Parallel Remote Method Invocation and M-by-N Data Redistribution

Kostadin Damevski and Steven Parker
School of Computing
University of Utah
Salt Lake City, UT 84112
{damevski,sparker}@cs.utah.edu

Abstract

Components can be a useful tool in software development, including the development of scientific computing applications. Many scientific applications require parallel execution, but commodity component models based on Remote Method Invocation (RMI) do not directly support the notion of parallel components. Parallel components raise questions about the semantics of method invocations and the mechanics of parallel data redistribution involving these components.

Allowing parallel components to exist within a component framework comes at very little extra cost to the framework designer. However, the interaction semantics (i.e. method invocations) between two parallel components or between a parallel and non-parallel component can be complex and should require support from the underlying runtime system.

The parallel data redistribution problem comes about when in order to increase efficiency, data is subdivided among cooperating parallel tasks within one component. When two or more components of this type are required to perform a separate computation on the same data, this data distribution must be decoded and mapped from the first component to the second component's specification.

We demonstrate a method to handle parallel method invocation and perform automatic data redistribution using the code generation process of an interface definition language (IDL) compiler. The generated code and runtime system accomplish the necessary data transfers and provide consistent behavior to method invocation. We describe the implementation of and semantics of Parallel Remote Method Invocation (PRMI). We describe how collective method calls can be used to provide coherent interactions between multiple components. Preliminary results and benchmarks are discussed.

1. Introduction and related work

Component technology is an important and widely used tool in software development. However, commodity component implementations do not support the needs of high performance scientific computing. To remedy this, the CCA (Common Component Architecture) [1] Forum was formed among various universities and research institutions, including the University of Utah, Indiana University, various Department of Energy research laboratories, and others. The group's goal is to add the functionality of components to existing scientific computing code while preserving the speed required by these applications.

Parallelism is a tool that is consistently leveraged by many scientific programmers in order to increase performance. As a result, the ability to support parallel components is crucial when trying to provide components for scientific computing. The choice of parallel programming model for this purpose is SPMD (Single Program Multiple Data). Parallel components can be based on MPI (Message Passing Interface), PVM (Parallel Virtual Machine) or any other product that facilitates this very common type of parallel programming. Allowing parallel components to exist within a component framework comes at very little extra cost to the framework designer. However, the interaction semantics (i.e. communication) between two parallel components and between a parallel and non-parallel component is not obvious and may benefit support from the component framework. This is especially true when the number of processes differ between the two interacting components.

Many component models rely upon an underlying object model that provides Remote Method Invocation (RMI). RMI provides method call functionality over a network, typically through a network proxy that is juxtaposed between two components, the caller and callee. RMI enables a component to interact with another without requiring knowledge of its internal structure. Extending this to the parallel component case is termed PRMI (Parallel Remote Method Invocation). Possible ways of defining PRMI can be seen in the work of Maasen et al. [7], however there is no evidence to date of a definite decision on the optimal semantics of PRMI. From a practical standpoint, a policy that will describe expected behavior when utilizing PRMI is required.

M-by-N data redistribution is also an important piece of the high performance scientific components puzzle. The M-by-N problem comes about when in order to increase efficiency, data is subdivided among M cooperating parallel tasks within one component, and N parallel tasks in a different component, where M and N are both positive integers. When two or more components of this type are required to perform a separate computation on the same data, this data distribution has to be decoded and mapped from the first component to the second component's specification. Since each component can require a different data distribution this problem can get complicated. Also, since components can be dynamically composed at runtime their distribution requirements may not be known prior. Therefore, data transfer schedules must be computed on the fly.

The M-by-N problem has been discussed among scientific computing researchers for a substantial period of time and systems such as PAWS [2], CUMULVS [3] and others have been developed which solve this problem for the limited case of multi-dimensional arrays. All of these systems are based on a specific data transfer API (Application Programming Interface). The data transfer API is separate from the actual method invocations or other control flow. These systems have created a general solution to the M-by-N problem. However, each of them has a unique API and has taken a different approach to data representation and the timing, locking, and synchronization of the data transfer. The emergence of the CCA group has created a unique opportunity to attempt to create a component framework standard for each of these issues.

