

PAWS: Collective Interactions and Data Transfers

Katarzyna Keahey *
Argonne National Laboratory
Argonne, IL 60439
keahey@mcs.anl.gov

Patricia Fasel and Susan Mniszewski
Los Alamos National Laboratory
Los Alamos, NM 87545
{pkf,smm}@lanl.gov

Abstract

In this paper we discuss problems and solutions pertaining to the interaction of components representing parallel applications. We introduce the notion of a collective port which is an extension of the Common Component Architecture (CCA) ports and allows collective components representing parallel applications to interact as one entity. We further describe a class of translation components, which translate between the distributed data format used by one parallel implementation to that used by another. A well known example of such components is the MxN component which translates between data distributed on M processors to data distributed on N processors. We describe its implementation in Parallel Application Work Space (PAWS), as well as the data structures PAWS uses to support it. We also present a mechanism allowing the framework to invoke this component on the programmer's behalf whenever such translation is necessary, freeing the programmer from treating collective component interactions as a special case. In doing that we introduce framework-based, user-defined distributed type casts. Finally, we discuss our initial experiments in building optimized complex translation components out of atomic functionalities.

1. Introduction

Most high performance scientific components or applications are implemented as parallel programs operating on physically or logically distributed data. We will call such components *collective components* since it is of advantage to interact collectively with all the processes participating in their computation. As we consider the interaction between such components, we have to focus on two major issues: (1) the definition of what exactly it means for two parallel components to interact, for example in terms of synchronization, and (2) how those components can most effi-

ciently exchange the distributed data they operate on. For many programmers the distributed data exchange problem becomes particularly complex as it is often composed of several different operations: the components may be using different data representations (eg. sparse versus dense), different data distributions (eg. striped versus checkerboard), be distributed over different sets of resources (MxN problem), or all of the above. In addition, the data translation may happen in place when both interacting components are using the same memory resources, remotely, when the components don't share any memory, or be a combination of the two.

Since defining collective component interactions and providing distributed data translation between them are a common feature, significant efforts have been expended to implement them efficiently. Many of those efforts were, and still are, undertaken by applications developers (see [5] for an example). Although usually highly optimized, these systems are often confined to a specific domain and data representation, thus prohibiting widespread, modular use, and interoperability between different projects. Several attempts have been made to develop generic frameworks solving this problem; [6, 9, 3, 7] have all addressed its aspects. Unfortunately, all of these solutions are limited to a set of applications that have fallen within the scope of experience of their developers, and therefore none of them have been fully successful in providing a solution that would be general across packages.

Several factors influence the difficulty of producing such a solution. First, data redistribution depends on data representation which is very often application-specific. Therefore developing a standardized solution for distributed data transfer depends on developing a standardized data representation. Further, different implementations rely on different assumptions about transfer (for example about timing, locking of data, and synchronization). Finally, the shape of abstractions in different systems depends on time and tolerance of different users.

The Common Component Architecture (CCA) group was formed with the objective to research and define the

*This work was completed at the Los Alamos National Laboratory as part of the PAWS project

needs of scientific high-performance application in the context of component technology [17]. Its members have a high degree of expertise in addressing these problems as evidenced by their work [11, 13, 2, 4, 15, 3, 10, 7]. The CCA effort is promising with respect to addressing the challenges outlined above as it has already introduced a standardized system of interactions [1] for non-collective components and is in the process of defining standardized representations for distributed data. This paper builds on those achievements, as well as on the experiences of the PAWS project [3], and extends the CCA interaction model (ports) to define minimal behavior required for collective components to usefully interact. This abstraction will allow the programmer to focus on high-level program design, rather than complexities of interaction, as collective components can be interacted with as one entity even though they are composed of a group of threads or processes. The usefulness and efficiency of similar abstractions has been discussed in [9, 10, 3]. This is a functionality not found in other existing standards of the day such as [12, 14] and represents a significant extension of these standards.

