

Flexible Control of Data Transfers between Parallel Programs *

Joe Shang-chieh Wu and Alan Sussman

Department of Computer Science

University of Maryland, College Park, MD 20742, USA

{meou,als}@cs.umd.edu

Abstract

Allowing loose coupling between complex e-Science applications has many advantages, such as being able to easily incorporate new applications and to flexibly specify how the applications are connected to transfer data between them. To facilitate efficient, flexible data transfers between applications, in this paper we describe methods for making decisions at runtime about when such transfers will occur, and for flexibly specifying when data transfers are desired. We also present preliminary experimental results that measure the overheads incurred by our approach.

1. Introduction

Compared to the development of traditional large scale scientific simulations, the new e-Science paradigm composed of heterogeneous applications has many advantages. Integrating well-tested modules that each best model some part of the physical system being simulated, and deploying them as a Grid-based loosely coupled system, rather than developing a single tightly coupled monolithic code, can be both a more efficient and flexible way to develop large-scale software systems.

The coupled approach also makes it easier to investigate the entire physical system broken down into its various components. For example, the family of various magnetohydrodynamics (MHD) equations used to model space weather is an example. As discussed by Gombosi et. al. [8], the MHD equations are used extensively in numerical-based large-scale space science simulations, and the impact of those different MHD codes can be investigated more easily in loosely coupled software systems.

In addition, the computational capabilities of emerging e-Science applications allow us to explore more thoroughly

important physical phenomena that are so complex that no single research group has the ability to attack the problem alone. For example, the Earth System Modeling Framework is a consortium of several U.S. federal agencies and fifteen universities [3]. The loosely coupled approach also benefits other multi-scale, multi-resolution simulations, as described for petroleum reservoir simulation in [13].

The various coupled applications may employ different scales in both time and space, either because of the scale of the phenomena being modeled or because of the numerical techniques being employed. The interfaces between programs, which are the shared boundaries between physical models, must be made consistent in both time and space to obtain correct results. As compared to issues related to dealing with large variances in time scales, spatial resolution differences are easier to deal with, at least from the coupling viewpoint. Figure 1 shows one way of dealing with multiple spatial scales — introducing an agent that performs interpolation on the grids used for the numerical modeling in the two components. The role of the agent includes transformation between coordinate systems and the proper scattering/gathering of data across shared boundaries. The MxN working group in the Common Component Architecture (CCA) Forum [2, 4] is also addressing some of these issues, but not the one we concentrate on here, namely dealing with the varying time scales in different applications.

In this paper we offer a low overhead, modular architecture, as well as a runtime solution for dealing with multiple time scales across applications. Both Interoperable MPI (IMPI) [7], which is a set of protocols to integrate multiple MPI programs, and MPICH-G2 [16], a Grid-enabled implementation of the MPI v1.1 standard, leave the solution of these issues up to the application. Meta-Chaos [5], InterComm [15], Parallel Application Work Space (PAWS) [11], and the Model Coupling Toolkit (MCT) [14] all focus on solving the problem of sharing distributed data structures between parallel application components. Roccom [9] offers an object-oriented software framework for parallel rocket simulations, and the Collaborative User Migration User Library for Visualization and

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408), and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095, and NASA under Grants #NAG5-11994 and #NAG5-12652.

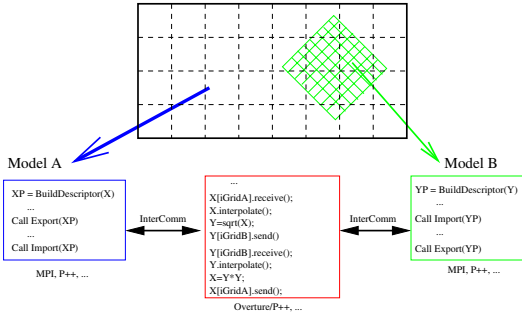


Figure 1. Dealing with different spatial resolutions

Steering (CUMULVS) [6] is middleware for the remote visualization and steering of parallel applications. All these systems mainly target support for moving data between parallel components, while this paper concentrates on the problem of when to perform the data transfers.

Coordination languages, such as Linda [1] and its variations, offer a general purpose tool for a distributed computing environment. However, the generalized functionality such languages offer does not work well in a high performance simulation environment. The inherent complexity of the generalized approach pays a high, and unnecessary, cost in overhead. We advocate that, instead of using a general and high overhead solution, a more targeted and low overhead approach to the problem is a better solution.

