Practical Parallel Remote Method Invocation for the Babel Compiler

Kostadin Damevski Keming Zhang Steven Parker

Scientific Computing and Imaging Institute
University of Utah, Salt Lake City, Utah 84112, USA
{damevski,kzhang,sparker}@cs.utah.edu

Abstract

Parallel components are types of software components that contain Single Program Multiple Data (SPMD) parallel code and are used and defined by the Common Component Architecture (CCA) component model. Parallel Remote Method Invocation (PRMI) defines a communication paradigm between two parallel components of this kind. Within the CCA community, we define PRMI to include two parts: collective invocations and data redistribution. In this paper, we devise a way to build PRMI onto the Babel compiler, which is a central supporting technology of CCA. We perform this integration cleanly, by preserving Babel's design principles and allowing user choice in the wire protocol and parallel communication library. In addition to this, we define a novel set of synchronization options for PRMI that allow trading off synchronization for better performance while not endangering the accuracy of the result.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Design

Keywords IDL Compiler, Parallel Remote Method Invocation

1. Introduction

In recent years, component technology has been a successful methodology for large-scale commercial software development. Component technology encapsulates a set of frequently used functions into a component and makes the implementation transparent to the users. Application developers typically use a group of components, connecting them to create an executable application. While component software technology is an important and widely used tool in software development, commodity component implementations do not support the needs of high performance scientific computing: low overhead, scientific data types, and parallel programming. To remedy this, the CCA (Common Component Architecture) [1] group was formed among various universities and research institutions to add the functionality of components to existing scientific computing code while preserving the natural speed and efficiency that this code contains.

The CCA component model relies on a Scientific Interface Definition Language (SIDL) to define components, ports, as well as define the component model itself. A component specification is written in SIDL and compiled into glue code that is later compiled into an executable together with the user code. The prevalent way of compiling SIDL is by using the Babel compiler [10]. Babel has the capability of compiling SIDL to bindings for several popular programming languages (C++, Java, Python, Fortran77 and Fortran95), enabling coupling of components written in any of these languages. Babel has a large and growing user community and is an important technology behind the CCA component model. Recently, the compiler was upgraded to produce bindings for distributed computing through Remote Method Invocation (RMI). The Babel RMI bindings provide a way for components existing on separate computing resources to communicate with little or no help from the user. Babel's RMI also provides a general interface within the compiler to plug in any kind of wire protocol. The wire protocol library is invoked by the Babel generated code to produce behavior that the user is expecting.

Scientific computing is adopting more complex simulations that combine multiple physical models. These simulations apply several live simulation programs as dynamic boundary conditions in place of the traditional static boundary condition approaches. This new approach is common in biological cell modeling, climate modeling, fusion energy simulations and other domains. It produces results that have higher fidelity, but requires coupling of separate simulations. Oftentimes this coupling needs to bridge different scales and different data decompositing. Parallel Remote Method Invocation (PRMI) is the component software solution to this new set of problems in scientific computing simulations. In fact, we are aware of several applications that currently use Babel that could benefit from the addition of a this infrastructure.

Parallel programming 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 most commonly used in scientific codes is SPMD (Single Program Multiple Data). Parallel components that encapsulate scientific code often rely on MPI (Message Passing Interface) [11], PVM (Parallel Virtual Machine) [13] or other products that facilitates parallel programming. Allowing parallel components of this kind 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 and between a parallel and nonparallel component does require outside support. This is especially true when the number of processes differ between the caller and callee components. We use the term PRMI to describe this type of a method invocation. In order to achieve the full practical potential of PRMI we use M-by-N data redistribution. The data redistribution problem comes about when, in order to increase efficiency, data is subdivided among M 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 has to be decoded and mapped from the first component to the second component's specification. Because each component can require a different data distribution to a separate number of parallel tasks, this problem can get complicated. Also, since components can be connected at runtime, their distribution requirements are not known a priori.