Since collective component interactions also involve distributed data transfer, we further discuss a class of *translation components* that translate between the distributed data format used by one parallel implementation, to that used by another. A well known example of such components is the *MxN component* which translates between data distributed on M processors to data distributed on N processors. We describe the PAWS distributed data representation, and use functionality abstracted from the PAWS system to build a library of CCA-style translation components. and show how it can be used in applications. In addition to the CCA interactive model, the *MxN component* assumes the PAWS data representation which can be used to capture the representation of any dense rectilinear data. We also present a mechanism allowing the framework to invoke this component on the programmer's behalf whenever such translation is necessary freeing the programmer from treating collective component interactions as a special case. In doing that we introduce user-defined distributed type casts. Finally, we discuss our initial experiments in building complex translation components out of atomic functionalities.

In summary, this paper describes the following results:

- it describes the customizable PAWS collective ports interactions designed to be compatible with CCA, and representing our proposal to the CCA forum for collective component interactions.
- describes a distributed data representation used in PAWS
- introduces and describes an *MxN* collective component, a part of a larger family of translation components
- describes an extensible framework functionality capable of handling *MxN* functionality as a component, rather than an integral part of the framework. In conjunction with other translation components, this capability allows us to do user-defined dynamic casting.

The rest of this paper is structured as follows. In the next section, we establish the terminology that we will be using throughout the rest of the paper, and describe the mechanisms of collective invocations, data structures, and translation components in PAWS. In the following section we give an example of how these mechanisms can be used in an application. Finally, we discuss operations on translation components. The last two sections conclude the paper and discuss our plans for future work.

2. Collective Invocations and Data Transfers

Collective components are a collaboration of multiple processes that logically represent one computation. In order to be both efficient and useful, an abstraction representing collective components must be able to leverage the existence of multiple processes, but at the same time allow the programmer to treat the collective component as one logical unit. This functionality is provided by *collective ports* through which the collective components interact. Collective ports are an extension to CCA ports described in [1], and are described in detail below. A very common special case of collective interaction is distributed data transfer from one collective component to another. This operation usually involves some sort of data translation, for example data redistribution, decimation, or formatting operations. We will call a component dedicated to translating distributed data between collective components, a *translation component*. The *MxN component* implements a very popular translation scheme translating data distributed on M processes to data distributed on N processes.

The sections below describe our experiences with combining different kinds of interactions taking place in a collective component scenario. Although it would be tempting to make data translation an easily-accessible framework service, we decided to make it a component instead. The reason for this is that parallel programs typically operate on many different data formats, and new data formats are designed every day, each with different translation performance trade-offs. It is therefore imperative that the translation components be easily modifiable, which is conveniently achieved by giving them the rank of an ordinary component. At the same time, it is important that they can be efficiently invoked by the *framework* as well as other components, which can be achieved by streamlining their interfaces and distributed data representation. Furthermore, the process of data translation is often in effect a compo-

	issues	policies
1	Which processes of a collective component are responsible for accepting the invocation?	a) all processes b) a proper subset (interactive subset)
2	When is the invocation accepted?	a) argument-based b) invocation-based
3	How does a component decide to accept a collective invocation?	a) per-process b) collectively by a designated process by majority voting
4	What synchronization guarantees can I expect?	a) blocking barrier b) non-blocking barrier c) none
5	What is guaranteed about the invocation order?	a) overtaking b) non-overtaking
6	How are the non-distributed arguments and return value delivered?	a) to designated process b) to every process in the interactive interface
7	Is the invocation on client's side blocking or not?	a) blocking b) non-blocking

Table 1. Policies customizing the functionality of a collective port as responses to programmer's issues; policies that PAWS is currently experimenting with are boldfaced.

sition of different translation operations. Since many combinations may be needed, and it is not practical to provide them all, it is necessary to incorporate a mechanism for efficiently combining translation components of atomic functionality into metacomponents. Given these considerations, our design is based on atomic, user-modifiable translation components that fit into the interactive framework we describe below.

The design has been implemented in PAWS; throughout our discussion we will be assuming the PAWS memory model [3], that is distributed memory. Furthermore when discussing collective objects we will assume that the sets of processes implementing them are disjoint.

