XChange: Coupling Parallel Applications in a Dynamic Environment

Hasan Abbasi, Matthew Wolf, Karsten Schwan, Greg Eisenhauer, Andrew Hilton
College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332–0250
E-mail: {habbasi,mwolf,schwan,eisen,adhilton}@cc.gatech.edu

Abstract

Modern computational science applications are becoming increasingly multi-disciplinary, involving widely distributed research teams and their underlying computational platforms. A common problem for the grid applications used in these environments is the necessity to couple multiple, parallel subsystems, with examples ranging from data exchanges between cooperating, linked parallel programs, to concurrent data streaming to distributed storage engines. This paper presents the $XChange_{mxn}$ middleware infrastructure for coupling componentized distributed applications. $XChange_{mxn}$ implements the basic functionality of well-known services like the CCA Forum's MxN project, by providing efficient data redistribution across parallel application components. Beyond such basic functionality, however, $XChange_{mxn}$ also addresses two of the problems faced by wide area scientific collaborations, which are (1) the need to deal with dynamic application/component behaviors, such as dynamic arrivals and departures due to the availability of additional resources, and (2) the need to 'match' data formats across disparate application components and research teams. In response to these needs, $XChange_{mxn}$ uses an anonymous publish/subscribe model for linking interacting components, and the data being exchanged is dynamically specialized and transformed to match end point requirements. The pub/sub paradigm makes it easy to deal with dynamic component arrivals and departures. Dynamic data transformation enables the 'in flight' correction of data or needs mismatches for cooperating components. This paper describes the design and implementation of $XChange_{mxn}$, and it evaluates its implementation compared to those of less flexible transports like MPI. It also highlights the utility of $XChange_{mxn}$'s 'in flight' data specialization, by applying it to the SmartPointer parallel data visualization environment developed at our institution. Interestingly, using $XChange_{mxn}$ did not significantly affect performance but led to a reduction in the size of the code base.

1. Introduction

Computational science is a distributed wide-area enterprise, requiring continuous interactions across geographically separated scientific data sources, across the computational simulations that use their data, distributed data repositories, and the remote output devices and portals to interact with ongoing computations. With the increasing size of simulation data, the demands on network bandwidth and storage capacity required by such applications are growing to the point where it is becoming a challenge to efficiently transfer and replicate these large data sets to remote nodes for analysis [5]. A particularly challenging application is the distributed Terrascale Supernova Initiative conducted by the Department of Energy (DOE) and university researchers in the United States [25], where the real-time interaction of distributed components and team members requires the sharing of data at Gbps rates across heterogeneous wide area networks. Similar characteristics may be observed for applications like earth simulations [27] or climate modeling [18], water contamination [22], and satellite data processing systems [9], which can easily consume the computational resources of multiple, grid-linked supercomputers.

One defining attribute of the large, multi-disciplinary applications described above is that they typically consist of multiple, parallel program components [2, 7], often separately maintained and improved by different research groups (e.g., chemical vs. physical models in atmospheric simulation [18]). The resulting component interactions impose interesting new constraints on the mechanisms used for data exchanges:

- **Diverse data formats.** In contrast to single parallel programs spread across distributed computing platforms, components are designed to be useful for many different scientific investigations. Hence, there will likely be mismatches between the different data representations used by communicating components. For example, a set of components implementing a parallel molecular dynamics application may distribute data according to its spatial positions, but a visualization component used by a research team may require data to be distributed according to atomic indices, or in a coupled atmospheric simulation the ocean model may provide data in a format different from that required by the climate model. A new role for a data exchange mechanisms, therefore, is to offer efficient means of data translation or transformation for interacting components.

- **Dynamic behaviors.** When constructing multi-disciplinary application codes, individual investigators cannot be assumed to know all details about the data formats or layouts used by individual components. Instead, such knowledge is captured by explicit metadata associated with components. Further, just like in commodity 'plug and play' applications where components arrive and depart at runtime, the meta-information itself is acquired dynamically, for the specific component instances used by a certain large-scale application.

This paper describes the $XChange_{mxn}$ data exchange mechanism for efficiently coupling parallel program components. $XChange_{mxn}$ extends earlier work on M-by-N transports by implementing entirely dynamic component interactions, permitting components to establish linkages whenever or wherever necessary on heterogeneous distributed computing infrastructures. $XChange_{mxn}$ implements the following dynamic functionality:

1. **Parallel M-by-N transport.** $XChange_{mxn}$ uses a simple, global description of distributed data exchanges, thereby accommodating multiple ways of distributing data across varying numbers of parallel processing elements.
2. **Data differentiation.** To exchange only the data needed by cooperating components, components can dynamically select the actual data elements to be exchanged from the potential data exchanges. This is attained by permitting applications to insert dynamically generated data filters into communication streams.
3. **Data transformation.** Application components store and represent data in formats best suited to the actual physical models under consideration. Potential mismatches in representation require the transformation of data during exchange. Such transformations cannot be expressed statically (at compile time) because of the need to support evolving data formats. $XChange_{mxn}$ provides functionality that allows applications to dynamically provide custom methods that transform an incoming data stream to the format or representation needed by each component.