The PRMI problem has had a few high level solutions within the CCA community [4]. Existing PRMI implementations use collective semantics for the invocation to involve all caller and callee processes regardless of whether their numbers match. A data redistribution mechanism is used to provide each process with the needed part of the domain. Many questions still remain of the synchronization guarantees that are provided by a collective invocation. Being overly asynchronous may get the wrong result while being completely synchronous and conservative means far slower execution times. The synchronization issue is further limited by different architectures that may not provide threads and parallel libraries (e.g. MPI) upon which PRMI is based may or may not be thread-safe.

Most PRMI implementations so far have not affected the larger CCA user community. We believe that building PRMI within the Babel compiler would have a large impact and would likely be welcomed by the larger HPC community. In this paper we contribute a way to add PRMI to the Babel compiler, while also giving the user choice whenever possible in the amount of synchronization. This paper presents an addition of the Babel compiler that: 1) supports PRMI and data redistribution, 2) can work with any communication protocol implemented for Babel, and 3) provides several synchronization options (constrained by the communication protocols in Babel and the parallel protocol used in the component).

We organize the discussion of our PRMI design as follows. Section 2 contains a discussion of the background knowledge behind PRMI and Babel. In Section 3 we present some of the related work, and in Section 4 we show a detailed view of our design and implementation of the PRMI in the Babel compiler. Section 5 contains an analysis of the synchronization options in PRMI. Finally, we finish with conclusions of the project in Section 6.

2. Background and Related Work

2.1 SIDL and Babel

The Babel compiler translates SIDL interface specifications into glue code for using these interfaces in several programming languages. The SIDL language is used to specify types in an implementable and platform independent form and is necessary in defining CCA components. It is able to define classes containing sets of implementable methods, multidimensional arrays, and an extensive set of basic types (e.g. integers, floats, strings, complex numbers, etc.). Classes can specify only methods and not variables. Multidimensional arrays can be declared in method signatures, but arrays of arrays are disallowed. Every argument in a method has three parts: a mode, a type and a name. The type and name are normal to most programming languages, while the mode is particular to IDLs. The mode specifies the flow of the argument when the method is called and can be one of three options: in, out or inout. Here is a class called UFO expressed in SIDL:

```
class UFO {
  void reportSighting(in int num_contacts,
                      out long ID);
}
```

The *reportSighting* method description above requires two parameters. The number of contacts should be a meaningful parameter to the method. However, the ID argument has pass-by-reference semantics. The method will place this argument in the buffer provided during its execution and return it to the calling code.

When compiled using the SIDL compiler, two corresponding parts are produced from the SIDL specification: the stub and the skeleton. These represent the "wiring" which needs to be in place for two components to interact. The stub is associated with the client code and the skeleton is coupled with the server implementation. The stub and skeleton code act as proxies for the method invocations from the consumer to the appropriate implementation routines of the producer. The book by Szyperski [14] gives an excellent overview of components and various commodity component models.

On input of one or more SIDL interfaces, Babel produces a set of files containing stub and skeleton code. To support distributed objects on any platform using any OS, the stub and skeleton are coupled with all of the necessary marshaling/unmarshaling and networking code. To use a Babel object in the same address space a proxy needs to be instantiated by using a *_create* method. This object's methods can subsequently be invoked on this proxy. A few additional user-level methods exist in Babel that were added to enable using an object in a separate address space (or RMI):

```
<T> _createRemote(in string url);
static <T> _connect(in string url);
string _getURL();
```

The *_createRemote* method is self-explanatory, it creates an object in a remote location specified by a URL and returns a proxy object in the local address space. The *_connect* method creates a proxy object for an already existing remote object when provided with a URL representing a listening object. *_getURL* returns this URL string representation of an instantiated object that can be used to create a proxy while also initializing the object to listen for remote method invocations. As in regular method invocations, the user calls methods on the proxy and receives a response. The user does nothing different to invoke methods on RMI proxies that will actually travel on the network to a remote object.