2.1. Collective Invocations and Returns

Collective ports are associated with a set of processes each of which can be in a different state with respect to component interaction in the most general case. Extending the invocation metaphor to collective ports requires therefore the identification of states in which such groups of processes are capable of meaningfully responding to interactions. These states describe to programmers what guarantees from the framework they can expect in implementing components, and conversely, describe to the framework what functionality it needs to provide in order to be capable of interacting with specific components. Furthermore, it re-

quires from the framework implementing the collective port abstraction the ability to manage those states which may involve synchronization, conflict resolution between groups of overlapping processes, and providing non-overtaking invocations.

For example, the framework could guarantee to the component programmer that on entering the invocation to a collective port all the processes forming the interactive interface of a collective component will be in the same state (ready to execute the code associated with that invocation), and that all arguments to that invocation will be stable in the memory associated with the processes (if appropriate). This scheme was described in [10]. It provides a very high guarantee to the programmer, required for most SPMD-style components, but at a price: the framework performs synchronization on behalf of the programmer. Many collective components will have less strict synchronization requirements and won't want to pay this price. Therefore we decided to extend this scheme in our design by customizing the collective port by a set of policies to be implemented by the framework, and chosen by the programmer implementing the collective object. Each policy corresponds to a state enabling a certain flavor of interaction on a collective port. We summarized these policies in table 1.

These policies allow the programmer to customize the interactive characteristics of a collective object to maximize his or her goals and imply certain trade-offs. For example in

the case of SPMD applications it makes sense to deliver the invocation to every process taking part in the computation (1a). Other components, such as the MxN component, may find it more efficient to interact through a subset of M processes (1b). The next policy takes account of the fact that transferring distributed arguments can be a very expensive operation. It therefore sometimes makes sense to accept invocations only after all argument transfers have been completed (2a) rather than accept invocations as they come and wait for transfer completion. On the other hand such policy violates fairness, so in environments where quality of service is important policy (2b) may be preferred. In general, the question of how the decision to accept an invocation is made can have far-reaching consequences. If an invocation is accepted collectively (3b) it entails some sort of synchronization and therefore a performance penalty; acceptance on a per-process basis does not have this problem, but results in very weak synchronization guarantees to the component. Collective invocation acceptance also introduces a gradation of synchronization guarantees: if it may happen that one invocation begins before another has finished, but the sequence of invocations is guaranteed, the programmer can perform only a non-blocking barrier (4b); with tighter synchronization restrictions a blocking barrier will be possible (4a). Under certain conditions non-blocking invocations can get ahead of each other. In this case, extra functionality on the part of the framework is needed to prevent them from overtaking (5 a and b). Mode of delivery of distributed arguments is a trade-off between generality (6b) and efficiency (6a). Finally, whether an invocation blocks or not has long been a matter of policy.

A policy gives the programmers a set of assumptions to work with while implementing the functionality of a collective component, and allows them to customize that set to reflect the needs of specific components. Afterwards, a set of policies that a given collective component is compatible with becomes its characteristic. Not all combinations of policies are possible.

2.2. Data Representation and MxN Transfer

PAWS defines a flexible data representation, the `DataField`, that can be used to represent any distributed dense rectilinear data structure composed of arbitrary user-defined non-distributed elements. `DataField` is not designed as a data structure for the programmer's use; it's goal is to "shadow" data structures used by an independent package, that is to overlay on a representation used by that package and extract from it the information necessary to perform for example an MxN translation. To maximize compatibility and efficiency, the `DataField` can deal with different schemes of non-contiguous memory layout, both arbitrary (a list of contiguous data blocks) and structured (for exam-

ple strided), and dynamically sized arrays. The interface to `DataField` is shown below.

```
template<class DataType>
class DataField{
public:
    DataField();
    DataField(const std::string& dataFieldName,
              Paws::Domain& gDom,
              Paws::order ord = PAWS_ROW_MAJOR);
    DataField(const std::string& dataFieldName,
              Paws::Domain& gDom, Paws::Domain& lDom,
              DataField* userDataPtr,
              Paws::order ord = PAWS_ROW_MAJOR);

    layoutInfo& layoutInfo();
    actualDataBase* actualData();
    void addDataBlock(DataField* d, const Paws::Domain lDom);
    void update(Paws::Domain& gDom,
               std::vector<Paws::Domain> lDom,
               std::vector<void*> dptr);
};
```