Although arbitrary coordination between the components (programs) in a loosely couple scientific system is not necessary, two properties should hold. First, only the specification of the interface to other programs needs to be known by each coupled program, making it easy to replace a program by another with the same interface. Second, the decision about when a data transfer will occur should be made by the runtime system, which is the only entity that has a complete view of all the interacting components. One implication of this design is that if a component generates data for which there is no other component that requires the data, the unneeded data should be discarded by the runtime system without affecting the behavior (or performance) of the entire system. All that is important in such a loosely coupled system is that data that is required by one component should be delivered in a timely manner from some other component, and data that is provided by a component but not needed by any other component should not cause any correctness or performance problems elsewhere.

The second property is related to the common behavior of numerical simulation techniques. Although classical algorithms compute solutions over time at a fixed frequency (time step), variable-sized intervals (explicit time steps) have theoretical and practical advantages in some

applications, as shown for example in [10]. Visualization of scientific simulation data, for example as shown in [12] may also benefit from variable-sized intervals. Rather than ingesting generated data at fixed intervals, the visualization program can use a loop to fetch whatever is the most current data as the simulation is running, and then generate the visualization data to be viewed. In this scenario, the interval between data fetches depends on runtime variables such as the graphics processing speed, the system load where the visualization is running, and the size of data to be transferred — it does not need to be a constant. This means that a good solution to the problem would not fix the time when the data exchanges should be performed in advance, but that all decisions should be made at runtime based on the current overall system state.

The rest of the paper is organized as follows. Section 2 describes the overall system architecture, and Section 3 explains how flexible matching between applications providing data and applications requiring data is specified and performed. Preliminary experimental results are described in Section 4. Section 5 concludes and describes some future work.

2. Architecture

Many scientific computing programs, for example in a set of coupled programs for a simulation of a physical system, employ numerical algorithms to solve systems of equations iteratively. Each iteration is typically composed of two parts: computation on the domain where that program is relevant and data exchange across physical boundaries shared with other programs. Our design provides methods for exporting (sending) and importing (receiving) data between programs, once the relevant (distributed) data structures are defined.

Although each program must define its contributions (called *regions* in our framework) to a data transfer, the related counterparts on the other side of the data transfer do not need to be defined. From the point of view of a data exporting program, the program defines its regions once, and exports the desired data as often as it desires, nominally when a new, consistent version of the data across the parallel program is produced (note that the data for a region can span multiple processes in the program, so the parallel program must ensure that a consistent version is exported). The program does not have to concern itself about which and how many programs will receive the data, or even whether a data transfer will actually occur. Data importing programs also only define their regions once, and import data as needed, without knowing anything about the corresponding exporters. The connection between importer and exporter programs is provided by the framework, and an example is shown in Figure 2.

<pre> define region Sr12 define region Sr4 define region Sr5 ... for(...) { export Sr12 export Sr4 export Sr5 ... } </pre>	<pre> define region Sr0 ... for(...) { import Sr0 ... } </pre>
Exporter Ap0 code	Importer Ap1 code

Figure 2. Example exporter and importer programs

Our framework uses a configuration file that is separate from the importing and exporting programs, and contains information about how to connect imported and exported regions as well as all the information required to execute the coupled programs. An example configuration file looks like:

```

Ap0 cluster0 /home/user1/bin/Ap0 2 ...
Ap1 cluster1 /home/user1/bin/Ap1 4 ...
Ap2 cluster2 /home/user1/bin/Ap2 16 ...
Ap4 cluster3 /home/user1/bin/Ap4 4 ...
#
Ap0.Sr12 Ap1.Sr0 REGL 0.05
Ap0.Sr12 Ap2.Sr0 REGU 0.1
Ap0.Sr4 Ap4.Sr0 REG 1.0

```

The configuration file contains two parts. The first part describes the required runtime environment information, including (1) what resource to run each program on, (2) the file system location of the executable program, and (3) what command and switches to use to run the program (including how many processors to run on for a parallel program). In the example, the parallel program *Ap0* will run on machines in cluster *cluster0*, its executable is located in */home/user1/bin/Ap0*, and it will run on 2 processors in the cluster. Runtime information is also shown for three other programs that are part of the coupled set of programs.

