic static analysis and optimizations.

III. Communicating with Directives

To begin the process of establishing a directives based expression for message passing, we implemented new communication directives in the Open64 compiler. The Open64 compiler accepts Fortran 77/90 and C/C++. It is a well-written compiler, performing state-of-the-art analysis on five levels of intermediate representations, where each level targets specific analysis and optimizations such as inter procedural and data flow analysis.

In the the next sections we introduce our communication directives, and give a brief overview of how they can be used to express message passing using regions of structured communication.

A. Overview of Directives

We have developed a set of directives that help to improve the way that programmers express message passing communication. These directives provide a communication interface that requires only basic information for transferring data between processes, and at the same time, allows programmers to associate communication with computation to construct regions of communication within their code.

The first of the directives, called `comm_parameters`, is used to encapsulate a region of code where a programmer can assert a set of communication parameters that apply to all communication within the region. The next directive is called, `comm_p2p`, and can be used in conjunction with `comm_parameters` or independently. This directive defines an instance of point-to-point communication and a region of computation that can overlap communication at run time.

The directives can be used in C, C++ and Fortran to generate MPI or SHMEM library calls, enable structured message passing communication at both the application and translation levels. At the application level, `comm_parameters` and `comm_p2p` are used with a list of clauses that associate groups of processes and buffers with an entire region of code in an application. This region of code defines a scope of communication similar to that of a function scope which can be used at compile time to enable minimal synchronization, communication/computation overlap, and optimal generation of message passing calls.

The two directives have eight clauses in common, and `comm_parameters` have two additional clauses that can be used to further structure the communication. The eight common clauses are `sender`, `receiver`, `sbuf`, `rbuf`, `sendwhen`, `receivewhen`, `target` and `count`. These clauses are used to define the relevant processes and buffers that will be used in the communication scope, as well as the data payload within the scope. When a combination of these clauses are used with the `comm_parameters` directive, their assertions apply to all instances of `comm_p2p` that occur within the `comm_parameters` region. So, individual instances of `comm_p2p` in this scope do not need to re-expresses these communication clauses, but may provide additional assertions for the communication if necessary.

While each of these clauses provide relevant communication information, they are not all required when using the directives. A programmer may choose to exclude the `sendwhen`, `receivewhen`, `target`, `count`, `max_comm_iter` and `place_sync` clauses, and allow the compiler to infer this information through analysis or to apply default behavior. Furthermore, `max_comm_iter` and `place_sync` may only be used with `comm_parameters` in order to facilitate the generation of synchronization calls. Section III-B provides examples and brief descriptions of each clause.

In addition to providing a scope of communication and a higher level structure for expressing message passing communication, the directives offer increased portability in the form of multiple translation options and data type handling, as well as, analysis for reduced synchronization. As stated earlier, the directive is translate to MPI or SHMEM calls. Each of these libraries have unique mechanisms for handling data types. With SHMEM, data type selection is tightly coupled with the communication call, in that the data type is embedded in the name of the library call. So for efficient communication, the buffers data type should match the type expressed in the name of the communication call. This matching is performed by the compiler, and communication calls that match the data type and storage size of the buffers are generated.

With MPI, we take advantage of MPI's derived data type functionality, and pass specific MPI types to the communication call. For buffers of primitive types, the C/C++ or Fortran type is mapped to an MPI basic type during compilation and passed to the MPI library call. If the buffer is a composite type, information about the type is extracted at compile time to dynamically create an MPI struct. For each element in the composite type, its displacement within the type, block length and correlating MPI basic type are accumulated into corresponding arrays to be used at run time. (Pointers within a composite type are prohibited as well as recursively nested composite types.) Once the displacement, block length and MPI data type arrays are populated, MPI library calls are generated to create and commit an MPI struct type. This new MPI data type is reused within the function scope for any communication directive with buffers of the same type.