`Paws::Domain` represents an n -dimensional array, or its fragment, as a vector of structured representation; for example a vector of tuples $\{first, last, stride, \dots\}$ for a strided representation. The global domain refers to the shape of the overall structure, and is partitioned into subdomains, or local domains, corresponding to contiguous memory blocks. This information is held by `layoutInfo` and used to compute efficient MxN transfer schedules. Complementary to the `layoutInfo` is the information about where the actual data is stored; `actualData` is a list of contiguous memory pointers corresponding to the domain information in `layoutInfo`.

Based on this data-access representation, PAWS implements a range of translation components. They include the MxN component, which provides efficient transfer of data distributed over M processes to data distributed over N processes, transforming the data into data of smaller or larger size by adjusting granularity (striding) or truncation, change of distribution template (eg. striped versus checkerboard distribution), change the data order from column major to row major (alias transpose), and flipping the data structure along selected dimensions. Implementing a translation component operating in a distributed environment involves computing a *schedule* determining which parts of data associated with a given process will be sent to specific processes on the destination side, performing those sends, and performing local operations on data if required.

Separating the layout information from actual data gives PAWS flexibility in manipulating data. For example, it can be used for efficiently buffering data of the same layout; as computation advances to the next copy in the buffer we just need to change the data pointers and since we can reuse the same layout the data translation schedules don't need to be recomputed. Another example of the flexibility of this scheme is when we need to "slide a window" over a large domain of data: as long as the dimensions of the window stay the same we can just advance the pointers to the data

fragment we want to access without recomputing schedules. This scheme can also be used in facilitating operations on ghost cells and similar cases.

In subsequent discussion we will use the MxN component as an example of a translation component. Using the PAWS MxN component involves the following steps: instantiating an appropriate translation component at composition time, connecting it to data structures so that it can extract information suitable for schedule computation, and invoking this component by the agent wishing to share the data. Shown below is the interface for the PAWS MxN component.

```
class MxN {
public:
    int translate(GlobalId<DataField> &local,
                 GlobalId<DataField> &remote);
};
```

GlobalId is a primitive equivalent of a global pointer. The MxN component is considered to be distributed over the sum of the processes of the collective components between which it translates and conforms to the functionality and policies outlined in the previous section.

Since all of our translation component are of atomic functionality, their interfaces are the same. PAWS allows superimposing translation components to form a metacomponent combining multiple functionalities, but its interface remains the same. Since the type of the data structure that emerges after these transformation is often different than the type that went in (eg. a smaller fixed array), the PAWS MxN object essentially performs a user-defined data cast.

Translation components can be invoked directly by a client component wishing to send data to another collective component (or vice versa), however the semantics of such invocations are limited. In order to expand them PAWS is considering using an abstraction similar to the CORBA push/pull channel [12] combined with the functionality of the translation component. This would extend the functionality of data transfers to include such useful functions as buffering, backup and replay, dealing with multiple clients and providers, etc.

2.3. Integration of Data Transfer and Invocation

So far we have discussed collective invocations and transfer of distributed data separately. Those two operations are often related as when a collective invocation operates on distributed data. As with data transfer, the programmer has to indicate at composition time which translation components should be used, but unlike in the case of data transfer the client of that invocation is not a user-level component, but the framework.

One way to make such invocation is by binding to the MxN component and invoking data translation explicitly as shown below:

```
// A, B are local arrays,
// A', B' are destination side handles
lock(A', B');
MxN.translate(A, A');
MxN.translate(B, B');
MyComponent.foo(A', B');
unlock(A', B');
```

The problem with this approach is that it deprives the programmer of the convenience that should be provided by a useful abstraction; not only do the transfers need to be done “by hand”, but it also requires the programmer to lock the access to the remote object in order to avoid race conditions between two competing clients. In addition, this approach prevents the framework from performing optimizations based on the knowledge that A' and B' are to be used as arguments to a function. Finally, this approach requires the programmer to establish five collective interactions (two with the MxN component and three with MyComponent) instead of just one.