Finally, $XChange_{mxn}$ provides an intuitive interface for (1) specifying global data distribution, and (2) creating and deploying transformation functions and source side filtering capabilities.

$XChange_{mxn}$ is part of the larger $XChange$ effort to create resource-aware middleware services for scientific collaboration. A future version of $XChange_{mxn}$, therefore, will be able to dynamically adjust the data exchanges being undertaken to the resources available on the underlying execution platforms, initially focused on network-awareness.

This paper evaluates the capabilities of $XChange_{mxn}$ using the two application scenarios described in Section 2. The SmartPointer framework is an example of the visualization application, and we compare its performance with $XChange_{mxn}$ with its performance as native implementation with the ECho middleware on which $XChange_{mxn}$ is based. Interactions between coupled parallel components are approximated with a simple matrix transpose benchmark, where we compare the performance attained by dynamic couplings realized with $XChange_{mxn}$ with the compile-time couplings established with MPI. Evaluation results are quite interesting. First, contrary to popular conception, the dynamic coupling methods used in $XChange_{mxn}$ do not lead to much performance degradation compared to static methods like MPI. In fact, the worst case additional overhead experienced by $XChange_{mxn}$ is only 15% larger than that attained by the static MPI implementation. Second, the implementation of $XChange_{mxn}$ imposes almost no additional overheads on parallel data transport compared with a native implementation using the underlying ECho middleware. Third, $XChange_{mxn}$ leads to substantial reductions in code size compared to native ECho or MPI solutions due to the simplicity of the $XChange_{mxn}$ application interface.

In the remainder of this paper, we first present relevant application scenarios in Section 2. Section 3 describes in detail how $XChange_{mxn}$ is implemented, with Section 4 describing the API it exposes to applications. The framework is evaluated in Section 5, and other related research is presented in Section 6. Conclusions and future research appear last.

2. Application Scenarios

Two classes of data exchanges between program components have driven our work on $XChange_{mxn}$: (1) remote data visualization or storage, implying 1-by-M or N-by-1 data exchanges, and (2) full M-by-N data exchanges between two explicitly parallelized program components. Each of these application classes is described in some detail below.

2.1. Data Visualization

When visualizing the data produced by parallel simulations, data is collected from multiple parallel program elements to some machine responsible for its display. $XChange_{mxn}$ can reduce the complexity of building visualization servers because of its built-in ability to deal with concurrent data transports. In addition, $XChange_{mxn}$ can deal with (1) the desire of end users to only visualize some of the output data (i.e., data of current interest) and (2) the different data representations used by visualization packages. Sample packages studied in our work include the CUMULVS [17] and SmartPointer [29] systems.

We illustrate the importance of data selection in conjunction with its exchange with Georgia Tech's SmartPointer system [29], which can be divided into three different elements, illustrated in Figure 1:

- **DataSource.** SmartPointer is designed to support parallel molecular dynamics applications. Its inputs, therefore, are received from a parallel code, resulting in an mx1 data exchange. This paper's experimentation, however, uses precomputed data emitted by a single data source, to remove potential variability in $XChange_{mxn}$ performance due to MD performance behavior.
- **BondServer.** This stage in the processing pipeline processes the atom locations and outputs a list of all bonds between atoms. The processing time of the application is $O(n^2)$, and the space bounds are $O(m^2)$, where n is the total number of atoms in the system and m is the total number of atoms close enough to have a bond present ($n >= m$). The BondServer component is implemented as a cohort of parallel processes that jointly implement its functionality.
- **Client.** The client is a component implemented by a parallel cohort of processes that receive bond information from the BondServer and either display it (if it is a visualization client) or store it (if it is a file client). The client processes do not perform much processing other than discarding unneeded data. For example, a client may partition all atoms of the MD simulation into those above a specific cutting plane or below, and implement each of those as a separate process. Alternatively, each process may filter out all atoms of a specific type.

SmartPointer exemplifies a general class applications that provide real-time visualization support for parallel programs [15, 24]. Data is generated by some parallelized program component, processed by additional parallel components, and then visualized using diverse displays or display packages. Efficient component coupling requires efficient support for parallel data transport and for data selection, so that particular components receive the data items they require. As currently implemented, SmartPointer does not require support for data transformation, but can benefit from it when receiving data from alternative data sources.