In addition, some of the M-by-N work suggests that in order to achieve maximum flexibility in the design, a specific M-by-N redistribution component needs to be in place [6]. This M-by-N redistribution component will stand between two components which require a data distribution and perform this distribution for them. We recognize the flexibility of this design, however, we argue against it because of its inherent inefficiency for many problems, and because of semantic differences between parallel and non-parallel interactions. We believe that this inefficiency stems from the fact that this component would make unnecessary copies of data to its address space in a distributed component environment. The semantic differences stem from the fact that RMI is used between non-parallel components, yet parallel-to-parallel interactions are performed using this alternate mechanism.

In another approach, we based our M-by-N redistribution mechanism on PRMI, treating distributed data as another method call argument in a PRMI invocation. Naturally, this choice led us to placing all of the necessary pieces to perform data redistribution in the Interface Definition Language (IDL) compiler and the stub/skeleton code it generates. This decision was similar to the design of PARDIS [5], which uses the CORBA IDL to express distributed arguments in a component environment. However, in order to provide some necessary flexibility in the design, our system also relies on two methods to describe the data distribution at runtime. The primary contribution of this work is providing parallel remote invocation semantics that would maximize the expressiveness of parallel components. By placing a useful tool such as M-by-N data redistribution on top of these parallel method invocation semantics we show that parallel remote method invocation and automatic data redistribution mechanisms can be combined with the help of an interface definition language compiler, resulting in a tool that can benefit a scientific software programmer. Moreover, this mechanism preserves the fundamental features of components, namely that a component should not be required to expose implementation details, in this case the details of internal data decomposition.

The design of the PRMI and M-by-N data redistribution has been implemented within the SCIRun2 component framework, a Problem Solving Environment (PSE) designed for large-scale parallel computation. This framework is compliant to the Common Component Architecture (CCA) framework specification. Specifically, the compiler upon which our design is based uses CCA's interface language, the SIDL (Scientific IDL). The PRMI and M-by-N data redistribution are solely implemented through the SIDL compiler and its runtime support library.

The discussion that follows will assume a distributed environment. The text is organized as follows: Chapter 2 gives an overview of our PRMI approach. The M-by-N data redistribution mechanism is discussed in chapter 3. Chapter 4 gives an overview of our array representation and the data redistribution schedule calculation. In chapter 5 we show several preliminary performance results. Finally, chapter 6 and 7 explains conclusions and future work of this project.

2. Parallel Remote Method Invocation

Parallel Remote Method Invocation occurs when a request is made for a method invocation on a parallel component or when a parallel component itself makes such a request. Since many options exist in PRMI design and it is not reasonable to provide them all, the ultimate goal in the design of the PRMI semantics was to cover most of the reasonable programming decisions in the design of parallel components. In order to accomplish this we recognized two types of PRMI behavior: collective and independent. We have extended the SIDL to use the keywords **collective** and **independent** as descriptors for each method. By default, methods are independent.

2.1. Collective PRMI

Collective components are defined as a collaboration of multiple processes that represent one computation [6]. A method invocation between components of this type can be inferred as one that requires the involvement of each parallel process/thread on the component, typically in unison. Examples of such behavior include all cases where collaboration (intra-component communication through MPI or some other library) is required between the component's parallel processes in order to solve a task. Many classic parallel algorithms fit this programming approach. Several cases are possible within the collective PRMI. These are: M=N, M>N, and M<N where M is the number of parallel processes on the caller component and N is the number of parallel processes on the callee component. Figure 1 describes the behavior of our system under each case, while ensuring the desired semantics.