In order to fix this problem PAWS enables the framework to connect to, and use the services of a collective translation component. This operation requires the following steps: at composition time the programmer has to provide a translation component (in other words: the programmer needs to define a cast); then at run time that component will be invoked by the framework when appropriate. At invocation time, the programmer needs only to make the following invocation:

```
MyComponent.foo(A, B);
```

The framework calls the translation unit however many times is necessary on the user's behalf, performs all the locking that is necessary, and is given the opportunity to optimize the interaction.

The framework's ability to perform the cast for the programmer without introducing inefficiencies relies on two things: the existence of generalized data structures and a streamlined interface (only a “translate” method is needed). In the current implementation of PAWS, we maintain the generality of this solution by ensuring that the programmer's data structure has a manually implemented cast to PAWS DataField which is used before control is transferred to the framework. Currently this cast is implemented as part of “porting” a package to work with PAWS. In the future, such cast will be generated automatically by the Scientific Interface Definition Language (SIDL) [15] compiler, based on the SIDL CCA data object representation, along with other stub codes. We are also intending to work with the SIDL group and the CCA Data Object group on ways of extending this user-defined cast mechanism to explore more complex translations and type matches, such as for example from sparse representations to dense, and on ways of enabling the user to safely specify casts, and the compiler to match them. Although the need for streamlined

interface can be obviated by using a mechanism similar to CORBA Dynamic Invocation Interface (DII), this approach may have adverse performance consequences and was therefore rejected.

3. Application Use of PAWS

The example below demonstrates the use of PAWS in an application developed at the Los Alamos National Laboratory (LANL) and consisting of a coupled model of water resources that places a river basin in its global context, the testbed being the Rio Grande from Colorado to Texas [16]. The coupled model is composed of four interacting components, but only two are being considered here. The first is a Regional Atmospheric Model System (RAMS) driven by actual global climate data applying a finite difference solution of the Navier-Stokes equation to predict potential temperature, mixing ratio of water, air pressure and horizontal and vertical components of wind. The second is a land surface hydrology model (LAHDS) using meteorological data, particularly precipitation, temperature, wind speed, short and long wave radiation, and air pressure, supplied by RAMS. The LAHDS model partitions precipitation into evaporation, transpiration, soil water storage, surface runoff and subsurface recharge.

There are communication problems between the two models that up to now were forcing the programmer to communicate through the writing and reading of files. In addition to the MxN problem encountered by most applications exchanging distributed data, it turns out that the scale and size of the data structures used are different, with RAMS working on a finer grid (`rams_x > ladhs_x`, etc.). Furthermore, the data structures are “flipped” along the second dimension as they get exchanged. Using PAWS user-defined casts solves these problems by performing a stride-based reduction of the RAMS data structure (with stride greater than one), applying a dimensional flip and performing the MxN transfer. These operations are specified at composition time (in a script) and then the three operations are applied all at once using an optimized module. The simplified code fragments below illustrate how the applications interact.

```
//RAMS main program fragment
// create RAMS data
RAMS_data* snow_data, rain_data;
allocate_and_init(&snow_data, &rain_data);

// Create PAWS structures
Paws::Index gDim0(rams_x), gDim1(rams_y), gDim2(rams_z);
Paws::Domain ramsDomain(gDim0, gDim1, gDim2);
// in a similar way create local domains: snowLD, rainLD
...
// create data fields
CCA::DataField snow('\snow'', ramsDomain,
                    PAWS_ROW_MAJOR);
CCA::DataField rain('\rain'', ramsDomain,
```

```
                    PAWS_ROW_MAJOR);
// Fill in the pointers to the local blocks of storage
snow.addDataBlock(ptr(snow_data), snowLD);
rain.addDataBlock(ptr(rain_data), rainLD);

// call the lahds hydrology model
lahds.compute(snow, rain);
```