2.2 PRMI

PRMI occurs when a request is made for a method invocation on a parallel component or when a parallel component itself makes such a request. This request can have different semantics. For instance, the request can be made to all or from all of the processes on the parallel component or it can involve only one representative process. The return value can be discarded, reduced, or copied to each process participating in the invocation. The number of options regarding parallel invocations is significant and there is no single expected behavior. Due to this, PRMI implementations are reluctant to provide only one specific option in the design of their systems. On the other hand, systems do not want to instantiate too many possibilities and create a state of confusion. The most common approach is to provide two types of PRMI semantics: collective and independent. The collective involves all proxy (caller) and server (callee) processes in the invocation, while the independent behaves like regular non-parallel RMI and is only single process to process. The independent invocations are in fact just regular RMI in Babel that exists in the parallel case for situations where parallelism is utilized as a means of load distribution; if many invocations are expected to a particular component, multiple parallel processes can be used to distribute the load. It is also our belief that these two PRMI options cover most of the reasonable programming decisions in the design of parallel components. A specifier to each method in SIDL can be used to distinguish between the two types of calls. The specifier *collective* is placed at the start of the method signature in the SIDL, while the independent case is the default and no specifier is required. These two options are provided to the component programmer to use whichever semantic choice is more fitting. The ability to support data redistribution is limited to the collective case, as this is the only scenario under which it makes sense to redistribute data.

Collective methods are defined as a collaboration of multiple processes that represent one computation [9]. A method invocation between components of this type can be inferred as one that requires the involvement of each parallel process/thread on the caller component. Examples of such behavior include all cases where collaboration is required between the component's parallel processes in order to solve a task. By collaboration we mean that the parallel processes will subdivide the problem space among themselves and work independently or through some level of communication to solve the problem. Most classic parallel algorithms fit this level of programming.

The data redistribution problem itself has been discussed among scientific computing researchers for a substantial period of time and systems such as PAWS [3], CUMULVS [8] and others have been developed that solve this problem for the limited case of multidimensional arrays. All of these systems are based on a specific data transfer API (Application Programming Interface). This API is independent from the actual parallel method invocations. Systems like these have created a general solution to the data redistribution problem, although each of them 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 that also addresses parallel method invocation semantics. Some of the previous data redistribution work suggests that in order to achieve maximum flexibility in the design, a data redistribution component needs to be developed [9]. This redistribution component would communicate between two components that require a data distribution and perform this distribution for them. We recognize the added flexibility of this design; however we argue against it simply because of its added complexity.

Recently, several groups within the CCA have developed prototypes for PRMI [7, 5, 4]. The main similarity between the separate efforts was the semantics of the collective invocations and data redistribution. Also, the data redistribution representations used were very similar. On the other hand, projects differed in the way of selecting processes into a proxy and in the means of creating the collective glue code. The SCIRun2 [7] PRMI project used an in-house SIDL compiler, and DCA [5] used a stub generator leveraging the MPI library for communication. All previous PRMI designs did not explicitly address the amount of synchronization, which directly and significantly affects performance. Furthermore, as we mentioned earlier, the only practical approach to enable PRMI use in the wider CCA user community is to apply it to a base CCA technology like the Babel compiler.

Baude et al. proposed a set of collective interfaces such as multicast and gathercast to address collective communications in distributed components on a grid [2]. These collective interfaces together form a solution to a similar problem as ours, but this grid approach is not suitable for parallel computation with closely-coupled components.

3. Design

From a user standpoint, the SIDL compiler is an excellent place from which to provide PRMI. It enables the user to interact with the system in the setup of the parallel object and proxy while not overburdening him and providing automated behavior as the target method is invoked. Implementing PRMI at a lower wire protocol level, at a higher level as a component, or as a separate PRMI library may allow the user some additional flexibility, but often at a large cost. In the PRMI as library or component cases this results in a performance overhead due to the decoupling of the actual method invocation from the PRMI. For instance, a PRMI component would add an additional hop to all parallel method invocation, while PRMI implemented in a library may require that the user ensures the synchronization between the PRMI parts with the actual method invocation. This artificial way of synchronizing with the collective method usually requires the timely invocation of additional library routines. Implementing PRMI as a wire protocol requires large additional information that the protocol normally does not have access to. Also, a protocol PRMI would often confuse the user by not clearly delineating between RMI and PRMI.