Collective calls are required to be invoked on all caller processes, and are guaranteed to be invoked only once every callee component. These calls are synchronous and non-overtaking. Their conception was motivated by the need to perform M-by-N data redistribution. In fact, the collective calls and data redistribution usually appear in unison. Within a collective invocation, there is no guarantee in place that determines which caller process communicates with which callee process. This suggests that all input and output data between each process of the method invocation should be the same, which is standard practice for collective operations using a SPMD programming model. Complex communication patterns can be expressed more effectively using M-by-N data redistribution. Synchronization primitives are in place to make successive calls non-overtaking. However, there is no implicit barrier primitive between the processes on the caller or on the callee when the method is invoked. This allows a degree of asynchronous behavior that increases efficiency. However, aspects of M-by-N data redistribution involve additional synchronization. These will be discussed later.

2.2. Independent PRMI

We recognized the need to provide some support for cases not involving invocations made on every participating process. We name these independent calls. The independent invocation suggests that any component process can and will satisfy the request. The assumption is that all of the parallel processes provide the same functionality to the user. One example of this is a parallel component implementing a *getRandomNumber()* method. All processes of this component have the functionality of producing a random number. Whenever this method is invoked we care only that a component produces a random number. We are not concerned which of the parallel process satisfy this request. The sole support we provided for the independent invocation is the ability to turn off the collective mechanism and direct the independent call as a regular method call to a component's parallel process.

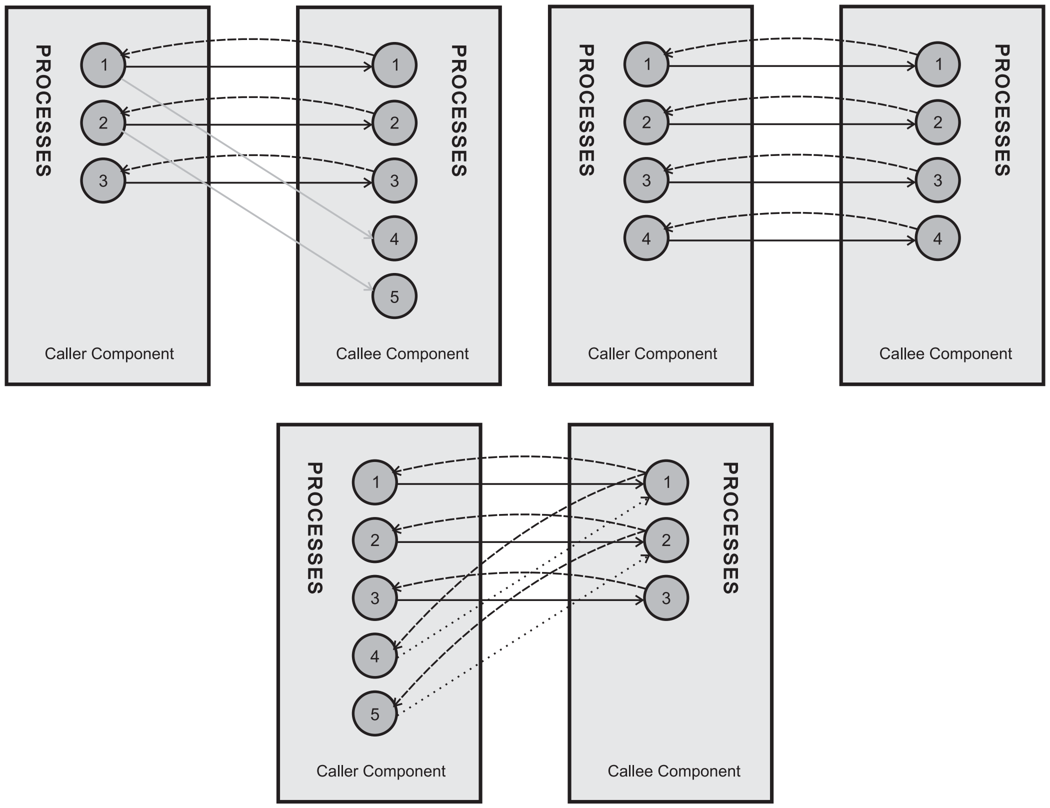

Figure 1. Scenarios for interaction between two parallel components. M is the number of processors in the caller, and N is the number in the callee. (top-right) M = N. The whole lines represent method invocations, while the dashed represent return values. (top-left) M < N. The whole lines represent method invocations, while the dashed represent return values. The grey lines stand for invocations which take place without the knowledge of the process out of which they were initiated. (bottom) M > N. The whole lines represent method invocations, while the dashed represent return values. The dotted lines stand for invocations that are special reply requests.