In this code fragment, corresponding to the RAMS client, the programmer first creates data in a representation specific to the package. Then, a shadow PAWS representation of the data is created and instantiated with the actual data. Finally, a LADHS application proxy method operating on that data is invoked. The proxy method implements the logistics of sending the invocation to the LADHS application from all the processes involved in RAMS computation. At present such proxy method is written by the programmer adjusting a given package to work with PAWS. In the future these stubs, as well as the code creating the PAWS representation (moved inside the proxy) will be generated by a CCA-compliant compiler.

```
//LADHS main program fragment
// Create the preferred initial distribution
// description used by LADHS and allocate space
// for the programmer's data as part of component
// registration procedure
LADHS_data* snow_data, rain_data;
allocate(&snow_data, &rain_data);
Paws::Index gDim0(ladhs_x), gDim1(ladhs_y), gDim2(ladhs_z);
Paws::Domain ladhsDomain(gDim0, gDim1, gDim2);
// create local domains: snowLD, rainLD
...
CCA::DataField snow('\snow'', ladhsDomain,
                    PAWS_ROW_MAJOR);
CCA::DataField rain('\rain'', ladhsDomain,
                    PAWS_ROW_MAJOR);
// Fill in the pointers to the local blocks of storage
snow.addDataBlock(ptr(snow_data), snowLD);
rain.addDataBlock(ptr(rain_data), rainLD);
// Process requests
PAWS.processEvents();
```

In the second code fragment we begin by creating a PAWS data representation for every provides port method argument based on allocated user's structures; in the current implementation it is required as part of the “component registration” process, although the programmer is allowed to update the distribution information later. After registration is completed, the system goes into a loop polling for events on the provides port. The “compute” method implemented by LADHS is called from there, again through a stub. Note that neither program invoked the translation components explicitly; this invocation is performed implicitly by the framework.

4. Translation Components

As we have seen, when translating data many different operations can be applied resulting in the need for a set of translation components. While fully modifiable within a specified interface, these operations can be invoked by the

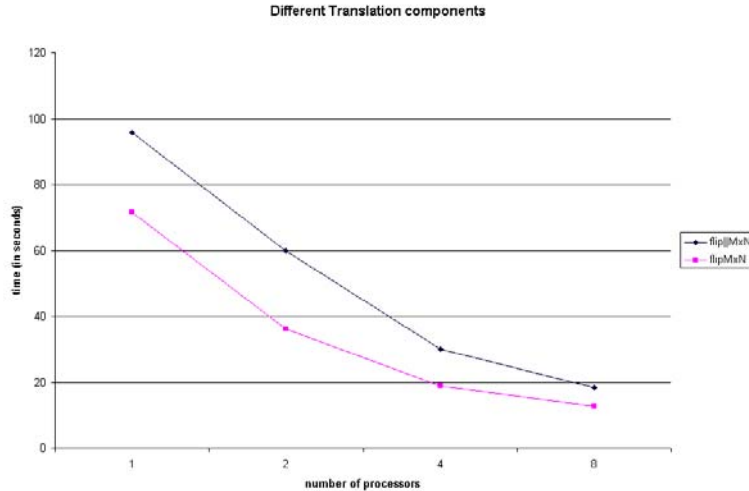


Figure 1. Comparison of dimensional flip and MxN performed at once and one after the other.

framework and extend its functionality. The relationship of these components to the framework is in practice the same as the relationship between for example the C standard library and the C language; while not a part of the framework itself they will be common enough, so that every framework will have to provide a reasonable library of them. We called these components *translation components*.

Sometimes interactions between components will require performing several of these operations on one piece of data. In that case, inefficiencies could arise out of the fact that it is generally faster to perform several operations at once rather than several operations one after another. The results shown in Figure 1 confirm this statement; it shows a comparison of those two approaches: a dimensional flip followed by an MxN translation, and both performed together, from the application described in previous section, with the latter being more efficient.

In this case it would be tempting to extend the functionality (and interface!) of the MxN component to account for this situation. However this approach would create two kinds of difficulty: a non-standardized interface would make it inefficient for the framework to call it, and the programmer would have a hard time finding and setting up a translation component answering to his or her needs among the multitude of combinations that can arise. Furthermore, given the large number of combinations, it is possible that some of them would simply not be implemented. We therefore prefer an approach in which every translation component is dedicated to one kind of functionality; if more than one translation operation is needed a translation meta-component is constructed, and later, optimized. That such

optimizations can be performed by a rule-based compiler has been shown by [8]. The added benefit of this approach is that even if a component performing several translation at once is not available, the programmer can still prototype the application using a composition of components, and later optimize it by implementing an “all at once” component. In other words, given a rich enough library of components such optimization could be performed automatically, but in the worst case the programmer has the freedom to do such hand-optimizations as seem suitable, and has a ready solution for prototyping.