In order to reduce synchronization calls, the directive performs a set of automatic analysis on adjacent `comm_p2p` directives. For every set of adjacent `comm_p2p` directives with independent buffers, synchronization is consolidated and reduced in most cases to one call at the end of all the adjacent communication.

B. Communication Clauses

The communication directives have ten clauses. Four of these clauses are required, six are optional and two

of the optional clauses may only be used with the comm_parameters directive. The required clauses are sender, receiver, sbuf and rbuf. The sender and receiver clauses accept expressions that evaluate to process ids (for MPI these process ids are relative to the MPI_COMM_WORLD communicator). For the sender clause, the expression evaluates to the id of the process that sends to the current process. The expression in the receiver clause evaluates to the id of the process that receives the message sent by the current process.

The sbuf and rbuf clauses accept a list of buffers that are the origin and destination of the message, respectively. These buffers must be pointers or arrays of primitive or composite type, and its allocation must conform to the requirements of the intended communication library. For example, SHMEM library implementations require that memory allocated to a buffer be symmetric; so, if the programmer indicates that the directive should perform the communication using SHMEM library calls, the buffers in sbuf and rbuf must also be symmetric data objects. Listing 1 gives an example of how to express a ring communication pattern using the required clauses.

Listing 1. Ring communication pattern using required clauses

```
prev = (rank-1+nprocs)%nprocs;
next = (rank+1)%nprocs;
#pragma comm_p2p sender(prev) receiver(next)
  sbuf(buf1) rbuf(buf2)
```

The optional clauses are sendwhen, receivewhen, target, count, place_sync and max_comm_iter. The sendwhen and receivewhen clauses accept Boolean expressions. The Boolean expression in the sendwhen clause is used to determine which processes send a message. If the sendwhen expression evaluates to true on a process, that processes buffer in the sbuf clause is the origin of a message. If the sendwhen clause is not present, all processes that reach the directive will send the message. The Boolean expression in the receivewhen clause is used to determine which processes receive a message. If the receivewhen expression evaluates to true on a process, that processes buffer in the rbuf clause is the destination of a message. If the receivewhen clause is not present, all processes that reach the directive will receive the message. While both these clauses are optional, the current implementation of the directive requires that they be used together. So they both must be present or both be omitted. Listing 2 shows a simple example of how a programmer could construct a communication pattern of processes with even process ids sending to the nearest odd numbered process.

Listing 2. Grouping processes with sendwhen and receivewhen clauses

```
#pragma comm_p2p sbuf(buf1) rbuf(buf2)
  sender(rank-1) receiver(rank+1)
  sendwhen(rank%2==0) receivewhen(rank%2==1)
```

The target clause accepts the keywords: TARGET_COMM_MPI_1SIDE, TARGET_COMM_MPI_2SIDE and TARGET_COMM_SHMEM. These keywords indicate which library calls the programmer wants to generate. If TARGET_COMM_MPI_1SIDE is specified, the directive will generate an MPI_Put to transfer the data between processes. If TARGET_COMM_MPI_2SIDE is specified, a non-blocking MPI_Isend and non-blocking MPI_Irecv will be generated to transfer data between processes. If TARGET_COMM_SHMEM is specified, a shmem_put that corresponds to the buffers' type size is generated to transfer data. If the target clause is not present, the default library calls that are generated are MPI non-blocking send and receive.

The count clause is an expression that evaluates to the number of elements of the sender's buffer(s) that will be passed to the receiver's buffer(s). The count clause may be omitted if a buffer in either sbuf or rbuf is an array. If one of these buffers is an array and count is omitted, the directive will generate code with a message size equal to the array size. If more than one of the buffers is an array, the message size will be the size of the smallest array.