We implemented a prototype implementation on top of a recent version of the Babel compiler. The PRMI Babel uses version 1.0.0 of the compiler and includes changes to both the runtime and code generation. We used the “SimpleProtocol” as our transport mechanism and we tested several simple examples.

An important aspect of collective invocations is the level of synchronization imposed on the user’s implementation by the underlying system. In order to provide a guarantee of collectiveness, systems often enforce a barrier on the collective invocation. This unarguably is necessary in certain cases to provide user-expected behavior, but at a very significant cost of execution speed, which is unacceptable for many high performance computing applications. We argue that at times when the wire protocol and other circumstances allow it, PRMI systems should extend unsynchronized behavior as much as possible. The main goal of the multiple kinds of PRMI synchronization presented in this paper is to loosen the synchronization reigns for a subset of the applications that regular implementation of PRMI inflict.

3.1 Parallel Remote Method Invocation

As in previous work in this area, our approach is to extend SIDL with a *collective* keyword. SIDL already has two other method keywords that we can leverage together with the *collective* keyword: *nonblocking* and *oneway*. We use combinations of these keywords in order to define different synchronization levels. The difference between these two keywords is that *nonblocking* provides a ticket mechanism to wait for a particular result (out arguments), while *oneway* methods allow no out arguments or synchronization and are fully asynchronous. We combine keywords in order to get three different locking behaviors: ”synchronous”, ”nonblocking asynchronous”, and ”oneway asynchronous”. A combination of *collective* and *oneway* keywords produces ”oneway asynchronous” calls, *collective* and *nonblocking* placed in front of a SIDL method produce ”nonblocking asynchronous” and in the case where only the *collective* keyword is specified we have just ”synchronous”. When Babel uses a wire protocol without threads, we conservatively allow only ”synchronous” PRMI. Similarly, to allow asynchronous PRMI we require a version of the parallel library (e.g. MPI, PVM) that is thread-safe. We give an overview of each of the different synchronization strategies:

• **synchronous**. These parallel invocations should work in all system environments so they are built to work without any thread support. Even if threads exist in the wire protocol, the parallel library may not be thread-safe so we synchronize conservatively in order to get the right behavior in all scenarios. The conservative synchronization style of these parallel invocation comes at a significant performance cost.

• **nonblocking asynchronous**. The standard nonblocking methods in Babel return immediately when invoked and each method returns a ticket that can be used to wait until the call finishes or to periodically check its status. We extend this existing ticket mechanism for Babel nonblocking calls to additionally control collective nonblocking calls. For parallel objects receiving invocations from multiple proxies, we provide a guarantee that two method calls coming from the same proxy (caller) will execute in order and will be non-overtaking on the callee.

• **oneway asynchronous**. Only ’in’ arguments are allowed so the only guarantee we provide for these calls is ordering and non-overtaking behavior of invocations coming from the same caller.

First, we take the time to explain the changes to the Babel runtime classes that we made in order to support PRMI. A list of the classes that we modified and added together with the added methods is given in Figure 1. These will likely only be useful to people already familiar with the inner workings of the Babel compiler. After survey-