3. M-by-N Data Redistribution

M-by-N data redistribution builds upon PRMI, where the data is redistributed automatically when the invocation is made. We limited the M-by-N problem to the case of multi-dimensional arrays on any number of dimensions. Furthermore, the approach we took in solving the M-by-N problem was in treating the redistribution data as another parameter in the parallel method invocation. In order to express this, we extended the CCA's SIDL specification to provide a distribution array type. This defined type is used in method signatures to represent a distribution array of a certain type and dimension. We chose to define a distribution array type separate from usual method and class definitions. This definition was chosen in order to limit the need to declare a distribution array, its dimensions and type for each use of the array. Furthermore, we have bound this type to a specific distribution and allow reuse of specific distributions and calculated data redistribution schedules using this mechanism. Redistribution schedule calculation is expensive in terms of communication so we wanted to allow mechanisms for its reuse. The type definition follows the expected scoping rules so that it is only valid in the scope it is defined and any nested scopes. The distribution array type can be included in more than one method declaration and would signify a distribution array as a parameter to a particular method. An example of the changes to the SIDL can be seen below.

```
package PingPong {
  distribution array D <int, 1>;
  interface PingPong {
    collective int pingpong(in D test);
  };
};
```

This is a pingpong example of the modified SIDL that represents M-by-N distribution types. The type “D" is defined as a one-dimensional distributed array of integers that is bound to a particular distribution schedule. In this example, the collective keyword is not strictly necessary, as the compiler will assume that methods containing a distribution type are always collective.

As we compiled the IDL code that contained a distributed array type, the stub/skeleton code changed significantly by adding the necessary code to perform the data redistribution. When executed, this code performs the necessary distributions. By doing this, we have alleviated the component programmer from any responsibility of redistributing the data. The distribution is done on a point to point basis in such a way that a data transfer link is established directly between two processes that require transfer of data. Synchronization primitives are in place so that the method does not execute until all of the data is completely transferred, even when the data is sent from multiple sources. The current system provides a data redistribution mechanism for *in*, *out* and *inout* arguments as well as return values.

In addition to the IDL modifications, two methods were provided in order to express and exchange the data distribution from each process' perspective at runtime:

```
setCalleeDistribution(
        DistributionHandle dist_handle,
     MxNArrayRep array_representation);

setCallerDistribution(
        DistributionHandle dist_handle,
     MxNArrayRep array_representation);
```

Each of these methods was designed to be called collectively by all of the callee and caller parallel processes respectively. Their end purpose is to make the infrastructure aware of the data distribution that the caller *has* and the distribution that the caller *wants*. Both of them expect the same group of arguments: a handle for the distribution parameter in question and a description of the array distribution that a particular process contains. Our implementation of the M-by-N data redistribution does not require the user to report the dimensions of the global array. The system infers this information. The array representation will be described in more detail below.

The immediate action of the *setCalleeDistribution* method is to establish the fact that a particular component is a callee in respect to a particular distribution. Also, the proper distribution objects are created and the callee waits to receive the distribution metadata from all of its callers. The *setCallerDistribution* method, on the other hand, performs a scatter/gather to all of the callee processes exchanging the appropriate metadata.

When the *setCallerDistribution* method is complete in all of the participating processes, both the callee and caller processes have the necessary metadata and the necessary objects instantiated that will perform the data distribution. We have chosen to report distributions at runtime since this provides more flexibility to the component writer. A lot of data distributions depend on the number of processes under which they are executed, therefore making it much more convenient for users of our system to support the reporting of distributions at runtime. A disadvantage of this particular decision is that it requires the redistribution schedule to be calculated at runtime.