The final clauses, place_sync and max_comm_iter, may only be used with comm_parameters. place_sync accepts the keywords END_PARAM_REGION, BEGIN_NEXT_PARAM_REGION and END_ADJ_PARAM_REGIONS. These keywords indicate where to place synchronization calls. If END_PARAM_REGION is specified, synchronization calls will be placed at the end of the comm_parameters region. BEGIN_NEXT_PARAM_REGION indicates that the synchronization calls should be place at the beginning of the next comm_parameters region, END_ADJ_PARAM_REGIONS delays all synchronization to the last comm_parameters region in a series of adjacent instances of comm_parameters.

The max_comm_iter clause should be used when loops are present in the comm_parameters region. The expression in this clause indicates the maximum number of iterations that a comm_p2p instance may be executed, which will facilitate code generation for synchronizations. Listing 3 gives an example of how the optional clauses may be used.

Listing 3. Use of optional clauses

```
#pragma comm_parameters sender(rank-1)
  receiver(rank+1) sendwhen(rank%2==0)
  receivewhen(rank%2==1) count(size)
  max_comm_iter(n) place_sync(END_PARAM_REGION)
{
 for(p=0; p < n; p++)
   #pragma comm_p2p sbuf(&buf1[p]) rbuf(&buf2[p])
}
```

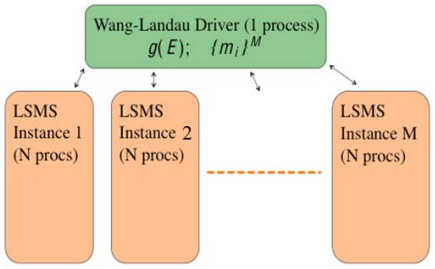

Figure 1. Modular view of WL-LSMS application[12]

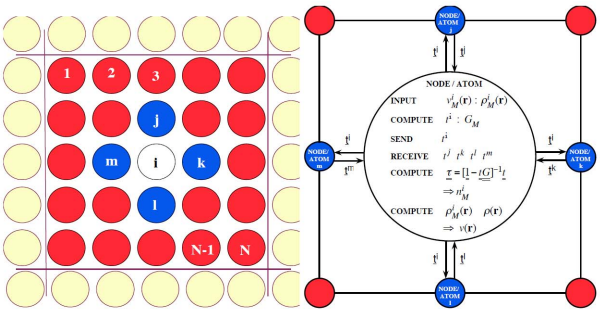

Figure 2. Formation of local interaction zone[12]

IV. Using Communication Directives in Scientific Application

To demonstrate how the communication directives can be used in scientific applications, we have implemented the directive in WL-LSMS (Wang-Landau Locally Self-Consistent Multiple Scattering)[12]. WL-LSMS is a thermodynamics application used to calculate density functional based first principles electronic structure. The code combines Wang Landau (WL), a Monte Carlo calculation for processing the Density Functional Hamiltonian of a physical system, with Locally Self-Consistent Multiple Scattering (LSMS), a method for first principles electronic structure calculation. In the application, there are one WL instance, *M* LSMS instances and *N* processes executing each instance of LSMS. A diagram of this modular configuration is shown in Figure 1. The WL process communicates with a designated privileged process in each LSMS, and within each LSMS the privileged process communicates with the other non-privileged processes. This area of communication within an LSMS is called the local interaction zone (LIZ). Figure 2 shows the LIZ formation and its internal communication pattern.

A. WL-LSMS with Communication Directives

Nearly all of the communication in WL-LSMS is point-to-point, with the majority of the point-to-point communication occurring within each LIZ. We have selected two significant portions of this communication to implement using the directives. The first is the initial distribution of the system's potentials and electron densities. Each atom in the system has corresponding potential and electron density data, and this atom data is distributed among the processes in each LSMS. Listing 4 shows the original code for distributing single atom data to the non-privileged processes in an LIZ.