The second part of the configuration file describes the mappings between exporter regions and importer regions, by specifying the data that will be transferred at runtime. In the example, the first two mappings specify that data must be transferred from region *Sr12* in program *Ap0* both to region *Sr0* in program *Ap1*, using a matching criterion of REGL with a precision of 0.05, and to region *Sr0* in program *Ap2*, using a different matching criterion of REGU with a precision of 0.1. The third mapping specifies that region *Sr4* in program *Ap0* transfers data to region *Sr0* in program *Ap4*, with a matching criterion of REG with a precision of 1.0. Note that even though exporter region *Sr5*

in program *Ap0* has been defined, as seen in Figure 2, that region will not be involved in any data transfers, because the configuration file does not specify any corresponding importer regions. The details of the three matching criteria (REGL, REGU, and REG) will be discussed in Section 3.

The design goal of using a separation configuration file is to provide flexibility. By defining the mappings between source and destination regions separately from each program, a user can easily change the runtime matching relationships, *without* modifying the source code in either the source or destination program. Similarly, it becomes straightforward to replace the source (or destination) program with another program that provides the same interface, by simply modifying the configuration file. Moreover, each program can be developed independently and only the author of the configuration file, who is the one coupling the programs that will run, needs to specify the detailed information about the runtime data transfers.

2.1. System Architecture

We briefly discuss the main components of the runtime system that supports the matching process, as shown in Figure 3. The configuration file discussed earlier is read in the initialization stage for each program. As shown on the left side of Figure 3, if an exported data object might get a match, meaning that it might match an import in some other program, a copy of it is saved by the system. However, if no matching specification exists that requires the exported data object, or the exported data object will not match any potential import, based on the matching criteria in the configuration file and the imports that have already been seen by the system, the system does nothing and returns to the caller.

A matching decision is made when an import request is received. Based on saved data and the matching criteria for that import request, the possible status for the received request will be *Yes*, *Never*, or *Pending*. As shown in the middle of Figure 3, a data transfer from the exporter to the importer will be triggered if the request can be satisfied by saved data (a *Yes* response). If a decision cannot be made yet because possible candidates will be exported in the future, the answer would be *Pending*. If the system can determine that a match can never be made now and in the future, the answer *Never* will be returned to the importer. The details of the matching process will be discussed in Section 3.

We now describe how an import request is handled in the importing part of the runtime system. As shown in the right side of Figure 3, when a data object is imported the system first checks, based on the information obtained from the configuration file, whether a corresponding exporter exists. If one exists, the system sends a request to the exporter, and waits for an answer, otherwise the request fails (returns