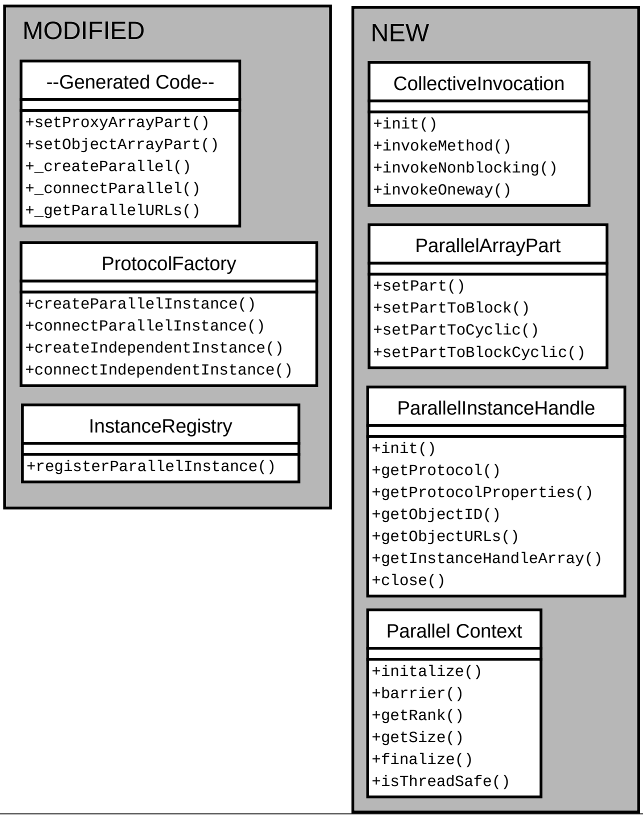

Figure 1. A list of the modified and new classes in Babel to enable PRMI.

ing these changes, we give the code for a simple example PRMI invocation between a parallel proxy and a parallel object. Our intention in presenting the example is to show the changes the user would make to their code if intending to use PRMI. Later, we discuss changes in the Babel code generated classes and changes to support data redistribution.

3.1.1 Changes to Babel Runtime

Babel provides two ways of connecting to remote components. One is using the *_createRemote* method that creates a remote object and connects to it. The other is using a *_connect* method to connect to an already existing remote object. A Babel Object Server (BOS) is needed on each node where a server object exists.

We propose two new methods to enable PRMI that are similar to the ones already used in Babel:

```
sidl.BaseClass
_createParallel(in array<string,1> nodeList);

sidl.BaseClass
_connectParallel(in array<string,1> objList);
```

We chose to return a *sidl.BaseClass* like the original Babel method and not a special proxy. Instead, the appropriate parallel method will be generated and connected in the proxy's function table. This approach is in line with the one Babel RMI used to build on top of in-process Babel invocations. A method to register a parallel server object is needed that is collective and synchronizes on a barrier ensuring that all of the processes call it:

```
collective void
InstanceRegistry::registerParallelInstance(
                  in sidl.BaseClass instance);
```

To implement this method we need to be able to call to the parallel library (e.g. MPI, PVM) to ensure that all cohorts have been initialized. We define a ParallelContext interface that offers methods that will wrap calls to barriers, initializers, etc. We present this interface fully later in this text. First we present our modification to the *ProtocolFactory* class that is used by Babel for establishing connections. We add four new methods to this class to support connecting to parallel components. These changes affect generated code only:

```
class ProtocolFactory {

...

static ParallelInstanceHandle
 createParallelInstance(
    in array<string,1> urls,
    in array<string,1> typeNames);
static ParallelInstanceHandle
 connectParallelInstance(
    in array<string,1> urls,
    in bool ar);

static sidl.rmi.InstanceHandle
 createIndependentInstance(
    in array<string,1> urls,
    in array<string,1> typeNames);

static sidl.rmi.InstanceHandle
 connectIndependentInstance(
    in array<string,1> urls,
    in bool ar);
}
```

The two calls added to *ProtocolFactory* create a new internal Babel class called a *ParallelInstanceHandle*. This new class is a collection of InstanceHandles used to make collective invocations. It is created by the upper two static methods, while the lower two "independent" calls only allocate a InstanceHandle used in order to make a serial RMI call. Both of these calls are necessary as PRMI is defined by the collective keyword on method granularity. Therefore, a proxy may have both parallel and serial methods requiring both a *InstanceHandle* and a *ParallelInstanceHandle*. A *ParallelInstanceHandle* is a container of several instance handles and is defined in SIDL as:

```
class ParallelInstanceHandle {
  bool init(
   in array<sidl.rmi.InstanceHandle,1> h);

  string getProtocol();
  string getObjectID();
  array<string,1> getObjectURLs();
  sidl.prmi.CollectiveInvocation
   createCollectiveInvocation(in string m);

  bool close();
}
```