Listing 4. Original Code to transfer single atom data

```
1
2   if(comm.rank==from)
3   {
4     int pos=0;
5     MPI_Pack(&local_id,1,MPI_INT,buf,s,&pos,comm.comm);
6     MPI_Pack(&atom.jmt,1,MPI_INT,buf,s,&pos,comm.comm);
7     MPI_Pack(&atom.jws,1,MPI_INT,buf,s,&pos,comm.comm);
8     MPI_Pack(&atom.xstart,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
9     MPI_Pack(&atom.rmt,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
10    MPI_Pack(atom.header,80,MPI_CHAR,buf,s,&pos,comm.comm);
11    MPI_Pack(&atom.alat,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
12    MPI_Pack(&atom.efermi,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
13    MPI_Pack(&atom.vdif,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
14    MPI_Pack(&atom.ztotss,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
15    MPI_Pack(&atom.zcorss,1,MPI_DOUBLE,buf,s,&pos,comm.comm);
16    MPI_Pack(atom.evec,3,MPI_DOUBLE,buf,s,&pos,comm.comm);
17    MPI_Pack(&atom.nspin,1,MPI_INT,buf,s,&pos,comm.comm);
18    MPI_Pack(&atom.numc,1,MPI_INT,buf,s,&pos,comm.comm);
19
20    t=atom.vr.n_row();
21
22    MPI_Pack(&t,1,MPI_INT,buf,s,&pos,comm.comm);
23    MPI_Pack(&atom.vr(0,0),2*t,MPI_DOUBLE,buf,s,&pos,comm.comm);
24    MPI_Pack(&atom.rhotot(0,0),2*t,MPI_DOUBLE,buf,s,&pos,comm.comm);
25
26    t=atom.ec.n_row();
27
28    MPI_Pack(&t,1,MPI_INT,buf,s,&pos,comm.comm);
29    MPI_Pack(&atom.ec(0,0),2*t,MPI_DOUBLE,buf,s,&pos,comm.comm);
30    MPI_Pack(&atom.nc(0,0),2*t,MPI_INT,buf,s,&pos,comm.comm);
31    MPI_Pack(&atom.lc(0,0),2*t,MPI_INT,buf,s,&pos,comm.comm);
32    MPI_Pack(&atom.kc(0,0),2*t,MPI_INT,buf,s,&pos,comm.comm);
33
34    MPI_Send(buf,s,MPI_PACKED,to,0,comm.comm);
35  }
36  if(comm.rank==to)
37  {
38    MPI_Status status;
39    MPI_Recv(buf,s,MPI_PACKED,from,0,comm.comm,&status);
40
41    int pos=0;
42    MPI_Unpack(buf,s,&pos,&local_id,1,MPI_INT,comm.comm);
43    MPI_Unpack(buf,s,&pos,&atom.jmt,1,MPI_INT,comm.comm);
44    MPI_Unpack(buf,s,&pos,&atom.jws,1,MPI_INT,comm.comm);
45    MPI_Unpack(buf,s,&pos,&atom.xstart,1,MPI_DOUBLE,comm.comm);
46    MPI_Unpack(buf,s,&pos,&atom.rmt,1,MPI_DOUBLE,comm.comm);
47    MPI_Unpack(buf,s,&pos,atom.header,80,MPI_CHAR,comm.comm);
48    MPI_Unpack(buf,s,&pos,&atom.alat,1,MPI_DOUBLE,comm.comm);
49    MPI_Unpack(buf,s,&pos,&atom.efermi,1,MPI_DOUBLE,comm.comm);
50    MPI_Unpack(buf,s,&pos,&atom.vdif,1,MPI_DOUBLE,comm.comm);
51    MPI_Unpack(buf,s,&pos,&atom.ztotss,1,MPI_DOUBLE,comm.comm);
52    MPI_Unpack(buf,s,&pos,&atom.zcorss,1,MPI_DOUBLE,comm.comm);
53    MPI_Unpack(buf,s,&pos,atom.evec,3,MPI_DOUBLE,comm.comm);
54    MPI_Unpack(buf,s,&pos,&atom.nspin,1,MPI_INT,comm.comm);
55    MPI_Unpack(buf,s,&pos,&atom.numc,1,MPI_INT,comm.comm);
56
57    MPI_Unpack(buf,s,&pos,&t,1,MPI_INT,comm.comm);
58
59    if(t<atom.vr.n_row())
60      atom.resizePotential(t+50);
61
62    MPI_Unpack(buf,s,&pos,&atom.vr(0,0),2*t,MPI_DOUBLE,comm.comm);
63    MPI_Unpack(buf,s,&pos,&atom.rhotot(0,0),2*t,MPI_DOUBLE,comm.comm);
64
65    MPI_Unpack(buf,s,&pos,&t,1,MPI_INT,comm.comm);
66
67    if(t<atom.nc.n_row())
68      atom.resizeCore(t);
69
70    MPI_Unpack(buf,s,&pos,&atom.ec(0,0),2*t,MPI_DOUBLE,comm.comm);
71    MPI_Unpack(buf,s,&pos,&atom.nc(0,0),2*t,MPI_INT,comm.comm);
72    MPI_Unpack(buf,s,&pos,&atom.lc(0,0),2*t,MPI_INT,comm.comm);
73    MPI_Unpack(buf,s,&pos,&atom.kc(0,0),2*t,MPI_INT,comm.comm);
74  }
```