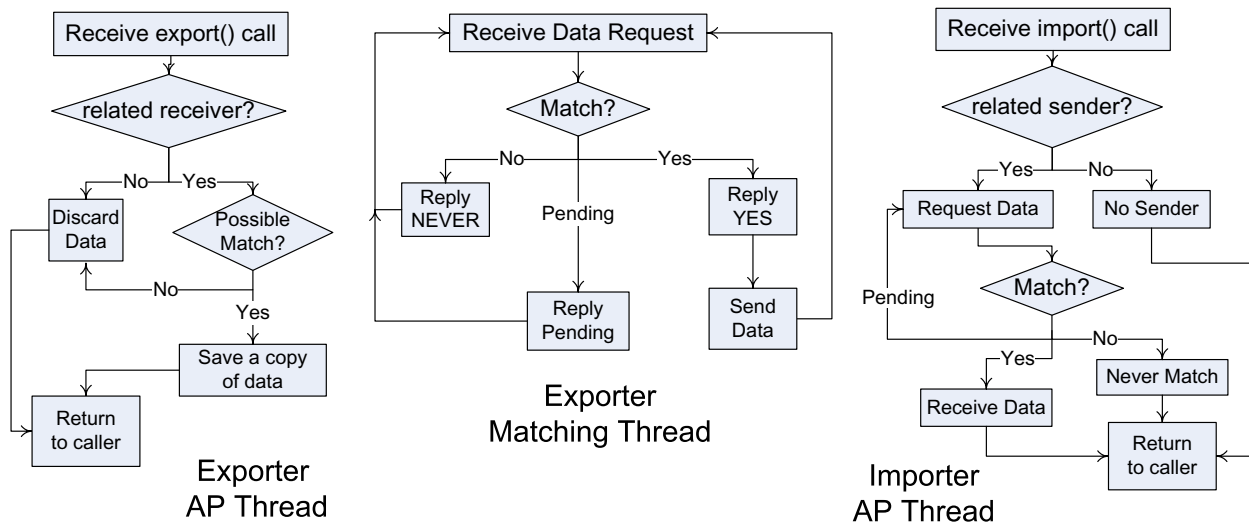


Figure 3. Main system components

Never) because no corresponding exporter exists. If the exporter returns Yes, the importing part of the system issues a receive for the matching data object. If the exporter returns Never, the system also returns that to the caller. However, there are two options when the exporter returns Pending. If a blocking import call is made the runtime system in the importing part of the system will wait until a Yes or Never answer is returned from the corresponding exporter (as specified in the configuration file). If the importer made a non-blocking import call, it must test the handle returned by the runtime system until the import call has either completed or failed (or wait until the operation completes).

Applications may be parallel programs, running as multiple processes on multiple host machines. For parallel programs (e.g., MPI programs), import and export requests are collective [17], meaning that all processes in a program must make the same calls in the same order, with consistent parameters. In our system, one of processes is selected by the runtime system as the *representative* process for those parallel programs. The representative process has three additional responsibilities. First, it forwards requests and replies from the other processes in this parallel program to other programs. Second, it exchanges forwarded requests and replies with representative processes of other programs. Third, the representative in the importer caches matching information obtained from the exporter program and makes it available to all processes within the importer program. With the help of those representatives, consistent decisions can be made across all processes in parallel programs and results are shown in Section 4.

3. Matching exports to imports

As discussed in Section 2, exporters generate time stamped data objects and importers request such objects. For example, if an exporter produces an array **A** (i.e., a data object containing the contents of array **A**) at time stamp 1.2, the stamped data is composed of the data from array **A** and a time stamp 1.2. For simplicity, we use `data@stamp` to denote the stamped data. In our example, the stamped data is `A@1.2`. Also we require the time stamps in a sequence of export or import calls for the same data object to be an increasing function of execution time — that is, if the most recently exported data object is `A@1.2`, the next data object from the same exporter (or importer) must be `A@y` with $y > 1.2$ (of course, the contents of **A** may have changed between the two exports).

Data transfers are initiated by a request from an importing component. For each request, the corresponding exporting component must determine which data object *matches* the request, if any. In other systems, this issue is solved implicitly — the logic is embedded in the applications. For example, if the importer needs data every 5 time units, the exporter will send data every 5 time units. This method is a simple and efficient solution when both the importer and exporter applications are implemented by the same person, or by the same research group. However, such a scenario becomes a problem when the importers and exporters are produced by different research groups, for example, as described in [3]. Similarly, it may be difficult to build the logic in each application to match exports and

imports if time stamps on objects are not generated in a regular fashion (e.g., the time scale in the importer is not a multiple of the one in the exporter).

3.1. Matching policies

We now describe our solution to the matching problem. Consider the following scenario: the exporter produces a sequence of data objects with time stamps: A@1.1, A@1.2, A@1.5 and A@1.9, and the importer requests a data object that matches A@1.3. The question becomes, which exported object matches the request, if any? We define a matching criterion denoted by a ⟨matching policy, precision⟩ pair. The matching policy determines whether and how a match is made between two time stamps. For example, one matching policy that we call the greatest lower bound (GLB) requires that A@1.2 be the match for the A@1.3 request (the names we use are from the point of view of the importer request). Another example matching policy is called the least upper bound (LUB), which for our example would match A@1.5 to the A@1.3 request. From the viewpoint of the importer, GLB can be viewed as matching the time stamp of the latest export with time stamp less than or equal to the requested one, and LUB as matching the earliest export with time stamp greater than or equal to the requested one.

Interestingly the same exported data object might be the match for different requests. As in the previous example, if the matching policy is GLB and the first request is A@1.3, A@1.2 is the match. If the next request is A@1.4, A@1.2 is once again the match.

3.2. Precision

Although the LUB and GLB policies offer some flexibility, bounding the time stamp values that are acceptable can also be useful — an application may not want to get a data object that has a stamp too far from what it has requested. We allow the specification of a *precision* for each matching policy, which enables such control over stamp matches — the precision determines how far apart the stamps in the exporter and importer are allowed to be.