4. Array Representation and Transfer Schedule

To represent the array distribution we use the PAWS [2] data model. It consists of the first element of the array, the last element of the array, and a stride denoting the number of array spaces in between two elements. For instance, the data representation (first = 0, last = 100, stride = 2) for an array named *arr* would represent the array starting at arr_0, ending at arr_{100}, and taking every second element in between (i.e. arr_0, arr_2, arr_4, arr_6 ... arr_{100}). Using some of PAWS's terminology we call this description an index, and we use one index to describe each dimension of the array in question.

A distribution schedule is expressed through a collection of intersecting indices. These indices, which are obtained by intersecting two of the regular data representation indices, describe the exact data that needs to be transferred between a given callee and caller process. The intersection of indices is in fact the calculation of the redistribution schedule. The intersection is of two indices at a time, so that indices are intersected for the same dimension of the two processes' array representations. The intersection algorithm rests upon Euclid’s theorem [11]:

```
(Thm.): There exists i, j in Z such that
a*i+b = c*j+d if and only if b-d=0 mod
gcd(a,c)
```

This theorem directly motivates the following algorithm for the intersection of array indices (in pseudo C/C++):

```
//Calculate lcm and gcd of the strides:
int lcm_stride = lcm(stride1,stride2);
int gcd_stride = gcd(stride1,stride2);

//Find first and stride of intersection
intersectionIndex->stride = lcm_stride;
if (first1 % gcd_stride) ==
    (first2 % gcd_stride) {

  extended_euclid(stride1,stride2,
                  &m,&n);
  I = first1 - (stride1*m*
     (first1-first2))/gcd_stride;
  J = I + lcm_stride * ceil(
      (max(first1,first2) -I)
      /lcm_stride);
  intersectionIndex->first = J;
  //Find the last
  intersectionIndex->last =
              min(last1,last2);
}
else {

  //No Intersection
  intersectionIndex->first = 0;
  intersectionIndex->last = 0;
  intersectionIndex->stride = 0;
}
```

The index intersection pseudo-code uses helper functions for lcm (Least Common Multiple), gcd (Greatest Common Divisor) and the the extended Euclid's algorithm. The extended Euclid’s algorithm finds the greatest common divisor, **g**, of two positive integers, **a** and **b**, and coefficients, **m** and **n**, such that **g = ma + nb**.

This algorithm provides us with an efficient method of calculating the redistribution schedule that is able to adapt to all possible combinations of first, last and stride. The algorithm can also adapt to negative strides. To our knowledge, this fully general algorithm has not been published previously.

5. Preliminary Results

A series of experiments have been performed and their results will be discussed in this chapter. The intention of these experiments was to demonstrate the functional capabilities of our system and to quantify the invocation overhead of the PRMI, with and without data redistribution. The purpose of this work was to facilitate high-performance component implementations, and not to develop them. Therefore, these results supply a validation to the design of our system, but are not intended to be ends by themselves.

The following tests were performed on a 256 node Dell PowerEdge 2650 cluster. Each node in this cluster contains two Inter P4 Xeon 2.4 GHz CPUs. The nodes run version 8.0 of Red Hat Linux and 100 Megabit Ethernet was used as the network interconnect.

The system we developed can be used by a variety of applications. In order to demonstrate this, we implemented a few “classic” parallel algorithms, including LU matrix factorization, Jacobi’s solution to the LaPlace heat equation, and odd-even merge sort. Each of these applications provides a different data redistribution pattern to exercise the system and show that all of them can be expressed. These examples ranged between 2 and 3 components. We experimented with different numbers of processes per each parallel component. Figure 2 shows a component implementation of the odd-even merge sort algorithm. This sorting algorithm was particularly easy to implement using the data redistribution capability that our system provides. The implementation relies on 3 components: sorter, splitter and starter. The started component manages the execution of the problem. The splitter component requires no internal implementation, as its purpose is only to act as a data redistributing proxy which splits and combines the even and odd elements of the array to be sorted. The sorter component implements a simple merge sort algorithm. This implementation of odd-even sort relies heavily on the M-by-N data redistribution mechanism to perform aspects of the algorithm itself. It is also important to note that this implementation is not recursive and due to that requires a number of parallel processes per each component corresponding to the number of arrays to be merged. This is a weakness of this algorithm design, not the underlying array redistribution mechanism.