In order to more clearly express this communication, we organized the scalar data into a single structure, and grouped each matrix according to it's communicated data payload, taking advantage of the directive feature that allows lists of buffers to be specified in the `sbuf` and `rbuf` clauses. The organization of communication requires only three instances of the `comm_p2p` directive which are arranged within a single region of the `comm_parameters` directive. Since

comm_p2p performs data type handling, explicit packing for contiguous data transfer or derived data type creation is no longer necessary. This results in fewer lines of code and more clearly expressed communication. Listing 5 shows the directive version of the communication.

Listing 5. Code to transfer single atom data using directive

```
#pragma comm_parameters sendwhen(rank==from_rank)
  receivewhen(rank==to_rank)
  sender(from_rank) receiver(to_rank)
{
  #pragma comm_p2p sbuf(scalaratomdata)
    rbuf(scalaratomdata) count(1)
  { }

  #pragma comm_p2p vsbuf(vr,rhotot)
    rbuf(vr,rhotot) count(size1)
  { }

  #pragma comm_p2p sbuf(ec,nc,lc,kc)
    rbuf(ec,nc,lc,kc) count(size2)
  { }
}
```

The other portion of the application that is communication intensive is the transfer of random spin configurations to all processes in an LIZ. The random spin configurations are generated by the WL process and initially distributed to the privileged processes in each LIZ. The privileged process then sends this data to the non-privileged processes in order to calculate the energies of the system. Listing 6 shows the original communication that occurs in the setEvec routine.

Listing 6. Original setEvec routine

```
if(rank==0)
{
  for(int p=0; p<num_types; p++)
  {
    MPI_Isend(&ev[3*p],3,MPI_DOUBLE,n,l,comm.comm
        ,&request[n_req++]);
  }
  for(int i=0; i<n_req; i++)
    MPI_Wait(&request[i],&status);
} else {
  for(int p=0; p<num_local; p++)
  {
    MPI_Irecv(&local.atom[p].evec[0],3,MPI_DOUBLE
        ,0,p,comm.comm,&request[p]);
  }
  for(int i=0; i<num_local; i++)
    MPI_Wait(&request[i],&status);
}
```

Once a process receives the random spin configuration that corresponds to its atom data, it executes a series of operations that compute individual energy values. The first of these computations occurs on data that is not dependent on the random spin configurations; so, this computation can be overlapped with the communication. Listing 7 shows the directive version of the communication in *setEvec* overlapped with the initial computation for computing individual energy values. In this version of the code, the computation is very easily and clearly overlapped with the communication providing opportunity for optimized performance. Synchronization calls are also efficiently generated using the max_comm_iter and place_sync clauses with the comm_parameters directive.