The *ParallelInstanceHandle* class is used to create a *CollectiveInvocation*. This new class is a container of *Invocation* classes belonging to each of the instance handles that exist in a ParallelInstanceHandle and it is used to make a collective invocation from one caller component process to one or more of the callee component's processes. The *CollectiveInvocation* is created only from instance handles that this particular caller cohort (if the proxy and object are both parallel) needs to invoke. For more information of how this decision is made consult our previous work [7]. The structure of *CollectiveInvocation* class is:

```
class CollectiveInvocation
 implements-all sidl.io.Serializer {

  bool init(
   in array<sidl.rmi.Invocation,1> invs);

  sidl.rmi.Response invokeMethod();
  sidl.rmi.Ticket invokeNonblocking();
  void invokeOneWay();
}
```

Notice that the *CollectiveInvocation* class extends the *Serializer* interface allowing the packing and unpacking of attributes to the invocation as is the case with serial remote invocations in Babel. Also the class contains methods to invoke in different ways (*invokeMethod, invokeNonblocking, and invokeOneWay*). These methods implement the semantics of each particular call that we make available in our PRMI design. The appropriate invocation method is used in the generated code that corresponds to the SIDL method keywords.For consistency to Babel RMI we also add a parallel version of RMI's *_getURL* method:

```
collective array<string,1> _getParallelURLs();
```

The *ParallelContext* interface we mentioned before is used to wrap calls to the surrounding parallel library. Parallel libraries like MPI and PVM implement concrete classes inheriting from this *ParallelContext* interface:

```
interface ParallelContext {
  void initialize();
  void barrier();
  void getRank();
  void getSize();
  void finalize();
  bool isThreadSafe();
}

class MPIParallelContext
 extends ParallelContext {
}
```

In order to be able to disallow asynchronous PRMI when the protocol or environment disallows it, we provide special method that we expect to be implemented by each protocol and parallel library in Babel. In the *ParallelContext* interface, implementing classes should implement a *isThreadSafe* method that Babel can invoke in order to make decisions on synchronization. Similar method are also implemented by the wire protocol class indicating whether the protocol uses threads and whether asynchronous behavior is possible.

3.1.2 Example of PRMI Initialization Code

In this section, we show an example of a parallel client and server using our Babel PRMI implementation. This simple example invokes one collective method (called *m1*) and will better show how the changes we made can be used by a CCA programmer. The server part of the example follows:

```
ProtocolFactory pf;
if(!pf.addProtocol("simhandle",
                   "sidlx.rmi.SimHandle")) {
  std::cout << "Error in addProtocol\n";
  exit(1);
}
SimpleOrb echo = SimpleOrb::_create();
echo.requestPort(12121);
int tid = echo.run();
ServerRegistry::registerServer(echo);
ServerObj server = ServerObj::_create();

server._getParallelURLs();
```

The *_getParallelURLs* method will internally call *barrier* on the *ParallelContext*. In turn, the *ParallelContext* will *initialize* the parallel library before calling a barrier if that is necessary. It is understood that initialization of the parallel library will be performed before any user code's method is invoked and therefore it need not be called explicitly by the user. An alternative to the *_getParallelURLs* method is the following method:

```
InstanceRegistry::
 registerParallelInstance(server);
```

This method performs the same service as *_getParallelURLs* and registers an callee instance with the InstanceRegistry allowing it to be called by a distributed parallel proxy. Similarly, the *registerParallelInstance* method will wait on barrier internally. The example code for the client looks like this:

```
ProtocolFactory pf;
if(!pf.addProtocol("simhandle",
                   "sidlx.rmi.SimHandle")) {
  std::cout << "Error in addProtocol\n";
  exit(1);
}

ServerObj client =
 ServerObj::_connect(urlList);
client.m1();
```

The meaning of *_connect* is straightforward. As with the server, all of these methods are collective and we want to make sure that all caller processes successfully connect to the callees so we also synchronize on a barrier. An alternative way to connecting is to create a remote object:

```
ServerObj client =
 ServerObj::_create(nodeList);
client.m1();
```

_create results with a barrier on the callers and *ParallelContext::initialize* called on the callees.