The following tables express the execution times our system exhibited while executing the characteristic applications we implemented. To fully stress the system, all applications were executed so that each parallel process was located on a separate cluster node (all times are in microseconds).

LaPlace Heat Equation	
Matrix Rank	Time
10	0.023293
50	0.73736
100	8.37276
150	32.7804
200	43.4538
250	78.431
300	131.924

Experiment used a 1 to 4 M-by-N distribution.

LU Factorization	
Matrix Rank	Time
40	0.035614266
100	0.14665567
160	0.473564593
200	0.676691
320	1.14721401
600	5.68645942
2000	60.994763

Experiment used a 2 to 6 M-by-N distribution.

Odd-Even Merge Sort	
Matrix Rank	Time
500	0.010483
1000	0.011189
2000	0.016644
5000	0.032934
10000	0.063244
50000	0.148052
100000	3.003914

Experiment used a 1 to 2 to 2 M-by-N distribution.

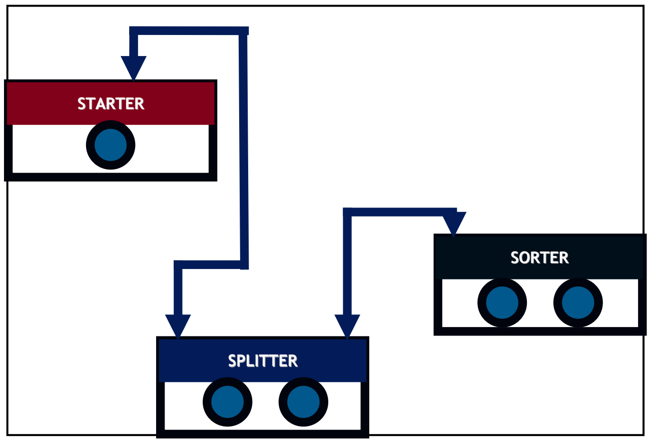

Figure 2: Components implementing the odd-even merge sort algorithm. Circles within components represent number of parallel processes.

5.1. Overhead Analysis

To be useful for scientific applications, this system must achieve high performance. Our system is a prototype and still requires several optimizations, but demonstrates that these mechanisms can be effective in parallel to parallel data transfer. This analysis of the overhead our system imposes on the execution of an application has the intention of providing us with a clear picture of our system, therefore allowing improvements to minimize the overhead. We used the applications we designed to run a series of experiments, noting the exact time required to accomplish the PRMI and M-by-N data redistribution from the perspective of one participating process. One experiment had the intention of partitioning and analyzing the data redistribution on one caller process. The benchmark was made on an application that was solving a 2000 by 2000 matrix using LU Factorization. The results of this experiment are provided in Figure 3.

The experiment shows that data marshaling is the most expensive task in the redistribution from the caller perspective. The data are marshaled element by element, constantly requiring the underlying communication library to be invoked. Currently, SCIRun2 relies on the Nexus communication library from the Globus Toolkit [12, 13, 14] for this purpose. The data marshaling is a task whose time share grows as the data grow larger and will be considered as a prime candidate in system optimization. Most notably, the Nexus API does not facilitate marshaling of strided data, and therefore requires a separate functional cal for each data element. Another interesting aspect of this experiment was the relatively brief time (0.23% of total redistribution time) it took for the system to calculate the redistribution schedule. The total time the caller process took to redistribute the data was 4.74 seconds. We measured 2.66 seconds for a representative callee process to receive and assemble the data in the same application.