In both instances where we have implemented the communication directives in place of explicit message passing library calls, we have eliminated the need for major code revisions to implement message passing calls other than MPI sends and receives. The directive enables portability with very minimal effort by the programmer. The programmer would only need to assert the desired message passing implementation using the target clause. In the next section we show performance results using the directives for both MPI and SHMEM targets.

Listing 7. Communication/Computation overlap with directive

```
while((rank == 0 && send_p < num_types)
     || (rank != 0 && recv_p < num_local))
{
 #pragma comm_parameters sendwhen(rank == 0)
   receivewhen(rank != 0) sender(rank0)
   receiver(rcv_rank) count(3)
   max_comm_iter(num_types)
   place_sync(END_PARAM_REGION)
 {
  while((rank == 0 && n == types[send_p].node)
       || (rank != 0 && recv_p < num_local))
  {
   #pragma comm_p2p sbuf(&ev[3*send_p])
     rbuf(&local.atom[p].evec[0])
   {
     calculateCoreState(comm,lsms,local,recv_p,!
         core_states_done);
   }
  }
 }
}
```

B. Performance Results

In addition to facilitating structured communication that more closely resembles the communication intent, the directives also generate message passing calls that exhibit comparable or better performance in our case study than the original MPI communication. This is demonstrated in our experimental results for the communication that is described in the previous section. The experiments were executed on a Cray XK7 system with 77 compute nodes. Each node has a 16-core AMD Opteron 6274 processor running at 2.2 GHz with 32 gigabytes of DDR3 memory, and Cray's high performance Gemini network. We perform each experiment on sixteen iron atoms, and compare the performance of WL-LSMS without using the communication directives and the performance when the communication directives are used for both MPI 2sided and SHMEM targets.

In our first experiment we measured the communication of the system's potentials and electron densities (referred to as single atom data). For the MPI target, the directive implementation utilizes MPI's derived data type creation feature to define an MPI struct that maps to the scalar data in this communication. The directive also automatically

reduces synchronization calls for both MPI and SHMEM targets to one synchronization call for the adjacent `comm_p2p` directives. As figure 3 illustrates, our experimental results show comparable performance for both targets to the original single atom data communication.

The next experiment measures the communication of random spin configurations that occurs within the setEvec routine. This communication is executed within the main loop on the processes that make up each LSMS. In this experiment there was considerable improvement in performance for both MPI and SHMEM targets. The MPI target gives an average speedup of about 4x and SHMEM has a 38x average speedup. This increase in performance in the case of MPI is primarily due to the difference in synchronization. In the directive implementation, synchronization is placed at the end of the communication region which generates an MPI_Waitall at the end of the region. This synchronization is a much more optimal synchronization than a loop over MPI_Wait for every non blocking send and receive, which is the synchronization in the original communication. To validate this claim, we changed the synchronization in the original communication to an MPI_Waitall for each loop. This modification improved performance by about 2.6x over the original communication, which yielded performance gains of about 1.4x for MPI and 14.5x for SHMEM directive targets. As for the significant speedup using SHMEM communication calls, this is primarily due to the bandwidth and latency differences of MPI and SHMEM[13], [14]. When taking into account both these factors in MPI and SHMEM implementations, their differences are most prominent when transferring small messages (8 to 256 bytes). So in scenarios with frequent small point to point data transfers and very little synchronization relative to the number of data transfers, SHMEM is more likely to out perform MPI. Figure 4 provides a graph plot of the results.

Our final experiments measured the random spin communication and the initial computation for calculating energy values in the calculateCoreStates routine. In the directive version of the application, the energy value calculations are overlapped with the communication. Since the overall ratio of computation time to communication time in WL-LSMS is 19 to 1, computation is by far the dominant factor when overlapping communication and computation. So this overlap is significantly bounded by the time to execute the calculateCoreStates routine. Therefore this optimization provides an improvement in performance of at most the time to communicate random spin configurations.