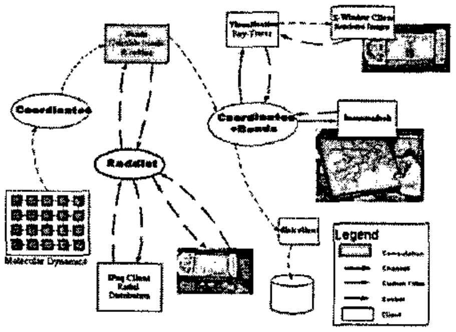


Figure 1. SmartPointer overview

2.2. Coupled Simulations

Consider the case of an atmospheric modeling application that simulates both chemical and physical processes in the atmosphere. In realistic deployments, the two coupled simulations each run on different hardware platforms and will use different levels of parallelism in order to match each others' speeds. Data exchanges between both simulations are bi-directional, and data representations are likely to differ across these components, requiring data selection and/or transformation along with exchanges. Another interesting aspect of this scenario is that differences in the speeds of the two components (based on their levels of parallelization and their mappings to machines) imply the need for flexibility concerning 'where' such transformation actions are carried out. If chemistry is the bottleneck, then for data sent from physics to chemistry, data transformation should be performed on the sender side, whereas for data sent from chemistry to physics, transformations should be done at the receiver. For $XChange_{mxn}$, this means that applications should be offered runtime choices on 'where' transformations are performed. $XChange_{mxn}$ provides these choices, where transformation and filtering functions can be run at the data source or the data consumer. Such functionality can improve the performance of coupled simulations by reducing the impact of computational and processing power mismatches.

3. Implementation

The implementation of $XChange_{mxn}$ avoids the need for prior knowledge in components about their runtime interaction. That is, at compile-time, components need only include the $XChange_{mxn}$ library with their code, but their decisions about which components they interact with, when such interactions occur, and how data exchanges and transformations are performed are made at runtime.

To accommodate dynamic component coupling, we employ the publish/subscribe model of component interaction, using the ECho [14] realization of pub/sub as an implementation basis. Specifically, $XChange_{mxn}$ treats data transfers as events that are published by data sources to event channels and then sent to data consumers who subscribe to these channels. 'Handlers' can apply operations to events when they are sent and when they are received. In fact, such handlers are the way in which $XChange_{mxn}$ extends the basic pub/sub model to implement data distribution and transformations. That is, each event has associated meta-information that describes the layout of the data it contains and the way in which such data is to be distributed to subscribers. We next describe how data distributions are specified and carried out.

To specify the global distribution of a data packet [11, 13, 17, 21], $XChange_{mxn}$ represents this distribution as a quintuple $\{start, end, stride, length\}$. *Start* is the beginning coordinate of the data block, *end* is the final coordinate of the block, *stride* is the length of the gaps within the block and *length* is the size of each subsection of the block. $XChange_{mxn}$ computes the actual data distribution to be carried out using an extended version of the redistribution calculation algorithm presented in [11], to handle variable- in addition to fixed-size blocks. Handler code implements the distribution actions computed by this algorithm.

There are several important attributes that distinguish $XChange_{mxn}$ from other implementations of mxn functionality:

1. Component independence. This property states that each component need only know about its own data distribution in order to successfully exchange data with another component. No prior information about the other component and/or its data distribution is required for successfully exchanging data between both. The implementation of $XChange_{mxn}$ and of the underlying pub/sub system are responsible for forming the appropriate network connections needed for efficient one-to-one data transmissions across the parallel processes (and the machines on which they run) that implement each component.

2. Subscriber-driven data differentiation and filtering. To realize a specific inter-component data re-distribution, $XChange_{mxn}$ combines the provider's distribution description with the consumers' description, whenever a subscription is initiated. To efficiently implement the combined distribution actions, $XChange_{mxn}$ leverages the ECho's ability to dynamically deploy handlers 'into' the address spaces of data providers, using dynamic code generation techniques. This facility permits the data provider to publish data without considering subscriber needs, but the actual data distribution and transformations being carried out in the data provider's address space do take into account those needs, thereby avoiding the expensive data copying or re-distribution actions required by concentrator- or overlay network-based realizations of mxn functionality.