5. Future Work

Our future work will focus on perfecting the collective invocation mechanism described here, interacting with the CCA group to bring PAWS up to date with the CCA standard, and influence that standard, and on optimizations of building complex translation components. Working on the collective invocation mechanism we primarily intend to explore and report on all the trade-offs involved in using different invocation policies, putting special emphasis on the performance aspect of that evaluation. Our interaction with CCA will focus on two areas: helping to define the functionality of collective ports and distributed data translation components, and also ways of exploring SIDL and the CCA Data Object research to automate and bring-up to standard converting user’s data to the CCA data representation. Finally, for the last topic on our agenda, we would like to work on extending our library of translation components, as well as experiment with optimizing compilation techniques

replacing a translation metacomponent with a component combining the functionality of its constituents. In addition to these topics, we want to explore the possibility of application of the CORBA channel functionality to distributed data transfer.

6. Conclusions

This paper described CCA-style abstractions and their prototype as implemented in PAWS. We described the abstraction of a collective port, an extension of the CCA port functionality supporting interactions with parallel programs, and introduced mechanisms through which it can be customized. We demonstrated how this abstraction can be used to interact with an MxN, or any other data translation mechanism, at argument translation time, thereby supporting user-defined type casting. We illustrated our discussion with a PAWS application. Finally, we outlined our experimentation in the process of designing a modular approach to constructing complex translation components.

The notion of a customizable collective port is a new abstraction not currently supported by the standards of the day. It strikes a balance between allowing the programmer to treat a relatively complex interaction as one concept and allowing him or her a reasonable degree of control over efficiency. Furthermore, incorporating the notion of atomic, programmer-modifiable translation components into the design and developing methods of optimizing them again tries to combine what is convenient and flexible with the efficiency requirements of high-performance computing.

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 Computation*, 1999.
- [2] R. C. Armstrong and A. Chung. Poet (parallel object-oriented environment and toolkit) and frameworks for scientific distributed computing. In *Hawaii International Conf. on System Sci.*, 1997.
- [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.
- [4] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of HPDC*, August 2000.
- [5] A partial list of available coupling software. http://www.atmos.ucla.edu/drummond/AGCM_infra/csw.html, 1999.
- [6] I. Foster, D. Kohr, R. Krishnaiyer, and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF via the Message Passing Interface. In *Supercomputing '96 Proceedings*, November 1996.
- [7] A. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing Fault Tolerance, Visualization and Steering of Parallel Applications. *The International Journal of Supercomputer Applications and High Performance Computing*, (11):224–235, 1997.
- [8] S. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of 2nd Conference on Domain Specific Languages*, pages "39–533", October 1999.
- [9] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation*, pages 31–39, August 1997.
- [10] K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-Level Parallel Distributed Computation. In *Supercomputing '97 Proceedings*, November 1997.
- [11] Mathematics and A. N. L. Computer Science Division.
- [12] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.0.* OMG Document, June 1995.
- [13] S. Parker, D. Weinstein, and C. Johnson. *The SCIRun Computational Steering Software System*, chapter Modern Software Tools in Scientific Computing. Birkhauser Press, 1997.
- [14] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.
- [15] B. Smolinski, S. Kohn, N. Elliott, and N. Dykman. Language interoperability for high-performance parallel scientific components. In *International Symposium on Object-Oriented Parallel Environments (ISOPE)*, December 1999.
- [16] E. P. Springer, C. L. Winter, and J. E. Bossert. Water resources simulation in the rio grande basin using coupled models. preparing for the 21st century. In *Proceedings of the 26th Annual Water Resources Planning and Management Conference*, pages "39–533", June 1999.
- [17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.