More formally, the precision specified for a match is the tolerance allowed between the stamps specified by the importer and the exporter for the matching data objects. For example, if the GLB policy is specified with a tolerance of 0.05, the importer request from our earlier example would return Never for the request A@1.3 because no exporter object has a stamp in the range [1.25, 1.3]. The A@1.2 in the exporter would be the GLB match for the importer request A@1.3, but it is not within the specified interval.

3.3. Supported matching policies

In addition to the previously described GLB and LUB policies, we also define several other matching policies that may be useful for coupling some types of applications. We use x as the requested time stamp from the importer, $f(x)$ as the potential matched stamps from the exporter, and p as the desired precision.

LUB Least upper bound match — the minimum $f(x)$ such that $f(x) \geq x$.

GLB Greatest lower bound match — the maximum $f(x)$ such that $f(x) \leq x$.

REG Region match — $f(x)$ minimizes $|f(x) - x|$, and $|f(x) - x| \leq p$.

REGU Region upper match — $f(x)$ minimizes $f(x) - x$, and $0 \leq f(x) - x \leq p$.

REGL Region lower match — $f(x)$ minimizes $x - f(x)$, and $0 \leq x - f(x) \leq p$.

FASTR Fast region match — any $f(x)$ such that $|f(x) - x| \leq p$.

FASTU Fast upper match — any $f(x)$ such that $0 \leq f(x) - x \leq p$.

FASTL Fast lower match — any $f(x)$ such that $0 \leq x - f(x) \leq p$.

We enumerate some observations about the various matching policies:

1. Given a requested stamp x , if a corresponding $f(x)$ cannot be found based on the given matching criteria (i.e., the matching policy and precision), Never would be returned to the importer.
2. The REGU (REGL) policy is the same as LUB (GLB) with a precision value added. LUB (GLB) can be viewed as REGU (REGL) with a precision of infinity.
3. For region matchings (LUB, GLB, REG, REGU, and REGL) the matching stamp, $f(x)$, is selected such that the difference between $f(x)$ and x , called the reference stamp here, is minimized if more than one stamp falls within the precision. However if multiple stamps are eligible, and the importer does not care which one is returned, FASTR, FASTU, and FASTL can be used and the overall performance would be better than for the more tightly constrained matching policies.

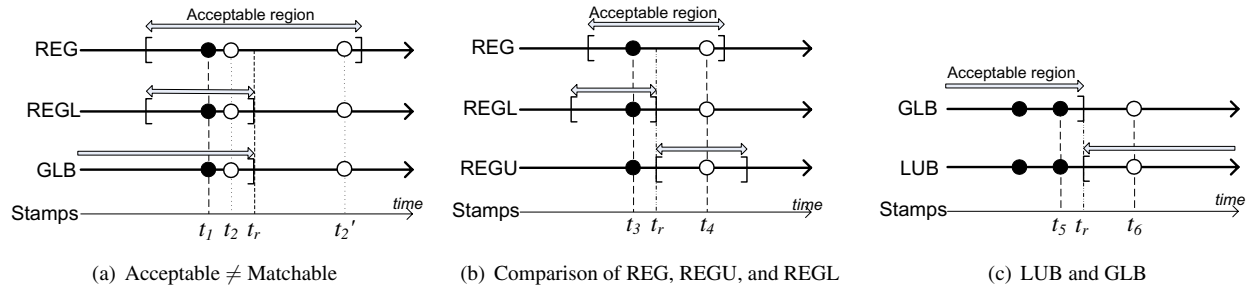


Figure 4. Region match properties

3.4. Region matchings properties

Several issues are specific to region matchings. First, for GLB and REGL, the matching result remains "Pending" even if the most current exported object stamp is acceptable — because the next exported object stamp *might* be the match, as shown in Figure 4(a). Here we denote exported stamps by filled circles and possibly future stamps by hollow circles. In this example, $A@t_1$ is the most current exported object in the system and is also the current match candidate, but it will not be *the* match if the next exported object is $A@t_2$. However a final decision that $A@t_1$ is really the match can be made if the next exported data object is $A@t_2'$ rather than $A@t_2$. If the most current stamp t_1 is also before the reference stamp t_r , the REG matching policy also has a similar property, as shown in Figure 4(a).