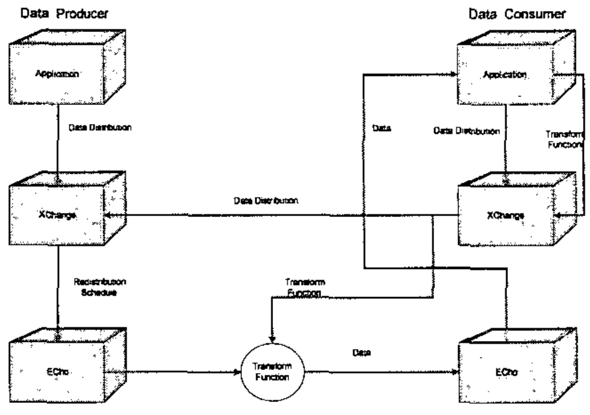


Figure 2. $XChange_{mxn}$ **overview**

Figure 2 describes the flow of data and parameters in the $XChange_{mxn}$ framework. Once a data producer application registers its data and data distribution with $XChange_{mxn}$, it then uses the $XChange_{mxn}$ api to submit data for transmission. Such transmissions typically trigger the execution of handlers that operate on the data, followed by the distribution of data via the underlying pub/sub infrastructure. As stated earlier, handlers are created dynamically, based on the additional distribution needs registered by data subscribers. Such subscribers can also provide transformation functions to convert data to correct format in which they wish to receive it and/or to perform data filtering. In our current implementation, this implies (1) that the subscriber sends the distribution to the data provider (using parameter transmission mechanisms provided by the underlying pub/sub infrastructure), (2) that a redistribution schedule is calculated and installed at the provider, (3) that in addition, a transformation function is installed, if desired, whereupon (4) all future events submitted by the provider are distributed and transformed for the subscriber as per its instructions. Finally, by integrating these actions into the implementation of the pub/sub system, all unnecessary data copying is avoided. This is demonstrated with performance measurements in Section 5.

4. Usage

```
server = initService(process_size, "test",
                    process_id, SERVER);
registerDistribution(server, no_tiles,
    distrib, sizeof(struct data));
registerData(server, data_field_list,
    data_format_list,test_data);
/*begin loop*/
/*process data*/
sendData(server);
updateDistribution(server, new_tiles,
                  new_distrib);
    /*end loop*/
```

Figure 3. *Producer API. The producer first registers the data and its distribution and then publishes the data at the end of each timestep.*

```
client = initService(process_size, "test",
    process_id, CLIENT);
registerDistribution(client, no_tiles,
    distrib, sizeof(struct data));
registerRcvData(client, data_field_list,
     data_format_list, filter,
     (unsigned char*)test_data);
addTransform(client, transform);
/*begin loop*/
rcvData(client);
/*process data*/
updateDistribution(client, new_tiles,
      new_distrib);
/*end loop*/
```

Figure 4. *Consumer API. The consumer registers data similar to the server but can also specify an additional transform function.*

$XChange_{mxn}$ exposes a simple API to the applications, involving data registration and description. Figure 3 shows the steps required by the data producer. The producer application initializes $XChange_{mxn}$ with the **initService** calls allowing other applications to dynamically discover the producer. It then registers the data and its distribution using the **registerData** and **registerDistribution** calls. The distribution is described as an array of **distStride** structures encapsulating the afore mentioned quintuple of $\{first, last, stride, length\}$. Once the data and the distribution have been successfully registered, the application can initiate data transfer using the **sendData** call. This is a blocking call and returns only when all the data has been

```
/*convert length to cm from inches*/
input.length = output.length • 2.5;
/*convert mass from kg to lbs*/
input.mass = output.mass • 2.2;
```

Figure 5. *A simple transformation function to convert units from the metric system to the imperial system*

sent, thereby allowing the application to modify the data buffer after it returns. Note that the call does not block until the data is *delivered*, to allowing a measure of asynchronocity between the producer and the consumer.

The data consumer also follows the same initialization and registration steps, but it has the added ability to specify one or more transform functions. These transform functions process the data before it is delivered to the client and can be executed either at the data source (thus reducing the load on the consumer and/or reducing the network bandwidth consumed by data transmission) or the data client (thus reducing the load on the producer). The client issues a **rcvData** call to get the data. This call is blocking and returns once the data has been collected from all participating data sources. Both the producer and the consumer can update their distributions at runtime using the **updateDistribution** function.

Transform functions are among the more interesting components of $XChange_{mxn}$ functionality. They may be written in ECL (a subset of C), which then results in their runtime installation via dynamic binary code generation on the target machine (we support most common computer architectures, including SPARC, MIPS, IA32, and in the near future IA64). Alternatively, resulting in some level of operating system dependence, they may be exported as dynamically loadable modules. In either case, the processing times of transform functions depend on their innate complexities. A simple transform is shown in Figure 4. This function performs a unit conversion. A more complex transform shown in Figure 4 performs image downsampling.

Transform functions have two uses. First, they can be used to change data so that the recipient gets it in the form desired, thereby avoiding additional data copying or manipulation to correct data mismatches. Second, they can be used as data filters, which allows applications to perform dynamic data differentiation. Such methods have been shown useful in the HPC community [5] for reducing network traffic [19], for reducing loads on storage or I/O subsystems, and for controlling the data volumes to which applications and end users are exposed.