Further work is being done to optimize the computation in WL-LSMS by porting the application to a GPU implementation. This optimized implementation is projected to deliver as much as a 10x speed up over the current implementation. Given this 10x speed up of the energy value calculations, we computed the computation/communication overlap execution time for the directive implementation and

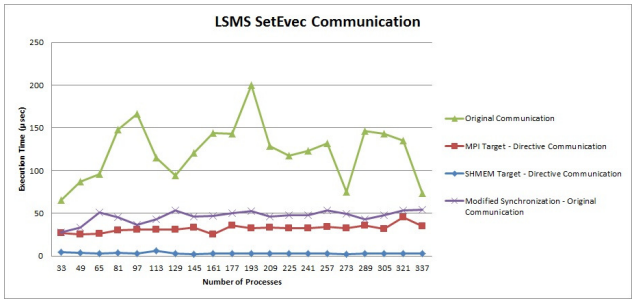

Figure 4. Experimental results for communication of random spin configurations

compared this to the original communication with optimized computation. These results are shown in figure 5.

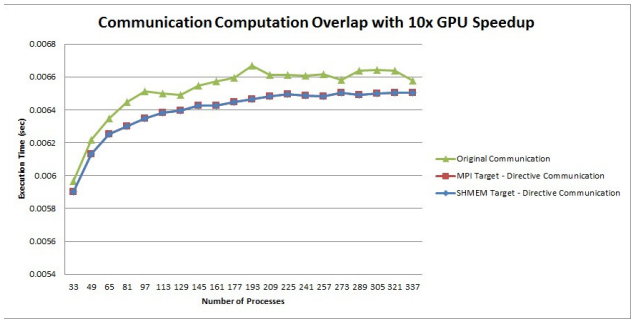

Figure 5. Execution time for directive communication/computation overlap

V. Conclusion

As the previous section demonstrates, the directive approach to expressing communication is beneficial for expressing point-to-point communication in scientific applications. The directives and its clauses clearly express a communication pattern, defines the roles of the processes in the communication and any conditions and data structures relevant to the communication. The directive also enables automatic data type handling with code generated by the compiler which increases the user's productivity. The communication directives can also be translated to several communication libraries that use two-sided or one-sided communication, allowing a programmer to express the communication intent of complex communication patterns in a simple and portable way.

While this is the first step toward expressing the communication intent in scientific applications, there are further advances to be made in the development of this approach. To provide a more complete solution, we are working to extend the directives to express groups of processes, and their collective communication/synchronization in a variety of many-to-one, one-to-many and all-to-all patterns. We plan