As an additional method of determining the overhead imposed by the PRMI and M-by-N data redistribution we measured method call times of different types of invocations. We measured the invocation time (time to execute a method which does nothing) of all the invocation types our system supports. The experiment was executed so that every process had a specific node to execute on, making all invocations distributed. The parallel component cases used one process for the caller and four callee component processes. The chart in Figure 5 shows preliminary results of this experiment. For the method invocations involving data redistribution, the callee requires a cyclic distribution, so the caller component must marshal and send every 4th array element (4 byte integers) to each of the callees. Figure 4 illustrates this benchmark setup.

This experiment requires data to be sent from the caller to each of the 4 callees. The collective invocation is able to mask the call latencies so that the collective call is only 2.8 times as long as a serial invocation. This is an improvement over the factor of 4 that you would expect if the caller performed these calls in sequence, and this ratio improves as more processors are added. Data redistribution (shown by the last two bars of Figure 4) was considerably more costly than a collective invocation. This is due to inefficiencies in our implementation of data packing and unpacking mechanisms rather than with PRMI itself. We also have not yet implemented reuse of communication schedules, so the schedule must be computed and communicated on every call. We believe that future optimizations, largely targeted to the underlying communication infrastructure, will dramatically increase the performance of our system.

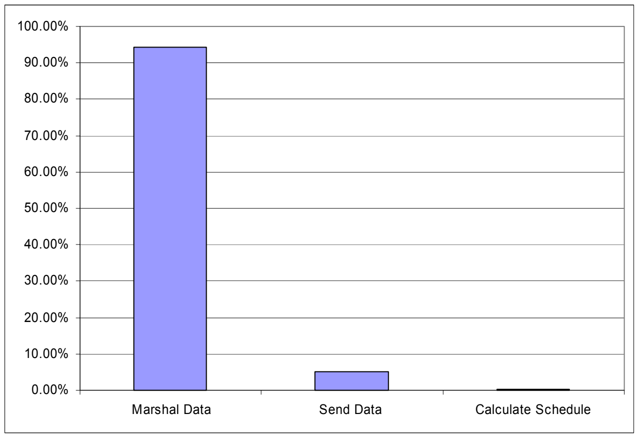

Figure 3: Overhead breakdown of the data redistribution of a 2000-by-2000 matrix.

6. Conclusions

This paper described an approach in building automatic M-by-N data redistribution through an Interface Definition Language (IDL). In order to accomplish this, we described a method for Parallel Remote Method Invocation (PRMI). Our goal was to illustrate a way of handling PRMI that would be simple and would encompass most invocation scenarios. We described our extensions to the Scientific IDL (SIDL), as well as the two methods we provided to report a component's distribution at runtime. Finally, we showed initial performance results of our system while executing a characteristic application.

Using a combination of an IDL and runtime array distribution API to perform these tasks is a novel approach in handling complex data distributions and PRMI. It allows a programmer to redistribute data from one component to another automatically, while allowing a significant degree of freedom for the

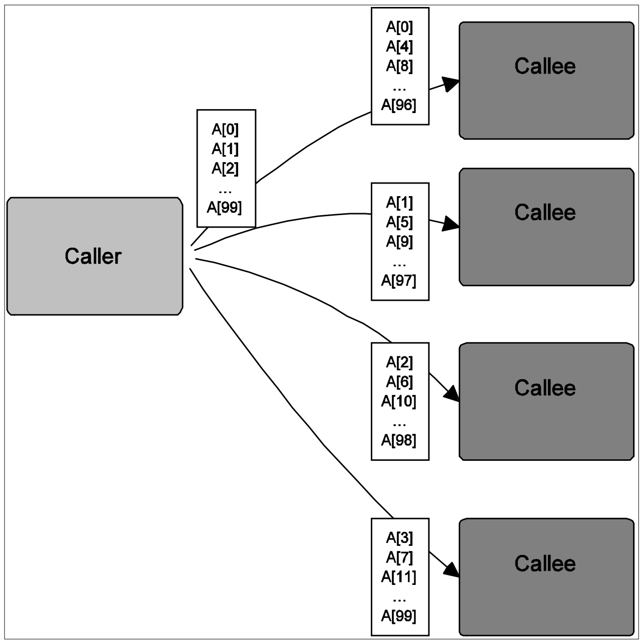

Figure 4: Benchmark setup where a single caller process splits a contiguous array into 4 strided sections.