The current implementation of $XChange_{mxn}$ assumes that transform functions are written by application developers. Our future work is automating their generation, by declaratively describing each component's data representa-

```
...
/*ip is the input event
op is the output event*/
for (row = 0; row < ip->height; ++row) {
 for ( col = 0; col < ip->width; ++col) {
  op->data[out_indx] =
  ((0.299 * ip->data[indx] +
   0.587 * ip->data[indx+1] +
   0.114 * ip->data[indx+2]) + 0.5 );
  indx += 3;
  ++out_indx;
  }
}
return 1;
```

Figure 6. *Filter function to convert a 24bit color image to 1bit black and white image*

tion via XML schemas and then using XML-level descriptions of the representational changes associated with component communications to create efficient binary codes that implement those changes.

5. Evaluation

Two different types of benchmarks are used to evaluate the $XChange_{mxn}$ system. The first are microbenchmarks to validate the basic performance of $XChange_{mxn}$ and to analyze the overheads associated with redistribution processes. The second uses the SmartPointer framework [29] as a realistic use case for the $XChange_{mxn}$ interface. All of the measurements reported in this paper are performed on a Linux cluster with gigabit Ethernet interfaces and Pentium III Xeon processors.

5.1. Matrix Transpose Microbenchmark

A simple and common method of data redistribution is matrix transposition. Our experiments use a matrix of size NxN that is shared by two components, each composed of a parallel cohort of $P/2$ processors. Emulating differences in data layout and representation, in one of the components, the matrix is stored striped over its columns, and in the other over its rows. At each iteration step, the column data is redistributed to the rows. Thus, each process P_i sends a chunk of size $2N^2/P$ to each of the other processes in the row component.

Measurements use two versions of this application, built with $XChange_{mxn}$ and MPI. The MPI code utilizes a custom MPI type that sends the correct data to the correct process with a single MPI call. Aside from the marshalling of data, no additional computation is carried out. The $XChange_{mxn}$ version is implemented using the system's generic redistribution mechanism.

Figure 7 shows the times taken for the transpose for various data sizes. Looking at the trend line, it is clear that both implementations approach an asymptotic value for large data sizes from which we can calculate the effective bandwidth usage of the data transfer. This bandwidth is necessarily lower than the total available bandwidth because we ignore the marshalling and unmarshalling overheads in the system. However, it can be seen that the performance difference between the two implementation is less than 15%. Additionally, in the small data regime (message size less than 20 kB), the $XChange_{mxn}$ performance meets or exceeds that of the MPI solution. Further, for a fixed global data size, the smaller message size corresponds to a larger number of parallel processors, which shows that $XChange_{mxn}$ should have good scalability to large numbers of processors.

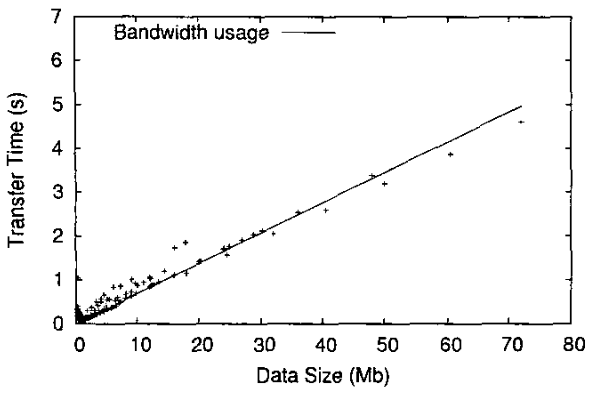
The higher overheads in $XChange_{mxn}$ compared to MPI can be attributed to the additional processing required for handling dynamic data exchanges. The MPI implementation uses a priori knowledge to realize appropriate data transfer schedules, which $XChange_{mxn}$ cannot do. We posit, however, that even in this case, the performance differences with using $XChange_{mxn}$ may be offset by the programmatic ease with which data distribution is handled.

5.2. The SmartPointer Benchmark

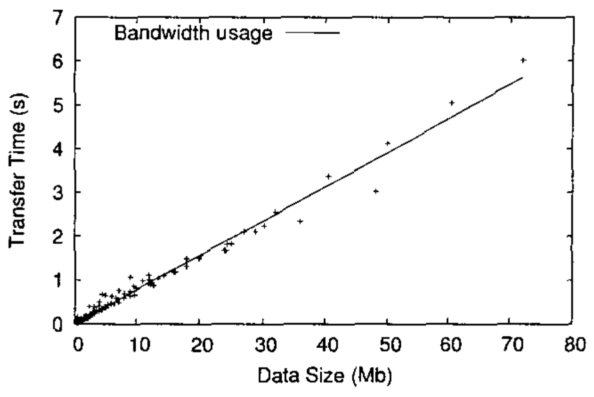
5.2.1 Comparison with *ECho*

To demonstrate the low overheads experienced by $XChange_{mxn}$, we next compare the performance of the $XChange_{mxn}$ version of SmartPointer with its original ECho implementation. Both versions use dynamic source-side filtering to handle the distribution of atom data from the DataSource to the BondServer. The $XChange_{mxn}$ version, however, has a much smaller code base, because of its more intuitive mechanism for describing the distribution and transfer of data.