In such situations, a deadlock may occur if the runtime system has only one buffer to store saved objects with different stamps — because the exporter has no place to save the second stamped data object, so the status would be "Pending" forever. Although using more than one buffer is the simplest solution to this problem, that might not be feasible if the size of the data objects is very large, which may not be uncommon in scientific applications. In this case, one buffer and one look-ahead can be used — the look-ahead checks only the stamp of the next exported data object to see if it is better than the previous one, so that the buffer can be overwritten if needed (the new object is the match rather than the previously saved one). If the new object is not acceptable, then the runtime system has determined that the current saved object is the match and the data transfer of the buffered object can be performed.

Second, for the REG, REGU, and REGL policies, the match request is for a bounded interval around the reference stamp which is either inside (REG) or on the boundary of (REGU, REGL) the interval. In all three cases, match decisions can only be made *after* a data object with a stamp value greater than the reference stamp is exported — otherwise the match cannot be determined for certain. This is

not a problem for the REGU policy, for which that waiting data object is the desired match, but some delay would be introduced in determining a match for the REGL policy, because the new data object is not the one that is returned as the match — it is only used to determine that the match can be made. For the REG policy, the additional delay may or may not be extra overhead — that depends on which stamp is the match, the one just before or after the reference stamp. In the example in Figure 4(b), t_r is the reference stamp and t_3 is the most current stamp. When $A@t_4$ is exported, it would be the match for the REGU policy and it also assures that $A@t_3$ is the match for the REG and REGL policies.

Third, there is an interesting connection between the LUB and GLB policies — their decision making process is very similar. The decision for when a match is made for both policies can only be made when a data object is exported with a time stamp greater than the reference stamp. More precisely, the decisions for both the LUB and GLB policies are made at exactly the same time (when the triggering export occurs), and the matching results are from two consecutive exports (the one just before the reference stamp and the one just after). In Figure 4(c), t_r is again the reference stamp and t_5 is the most current stamp. Both the matches for the LUB and GLB policies can be determined only after $A@t_6$ is exported. Here LUB will match $A@t_6$ while GLB will match $A@t_5$.

4. Preliminary Experimental Results

Our runtime system is implemented using C++, the C++ Standard Template Library, and Pthreads, and uses the InterComm library [15] to perform the parallel data transfers. To investigate the behavior and performance of the matching techniques, we performed experiments using a two-dimensional linear partial differential equation solved via the finite element method. The experimental environment is described as follows:

- The equation is $u_{tt} = u_{xx} + u_{yy} + f(t, x, y)$, a two-

dimensional diffusion equation with a forcing function. The forcing function can be viewed as the external input for u .

- $u(t, x, y)$ is a 512×512 array. In addition to its four edges, the following elements are also zero. $u(255, y, t)$, $u(256, y, t)$, $u(x, 255, t)$, and $u(x, 256, t)$. So the array is composed of four 256×256 arrays whose boundaries are set to 0.
- Program Ap0 runs with two processes (P00 and P01), and handles the forcing function f . Each process is responsible for a 32×256 local array, half of the total 32×512 array.
- Program Ap1 runs with four processes (P10, P11, P12, and P13), and is the numerical code for solving the diffusion equation. Each process contains a 256×256 local array, representing the global 512×512 array.
- Ap0 is the exporting program, and generates data at every basic time unit (for this example, the time units are arbitrary). Ap1 is the importing program and requests external data every ten basic time units. The matching criterion specified in the configuration file is $\langle \text{REGL}, 0.05 \rangle$. All the data in the Ap0 array are used as the exported data object.

In this experiment process P00 is the representative for Ap0 and process P10 is the representative process for Ap1. If we compare them to the other processes, the representative processes have extra work to perform. To investigate the performance implications of the extra work, we consider data transfers into three different data regions in Ap1, as shown in Figure 5, measuring overheads for each case. However we must first identify where the overhead comes from, which requires looking at a part of Ap1's source code:

```
for (...) {
    import region;           // step 1
    Compute u by finite difference; // step 2
}
```