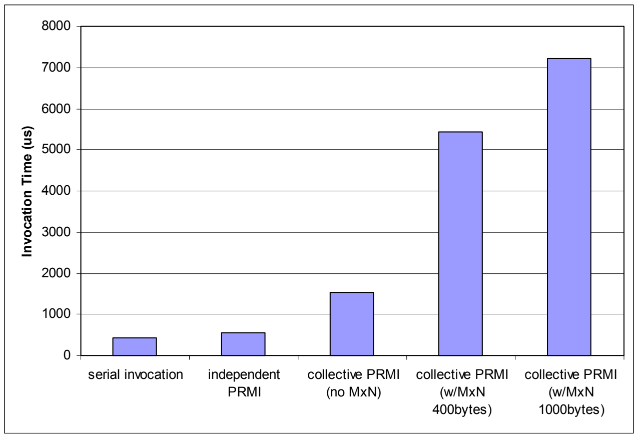

Figure 5: Comparison of invocation types supported by the SCIRun2 system.

prgrammer to have direct control over the redistribution process.

7. Future Work

A feature that we value as very important to the success of out project is the ability to express subsets of processes upon which we can leverage the PRMI and M-by-N data redistribution. For instance, a set of parallel processes on the caller and callee component could be specified as ones that should participate in a collective call, therefore relaxing the requirement of all processes to be involved in the invocation. This will increase flexibility as well as allow for our system to accommodate an even larger array of application.

The future work of this project also involves permeating our M-by-N and PRMI infrastructure through the component framework and GUI (Graphical User Interface) of our system. Another interesting feature we plan on adding are less restrictive distribution descriptions. Through this we can allow a process to specify, for instance, that it only requires a stride of two for its array and is not concerned with the size of the array it receives. This would make our redistribution system more powerful and usable. This feature may also be interesting in optimizing the system to various architectures. Finally, we plan to improve the performance of the system.

8. Acknowledgements

This work was funded by the Department of Energy Center for Component Technology for Terascale Simulation Software (CCTTSS) and by NSF ACI-0113829. We wish to acknowledge Scott Owens for discussions regarding and help with the array redistribution problem.

References

[1] R. Armstrong, D. Gannon, A. Geist, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, July 1999.

[2] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*, July 1998.

[3] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing fault- tolerance, visualization and steering of parallel applications. In *Environment and Tools for Parallel Scientific Computing Workshop*, Domaine de Faverges-de-la-Tour, Lyon, France, August 1996.

[4] Object Management Group, 2002. URL: http://www.omg.org.

[5] K. Keahey and D. Gannon, PARDIS: A Parallel Approach to CORBA, In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing* (best paper award), August 1997.

[6] K. Keahey, P. K. Fasel, and S. M. Mniszewski. PAWS: Collective invocations and data transfers. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computation*, July 2001.

[7] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and efficient group method invocation for parallel programming. Technical report, Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 2001.

[8] Steven G. Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, The University of Utah, August 1999.

[9] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, 1998.

[10] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. *Divorcing Language Dependencies from a Scientific Software Library*. In *Proceedings of the 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 12-14, 2001.

[11] Tom Epperly, Scott Kohn, and Gary Kumfert. *Component Technology for High-Performance Scientific Simulation Software*. Working Conference on "Software Architectures for Scientific Computing Applications", International Federation for Information Processing, Ottawa, Ontario, Canada, October 2-4, 2000.

[12] Kenneth H. Rosen. *Elementary Number Theory and Its Applications.* Addison-Wesley Publishing Company, 1984.

[13] I. Foster, N. Karonis, C. Kesselman, G. Koenig, and S. Tuecke. A secure communications infrastructure for high-performance distributed computing. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation*, July 1997.

[14] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115{128, Summer 1997.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multi-threading and communication. *Journal of Parallel and Distributed Computing*, 37 (1):70{82, 1996.