Two cases are of interest. In Figure 8(a), we show the performance of both implementations when the computation time is significantly greater than the network transfer time. In Figure 8(b), we show the performance when the network transfer time dominates total time. Both cases illustrate that the overhead of using $XChange_{mxn}$ as an abstraction over ECho is quite small. In fact, in the case of a simple 1-by-1 transfer, $XChange_{mxn}$ is significantly faster. This is due to two factors: (1) $XChange_{mxn}$ uses latency hiding to overlap computation with network transfers and (2) $XChange_{mxn}$ has several additional optimizations for the special case in which all data is sent to a single node (the M-by-1 case).



(a) MPI data transfer time



(b) $XChange_{mxn}$ data transfer time

Figure 7. *A comparison of the effective bandwidth utilization between the static MPI solution and $XChange_{mxn}$. We interpolate bandwidth by only considering the size of the data set and the time taken to transfer the data, ignoring marshalling and redistribution overheads. The effective bandwidth for MPI at large data sizes is 115 Mbits/s and for $XChange_{mxn}$, it is 102 Mbits/s. Note that at small data sizes/large numbers of processors, $XChange_{mxn}$ and MPI have no appreciable difference in performance.*

5.2.2 Data-driven Distribution

This section's experiments demonstrate the advantage of using the $XChange_{mxn}$ approach to data-driven distribution, by allowing the DataClient to customize data emitted by the BondServer.

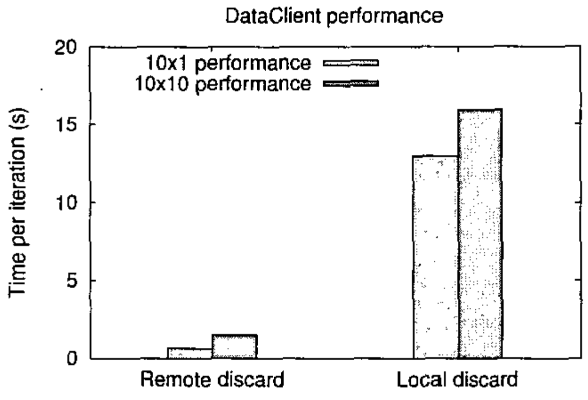
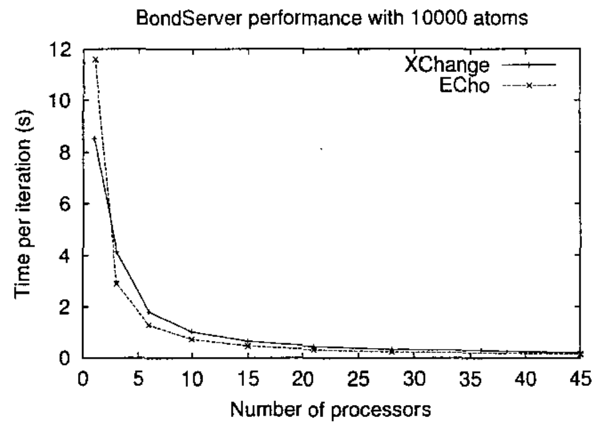


Figure 9. *Data-driven distribution performance. Low overhead of redistribution is demonstrated by comparing the performance of a 10x1 and a 10x10 interaction*

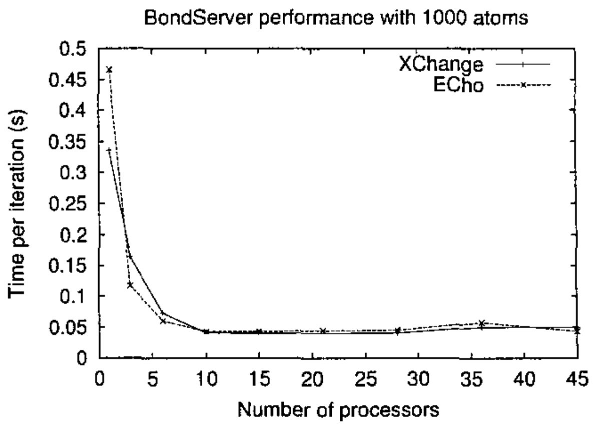
It is not unusual for DataClients (including restart files and visualization clients) to require only parts of the data being published by a source. We measure the improvement in performance for two possible solutions to this problem. A client can either discard data locally after receiving it (thus allowing the source to continue unfettered), or the source can discard data while sending it. We term the first scenario "local discard" and the second "remote discard". Figure 9 shows the performance improvements obtained by using remote discard.