In each iteration, Ap1 obtains external data via an import operation, and then computes a new value for u . The import operation (step 1), also shown in the right of Table 2, has to do two things. First, it must ask the Ap1 representative process to request a match from the exporter, Ap0. Second, if the request succeeds, Ap1 initiates the data transfer with exporter Ap0, as shown on the right side of Figure 3. Therefore Ap1 performs the following actions in each iteration, and we can measure the execution time for each action using the standard Posix `gettimeofday()` function:

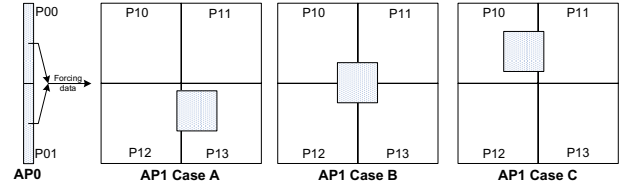


Figure 5. Experimental environment

```
for (...) {
    Request a match;           // step 1a
    Perform a data transfer;   // step 1b
    Compute u by finite differences; // step 2
}
```

Since its computation is much less expensive, Ap0 runs faster than Ap1, so Ap1's request is always satisfied immediately. Therefore, our measurements are for the case where there is no extra delay introduced by the exporter not having produced the desired data.

In the traditional approach to transferring data between applications, all matching information is embedded into the application, so only the data transfer (step 1b) and the computation (step 2) are needed. Therefore the overhead for doing the matching is the time spent in the match request (step 1a).

4.1. Experimental setup and result analysis

For our experiment, we measure the overhead on 6 processors of a cluster of Pentium-III 600MHz machines, connected via Fast Ethernet, running the experiment eleven times and averaging the times for the results for all three cases. In each run, 1001 matches are requested, and the measured time is the time for the slowest process to finish the matches. For all three cases for AP1's import region, the total execution time averaged across the eleven experimental runs is shown in Table 1, with standard deviations shown in parentheses. The execution time for each step in the slowest process as well as the time for step 1a in the fastest process is shown in Table 2. From the standard deviations seen in Table 1, we see that the measured results are consistent so the average values do indeed represent the actual performance seen for matches by an application.

For case A, most of the data object region in the importer is located in the memory of P13. and the representative process P10 owns no part of that region. As shown in Table 1, P13 requires the longest time to run, since it is receiving most of the transferred data.

The overhead actually seen by the whole program can be measured by looking at the slowest process. By measuring the time for step 1a, we can see the overhead that

	P10	P11	P12	P13
Case A	341 (3.5)	336 (4.0)	610 (2.2)	614 (1.5)
Case B	620 (1.5)	618 (1.4)	618 (1.4)	618 (1.4)
Case C	624 (4.1)	612 (3.8)	340 (3.4)	339 (5.0)

Table 1. Average execution time (standard deviation), in seconds

	the slowest process				fastest
	step 1a	step 1b	step 2	Ov	step 1a
Case A	944 μ s	6.1ms	605ms	13%	4394 μ s
Case B	708 μ s	2.9ms	613ms	20%	3468 μ s
Case C	535 μ s	6.8ms	614ms	7%	3703 μ s

Table 2. Overhead in the slowest process

is introduced by our techniques. As shown in Table 2, although 13% (944 μ s) of step 1 time is for step 1a, the time transferring the data in step 1b depends on the size of the data region transferred, while the time for doing the match is independent of the size of the data region. So the overhead would be a smaller percentage of the time for larger data transfers.

One of the tasks of the representative process is to cache the results of the match procedures, so that they can be used by other processes in the same parallel program. This makes it easy to ensure that all processes in the same program will get the *same* match, but not necessarily at exactly the same time. In fact, the fastest process (in this experiment P11) always makes the remote request to the exporter, so the other processes, including the slowest one (P13), only end up making a local request to the representative process. Therefore the 944 μ s overhead seen in P13 is really the cost for local communication between P13 and P10 within the parallel application and the expensive remote request (taking 4394 μ s, as shown in Table 2) is hidden behind the time taken in the slow process.

Next we consider an even distribution of the requested data across the processes (case B). In this case, the data region is equally partitioned across the four processes. As shown in Table 1, P11, P12, and P13 take about the same amount of time, and P10 is somewhat slower because of the extra work for being the representative process. We again look at the overhead in the slowest process, this time P10¹. As shown in Table 2 the slowest process also makes local match requests, and the expensive remote request to the exporter, taking 3468 μ s, is again hidden.