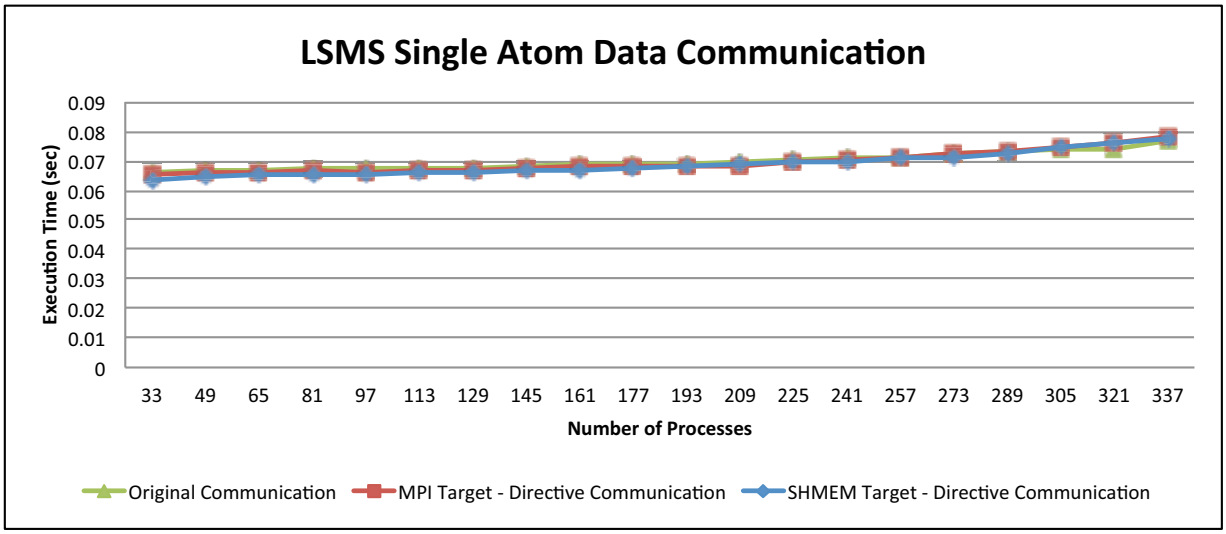

Figure 3. Experimental results for communication of single atom data

to use this high level communication to extend compilers' data flow analysis, to understand the dependencies between communication and computation, relax synchronization, and provide a way to understand how communication patterns affect the program's data and the communication requirements of an application.

ACKNOWLEDGMENT

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy, including the use of resources of the Oak Ridge Leadership Computing Facility. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. This manuscript has been authored by a contractor of the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

REFERENCES

[1] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 853 – 865, 2003, ¡ce:title¿Special Section on the Best Papers from the 2002 International Parallel and Distributed Processing Symposium¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731503001047

[2] J. Kim and D. J. Lilja, "Characterization of communication patterns in message-passing parallel scientific application programs," in *Proceedings of the Second International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, ser. CANPC '98. London, UK, UK: Springer-Verlag, 1998, pp. 202–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=646092.680542

[3] R. Riesen, "Communication patterns [message-passing patterns]," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006, p. 8 pp.

[4] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn, "Basic linear algebra communication subprograms," in *Distributed Memory Computing Conference, 1991. Proceedings., The Sixth*, apr-1 may 1991, pp. 287 –290.

[5] S. Balay, J. Brown, , K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.3, 2012.

[6] G. Fox, *Draft High Performance Fortran Language Specification: High Performance Fortran Forum, CRPC-TR92225, November 1992*. Computer Information Technical, Jun. 1991.

[7] D. Loveman, "High performance fortran," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 1, no. 1, pp. 25 –42, feb 1993.

[8] S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima, "High-level management of communication schedules in HPF-like languages," in *Proceedings of the 12th international conference on Supercomputing*, ser. ICS '98. New York, NY, USA: ACM, 1998, pp. 109–116. [Online]. Available: http://doi.acm.org/10.1145/277830.277855

[9] J. Lee, M. Sato, and T. Boku, "Openmpd: A directive-based data parallel language extension for distributed memory systems," in *Parallel Processing - Workshops, 2008. ICPP-W '08. International Conference on*, sept. 2008, pp. 121 –128.

[10] T. Baku, M. Sato, M. Matsubara, and D. Takahashi, "Openmpi : Openmp like tool for easy programming in mpi," in *Sixth European Workshop on OpenMP*, ser. EWOMP'04, 2004, pp. 83–88.

[11] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, "Bamboo: translating MPI applications to a latency-tolerant, data-driven form," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 39:1–39:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389050

[12] M. Eisenbach, C.-G. Zhou, D. M. Nicholson, G. Brown, J. Larkin, and T. C. Schulthess, "A scalable method for ab initio computation of free energies in nanoscale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 64:1–64:8. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654125

[13] H. Shan and J. P. Singh, "A comparison of mpi, shmem and cache-coherent shared address space programming models on a tightly-coupled multiprocessors," *Int. J. Parallel Program.*, vol. 29, no. 3, pp. 283–318, Jun. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011120120698

[14] E. Strohmaier and H. Shan, "Apex-map: A global data access benchmark to analyze hpc systems and parallel programming paradigms," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 49–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.13