A potential issue with remote discard is the need to perform additional processing at the data source, thereby potentially causing a slowdown of the data source. We measure this slowdown by comparing its performance with a single client (thus running only a single function at the source) against 10 clients. It is evident from our measurements that changes in data distribution (i.e., the customization of distributions to different client needs) do not result in substantial server-side overheads. Of course, this does not hold for complex per-client customizations via transformation functions (e.g., data compression via general compression methods [28]. For such cases, we plan to introduce the additional notion of overlay networks into a future implementation of $XChange_{mxn}$.

A potential advantage of performing remote discard is a reduction in network overhead at the data source. In our experiments, we noticed a very slight increase in the time



(a) Computation limited



(b) Network overhead limited

Figure 8. *Smart Pointer performance comparison between the original ECho-based implementation and the $XChange_{mxn}$ implementation*

per iteration for the BondServer moving from 1 client to 10 clients, but a marked reduction in moving from local discard to remote discard. This is because of the reduced number of executions of the network protocol stack at the server due to reduced data volumes. More generally, by giving applications the ability to make their own choices about the transformation functions to be used and the locations at which they are used (i.e., in sender or in receiver), $XChange_{mxn}$ provides to end users a flexible framework for implementing M-by-N data exchanges that best match the computational platforms they are currently using.

6. Related Work

There have been several previous efforts at creating general frameworks for high performance data distribution mechanisms. Highly efficient, domain-specific architectures for redistributing data between parallel applications are described in [10, 20, 23, 26]. In [16], an 'M-by-N' distribution mechanism for the efficient visualization of parallel data is proposed and evaluated. Generalizing from such domain-specific solutions, the PAWS system [3] provides a coupling mechanism for parallel applications that provides collective invocation as well as parallel data redistribution. The problem domain addressed by this effort is that of coupled parallel applications and hence, does not address large-scale scientific experiments in which the number of components changes dynamically or where different types of clients dynamically attach to data sources. In [11], the authors propose a more dynamic solution to the problem initially addressed in [3]. The distributions can be varied dynamically, but no support is provided for differing types of distributions or differing data formats.

MetaChaos [13] provides generic functionality for connecting separately developed applications. MetaChaos requires the builder of a data parallel library to provide *linearization* functions for the parallel data. This approach works well when no data transformations are required between the data source and data sink, but the lack of data transformation and reduction limits its ability to couple applications with different data requirements. Its notion of data linearization is an interesting precursor to the $XChange_{mxn}$ mechanism's ability to dynamically install custom data filters and transformers. DataCutter [5] allows a developer to perform *filtering* of data at or near (in network terms) a data source. This is accomplished by providing functionality for running filters on the data at the source, but there is no support for dynamic data redistribution. Finally, the Common Component Architecture (CCA [2, 7]) is developing a generic interface for coupling parallel applications [8]. We intend for the $XChange_{mxn}$ middleware to conform to these specifications when they have been completed, but we note that this initiative does not currently address the unique problems associated with coupling format translations with component interactions. NetFx [12] is a combination of compiler and runtime enhancements that allow applications to describe and distribute data in parallel applications. Unlike NetFx, $XChange_{mxn}$ uses only a runtime extension and provides additional support for data filtering and transformation.

Loose couplings like those targeted by $XChange_{mxn}$ are an important part of web services infrastructures in enterprise computing. Middleware packages such as CORBA, DCOM and Java RMI don't allow interactions between applications that don't support the same data type. However, there have been efforts to allow applications to evolve their message formats and interoperate between differing message formats [1, 6]. There has also been a drive towards transformation languages such as XSLT and fxt [4]. XML is not designed as a high performance transport, thereby making its use difficult for scientific applications. Furthermore, no support is provided for addressing the different levels of parallelism that exist in scientific applications, nor is there any push for reducing network traffic by filtering out unneeded data at the server. $XChange_{mxn}$ seeks to address these problems of loose coupling and application evolution in the high performance domain.

7. Conclusion

This paper presents a framework for dynamically coupling applications in the high performance and scientific computing domains. The importance of addressing this issue is derived from the growing complexity of modern high performance applications. This has caused developers to move to a component-based approach, with each component potentially using different internal data representations and parallelization methods. This results in the need to efficiently distribute and re-distribute data across communicating, parallel components running on heterogeneous underlying computing platforms. Such platforms range from sets of clusters connected via commodity networks to supercomputers linked with specialized high end networks.

The $XChange_{mxn}$ middleware implements mechanisms for the efficient coupling of parallel components across distributed computing infrastructures. Additional flexibility compared to previous work on M-by-N interchanges is provided via a 'blind' publish/subscribe interface, enabling entirely dynamic component couplings, whenever and wherever desired. The middleware also permits the data-driven customization of such component interactions, and in addition, clients can transform the formats of incoming data. The idea is to implement efficient couplings even across somewhat mismatched components, thereby broadening the utility and applicability of the middleware mechanisms.