Interestingly, the overhead for case A (944 μ s) is greater than that for case B (708 μ s). In case A, the only work for P10 is as the representative process, but in case B, P10 has

¹Case B is faster for step 1b than for Case A because only a quarter of data must be transferred into the slowest process.

additional work — transferring one quarter of the data. It seems that the overhead in case A should be smaller than in case B. The reason for this behavior is that a match request via the local network to the representative process is slower than a request that is satisfied via local memory. Our implementation of the runtime system is multi-threaded, and the representative process uses a separate thread to store match results and answer match requests from other processes in the same application. P10 is the representative process for both cases. In case A, P13 is the slowest process, and each local match request is made via the network. However in case B P10 is also the slowest process, so the match request requires only reading the cached match result from inside the *same* process.

The last scenario (case C) is that the representative process P10 receives most of the requested data. As shown in Table 1, P10 is the slowest process. As shown in Table 2, the overhead for the remote request (3703 μ s) is again hidden behind the work done in the slowest process, which makes a fast local request. In this case, the overhead is 535 μ s. If we compare the overhead of case B (708 μ s) against that of case C (535 μ s), the explanation above comparing cases A and B does not apply because P10 is the slowest process in both cases. The overhead for case C is smaller than for case B due to network congestion. Although our network has a full duplex Fast Ethernet switch, the bottleneck is the data sources, P20 and P21, that must send messages to all 4 processes in case B, but only single messages to process P10 in case C.

5. Conclusions and Future Work

We have described a low overhead solution to the problem of flexible control of data transfer between Grid-based loosely coupled programs. Our methods allow for efficient runtime decision to be made, enabling applications with widely varying scales in both space and time to communicate efficiently. This is a work in progress – we are currently investigating various issues, including the effect of busy exporters on performance, the performance of different matching policies, techniques for optimizing buffer management across multiple applications, and employing the techniques in large-scale scientific applications.

References

- [1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.
- [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 *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed*

- Computing*, pages 115–124. IEEE Computer Society Press, August 1999.
- [3] G. R. Asrar. A pathway to decisions on Earth’s environment and natural resources. *IEEE Computing in Science and Engineering*, 6(2):13–16, January/February 2004.
- [4] Common Component Architecture (CCA) MxN working group. <http://www.csm.ornl.gov/cca/mxn/>.
- [5] 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.
- [6] G. A. Geist, II, 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.
- [7] W. L. George, J. G. Hagedorn, and J. E. Devaney. Parallel programming with interoperable MPI. *Dr. Dobb’s Journal*, (357):49–53, February 2004.
- [8] T. Gombosi, K. Powell, D. De Zeeuw, C. Clauer, K. Hansen, W. Manchester, A. Ridley, I. Roussev, I. Sokolov, Q. Stout, and G. Toth. Solution-adaptive magnetohydrodynamics for space plasmas: Sun-to-Earth simulations. *IEEE Computing in Science and Engineering*, 6(2):14–27, 2004.
- [9] X. Jiao, M. T. Campbell, and M. T. Heath. Roccom: An object-oriented, data centric software integration framework for multiphysics simulations. In *Proceedings of 17th Annual ACM International Conference on Supercomputing*, pages 358–368, June 2003.
- [10] O. Karakashian and C. Makridakis. A space-time finite element method for the nonlinear Schrödinger equation: The continuous Galerkin method. *SIAM Journal on Numerical Analysis*, 36(6):1779–1807, 1999.
- [11] K. Keahey, P. Fasel, and S. Mniszewski. PAWS: Collective Interactions and Data Transfers. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing*, pages 47–54. IEEE Computer Society Press, August 2001.
- [12] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Visualization of large datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4):24–33, July/August 2001.
- [13] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, M. Peszynska, R. Martino, M. Wheeler, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar. A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience*, 2004. To appear.
- [14] J. W. Larson, R. Jacob, I. Foster, and J. Guo. The Model Coupling Toolkit. In *Proceedings of International Conference on Computational Science*. Springer-Verlag, April 2001.
- [15] J.-Y. Lee and A. Sussman. Efficient communication between parallel programs with InterComm. Technical Report CS-TR-4557 and UMIACS-TR-2004-04, University of Maryland, Department of Computer Science and UMIACS, 2004.
- [16] MPICH-G2. <http://www3.niu.edu/mpi>.
- [17] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.