3.1.3 Parallel Exceptions

The semantics of PRMI create a scenario where individual caller processes in a parallel component may not be directly aware of exceptions occuring in a callee process. The collective semantics require that every caller process be informed of any and all exceptions. Additionally, multiple heterogeneous exceptions can occur during the execution of a parallel component, necessitating an exception reporting system able to handle more than one exception at a time. We did not address parallel component exception in our prototype, but previous work in this area details several approaches in handling this problem [6, 12].

3.2 Data Redistribution

In past work on this problem several representations were used to explain domain decomposition and data redistribution. In our implementation, we choose a simple and effective design of providing a first, last, and stride data description strategy. This strategy prescribes that for each part of the global array space that a process has been assigned or has requested, the user specifies the first element, the last element and the stride of the elements in between. Multiple specifications are possible per each array dimension.

To specify the data representation in Babel we add a class to the Babel runtime called *ParallelArrayPart*. This class will be used to describe which portions of the global array we have or want. We provide a couple of built in methods for some common data distribution types such as block, cyclic, and blockcyclic. Here is the interface for the *ParallelArrayPart* class:

```
class ParallelArrayPart {
 setPart(int dim, int first,
         int last, int stride);

 setPartToBlock(int dim);
 setPartToCyclic(int dim);
 setPartToBlockCyclic(int dim);
}
```

We allow an arbitrary number of ArrayParts per dimension as long as they make sense together. If a dimension is not specified we assume that the whole dimension is given by the caller and in the converse case that none of it is needed by the callee. We add a couple of methods to distribute the ParallelArrayParts between the proxy/object process in order to calculate a data redistribution schedule:

```
void setProxyArrayPart(string array_arg_name,
             ParallelArrayPart [] parts);
void setObjectArrayPart(string array_arg_name,
             ParallelArrayPart [] parts);
```

The actual data is transferred when a method containing the distributed array is invoked. This method has to be collective and contain an additional *parallel* keyword placed in front of an array data type that we use to identify distributed arrays. Changes to the Babel code generator account for the rest of the modifications that were required in order to enable data redistribution in Babel. Since data redistribution is only possible for collective invocations, the methods designed for these invocations were extended with this additional functionality.

4. Synchronization Modes

We presented three synchronization options in the design of PRMI: oneway asynchronous, nonblocking asynchronous and synchronous. An example of the progression of the thread of execution for each type of method is given in Figure 2. In this section we'll try to explain our choice of synchronization trade-offs to enable three PRMI synchronization types and explain some of the high-level implementation parts that we needed in order to make these synchronization choices available.

When designing PRMI, it is crucial to provide the user with a consistent set of guarantees. In the case of most programming language execution order is something that a programmer relies on for program correctness. When programming in a middleware that is multithreaded and offers support for an operation such as PRMI, the invocation order given by a user should be preserved. In the past we have identified situations when some reordering may take place [6] with user awareness. However, in implementing general PRMI, we choose not to reorder any invocations and ensure that calls made on the proxy execute in order on the server. An object may receive calls from multiple proxies, and we see no reason why synchronization should preserve inter-proxy order. Therefore, our implementation makes no guarantees of ordering of invocations from more than one proxy.

To provide the single proxy invocation ordering some synchronization is necessary. Another choice we make is to synchronize on the object (callee) side and never on the caller side. Previous implementations have synchronized on the proxy (caller) side, but in a threaded environment this is not necessary. We eliminate any proxy synchronization for all but the conservative "collective" invocations which have to be deadlock free in the absence of threads.

The algorithm of maintaining ordering of a PRMI is based on three assumptions:

1. only the callees check the invocation order, callers do not check or wait.
2. each parallel stub (proxy) is identified by a UUID.
3. the invocation and/or data redistribution follow all-to-all pattern

When a PRMI *Ix* from a member caller arrives at a callee, *Ix* carries its rank and size. (*Ix*,rank) registers with the callee if it is not already registered. If (*Ix*,rank) is already registered, then *Ix* is put into a pending queue. The registration entry (*Ix*, rank) remains until all (*Ix*, rank) from each member caller has registered and responses have been all sent back. The pending *Ix*'s should now be all reset and treated as if they are just received.