The need for integrating dynamic data distribution, differentiation and even transformation into an efficient parallel data transport is demonstrated with multiple parallel applications. $XChange_{mxn}$ is evaluated with microbenchmarks and with a multi-component framework for scientific modeling and real-time collaboration, termed SmartPointer. Measurements show that $XChange_{mxn}$ performs well when compared to current static solutions for component coupling (i.e., MPI), and it can outperform static solutions due to its ability to control the locations of the data transformation and filters associated with parallel data transports.

Our future work will concentrate on developing more intuitive descriptions for the distribution and re-distribution of data structures, perhaps using a high level structured language like XML. In addition, we are working on the automatic generation of the required transformation code, in order to minimize necessary user involvement. An interesting topic is the addition of QoS support to the framework. At present, two coupled applications with a two way data exchange will not perform optimally when there is a difference in the resources available to each application. An approach to solving this problem is to dynamically vary the distribution of data to optimize application interactions. One element of that solution would be the addition of communication schedules to reduce network congestion between coupled applications. The combination of generating transformation code with resource-awareness in the framework would be a powerful solution to the problem of dynamic application coupling.

References

[1] S. Agarwala, G. Eisenhauer, and K. Schwan. Morphable messaging: Efficient support for evolution in distributed applications. In *Challenges of Large Applications in Distributed Environments (CLADE)*, June 2004.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. McInnes, S. R. Parker, and B. A. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *HPDC*, 1999.

[3] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *The Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[4] A. Berlea and H. Seidl. Transforming xml documents using fxt. *Journal of Computing and Information Technology CIT, Special Issue on Domain-Specific Languages*, January 2001.

[5] M. Beynon, R. Ferreira, T. M. Kurc, A. Sussman, and J. H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[6] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Technical Report 115, Digital Equipment Corporation, Systems Research Center, December 1995.

[7] 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 Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 51–59. IEEE Computer Society Press, August 2000.

[8] *MxN Parallel Data Redistribution @ ORNL.* http://www.csm.ornl.gov/cca/mxn/, Jan. 2004.

[9] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a high-performance remote-sensing database. In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, U.K., 1997.

[10] Y.-C. Chung, C.-H. Hsu, and S.-W. Bai. A basic-cycle calculation technique for efficient dynamic data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(4), 1998.

[11] K. Damevski and S. Parker. Parallel remote method invocation and m-by-n data redistribution. *The International Journal of HighPerformance Computer Applications (Special Issue)*, page (Accepted), 2003.

[12] P. A. Dinda, D. R. O'Hallaron, J. Subhlok, J. A. Webb, and B. Yang. Language and run-time support for network parallel computing. In *In Proc. 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC95)*, August 1995.

[13] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the Eleventh International Parallel Processing Symposium*. IEEE Computer Society Press, April 1997.

[14] G. Eisenhauer, F. E. Bustamante, and K. Schwan. Event services for high performance computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 113. IEEE Computer Society, 2000.

[15] G. Eisenhauer, B. Schroeder, and K. Schwan. From interactive high performance programs to distributed laboratories: A research agenda, 1996.

[16] G. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing faulttolerance, visualization, and steering of parallel applications, 1996.

[17] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of High Performance Computing Applications*, 11(3):224–236, August 1997.

[18] C. Hill, C. DeLuca, V. Balaji, M. Suarez, A. da Silva, and the ESMF joint Specification Team. The Architecture of the Earth System Modeling Framework. *Computing in Science and Engineering*, January 2004.

[19] C. Isert and K. Schwan. ACDS: Adapting Computational Data Streams for High Performance). In *Intl. Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[20] S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase array redistribution: modeling and evaluation. In *Proceedings of the 9th International Symposium on Parallel Processing*, pages 441–445. IEEE Computer Society, 1995.

[21] K. Keahey, P. Fasel, and S. Mniszewsk. PAWS: Collective Interactions and Data Transfers. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*. IEEE, August 2001.

[22] T. M. Kurc, A. Sussman, and J. Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[23] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proceedings of Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, Feburary 1995.

[24] B. Schroeder, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin, S. Szou, J. Vetter, R. Wang, F. Alyea, B. Ribarsky, and M. Trauner. Framework for collaborative steering of scientific applications, 1997.

[25] *Department of Energy TerraScale Supernova Initiative.* http://www.phy.ornl.gov/tsi/.

[26] D. W. Walker and S. W. Otto. Redistribution of block-cyclic data distributions using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.

[27] K. Wang, S. Kim, J. Zhang, K. Nakajima, and H. Okuda. Global and localized parallel preconditioning techniques for large scale solid earth simulations. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 258.1. IEEE Computer Society, 2003.

[28] Y. Wiseman and K. Schwan. Efficient end to end data exchange using configurable compression. In *The 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[29] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smartpointers: Personalized scientific data portals in your hand. In *Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, page 20. IEEE Computer Society, 2002.