This algorithm guarantees that the PRMI requests from the same parallel caller proxy do not overtake the previous PRMI, but allows

- each member caller to receive response and sends request without any delay from the caller side;
- the callee to process a request for a member caller if it has receives all required distribute data;
- the callee to sends a processed response back to the caller without waiting for the completion PRMI requests from other caller members.

5. Conclusions

This paper presents a method to build Parallel Remote Method Invocation (PRMI) into the SIDL compiler Babel. The method we propose is relatively simple to implement, preserves Babel's design, and presents three PRMI synchronization options to the user. We believe that if PRMI was provided by Babel in the manner that we propose, many new application paradigms would be able to leverage the CCA software engineering principles.

This design is also more generally applicable to any existing and future SIDL compilers. The Babel classes and implementational details may be somewhat different, but the general ideas are the same. Similarly, the synchronization paradigm we propose is efficient and flexible enough to be used by any PRMI tool.

6. Acknowledgments

The authors gratefully acknowledge support from NIH NCRR and the DOE ASCI and SciDAC programs.

References

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, 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 Computing*, 1999.

[2] F. Baude, D. Caromel, L. Henrio, and M. Morel. Collective interfaces for distributed components. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 599–610. IEEE Computer Society, 2007.

[3] 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.

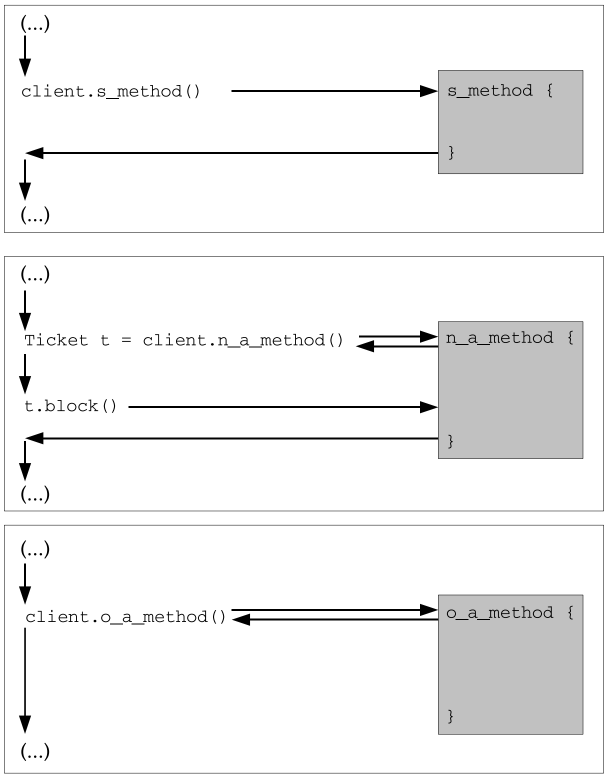

Figure 2. The synchronization options given for (top to bottom) synchronous (s_method), nonblocking asynchronous (n_a_method) and oneway asynchronous (o_a_method) parallel remote method invocation.

[4] F. Bertrand, R. Bramley, A. Sussman, D. E. Bernholdt, J. A. Kohl, J. W. Larson, and K. B. Damevski. Data redistribution and remote method invocation in parallel component architectures. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[5] F. Bertrand, Y. Yuan, K. Chiu, and R. Bramley. An approach to parallel mxn communication. *Special Issue of the International Journal of High Performance Computer Applications*, 19(4), 2005.

[6] K. Damevski and S. Parker. Imprecise exceptions in distributed parallel components. In *Proceedings of 10th International Euro-Par Conference (Euro-Par 2004 Parallel Processing)*, volume 3149 of *Lecture Notes in Computer Science*. Springer, 2004.

[7] K. Damevski and S. Parker. M x N data redistribution through parallel remote method invocation. *Special Issue of the International Journal of High Performance Computer Applications*, 19(4), 2005.

[8] 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.

[9] 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.

[10] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[12] C. Pérez, A. Ribes, and T. Priol. Handling exceptions between parallel objects. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 671–678. Springer, 2004.

[13] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

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