ABSTRACT

Title of Dissertation: Flexible and Efficient Control of Data Transfers for Loosely Coupled Components

Shang-Chieh J. Wu, Doctor of Philosophy, 2008

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

Allowing loose coupling between the components of complex applications has many advantages, such as flexibility in the components that can participate and making it easier to model multiscale physical phenomena.

To support coupling of parallel and sequential application components, I have designed and implemented a loosely coupled framework which has the following characteristics: (1) connections between participating components are separately identified from the individual components, (2) all data transfers between data exporting and importing components are determined by a runtime-based low overhead method (approximate match), (3) two runtime-based optimization approaches, collective buffering and inverse-match cache, are applied to speed up the applications in many common coupling modes, and (4) a multi-threaded multi-process control protocol that can be systematically constructed by the composition of sub-tasks protocols.

The proposed framework has been applied to two real world applications, and the deployment approach and runtime performance are also studied. Currently the framework runs on x86 Linux clusters, and porting strategies for multicore x86 processors and advanced high performance computer architectures are also explored.

Flexible and Efficient Control of Data Transfers for Loosely Coupled Components

by

Shang-Chieh J. Wu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

DEDICATION

To my parents, my wife, and my two cats

ACKNOWLEDGEMENTS

To whom it may deserve

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Compared to traditional large scale scientific simulations, the new e-Science paradigm, composed of heterogeneous application components, has many advantages. Integrating well-tested modules that each best model some part of the physical system being simulated, and deploying them within a loosely coupled framework, rather than developing a single tightly coupled monolithic code, is a more efficient and flexible ways to develop large-scale software frameworks.

The loosely coupled approach also makes it easier to investigate an entire physical system broken down into its various components. An example is discussed by Gombosi et. al. [49]: the magnetohydrodynamics (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 a loosely coupled software framework.

The approach also benefits multiscale, multiresolution simulations and models, which are able to describe physical phenomena occurring over various space and time scales such as petroleum reservoir simulation [63], adaptively capturing small-scale noise [5], complex fluids and dense suspension modeling [102], and patch dynamics [55]. Besides, integrated computational power allows 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 [24], which is composed of several U.S. federal agencies and fifteen universities [7], is a software framework for building and coupling weather, climate, and other Earth science models.

However the integration of those components is not easy. For example, different components, because of (1) being developed by different people, (2) the unique characteristics of the phenomena being modeled, or (3) the numerical algorithms employed, might operate on different simulation time and space scales such that the exchange of the data in overlapped regions or at shared boundaries become a serious issue. To obtain correct results, those data must be consistent in both time and space .

The spatial resolution problem is shown in Figure 1.1 and one solution is shown – 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 proper scattering/gathering of data across shared boundaries. The MxN working group in the open Common Component Architecture (CCA) Forum [6, 67, 12, 113] is addressing similar issues.

Traditionally the temporal issue is handled directly in the source codes of the components, through mechanisms such as discarding or merging multiple versions of output data at different spatial or temporal scales, or sampling of fine scale input data for components requiring coarser scale data, and so on. Integration at the source code level has advantages, including guaranteed availability of data objects across components produced or needed at different time scales, and good performance because of the tight coupling among components. Nevertheless, this approach is inflexible, with inherent limitations. For example, replacing some components of the whole system could force source code level modifications of *all* components that must communicate with the new

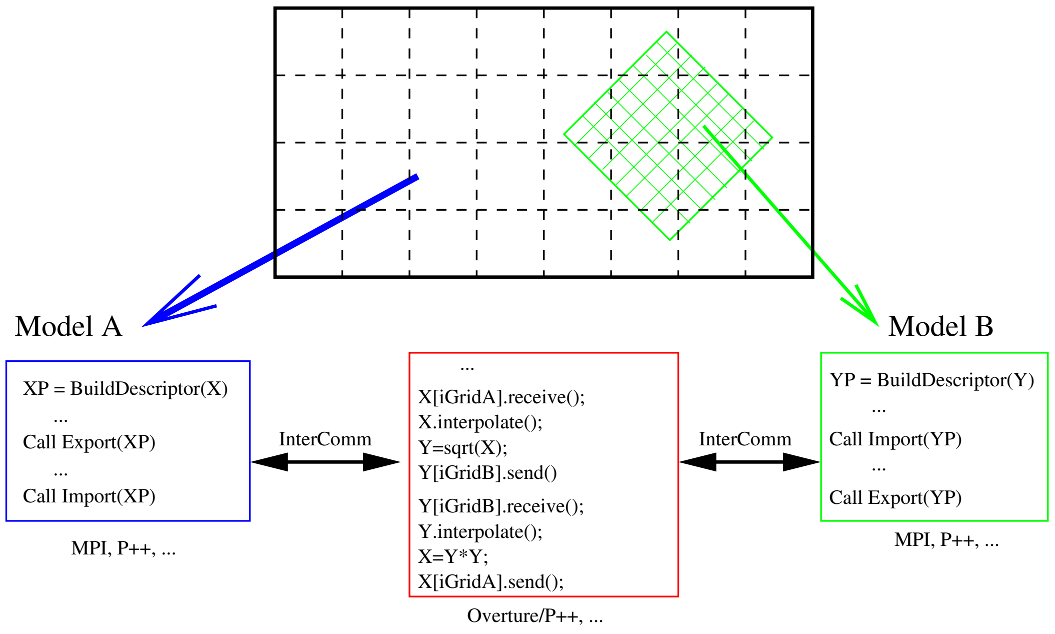


Figure 1.1: Dealing with different spatial resolutions

ones, even if the new components and the replaced ones simulate the same phenomena but with different algorithms. When components are large (in terms of code size), complicated in structure, or perhaps developed by different people or groups, such maintenance work becomes quite difficult.

In this dissertation, to support multiple time scale data exchanges, we describe a low-overhead loosely coupled framework, in which only the following two assumptions are made and arbitrary coordination between the application components is not necessary.

First, instead of identifying data import and export relationships (connections) between application source code components, only the specification of the interface to other components needs to be known by each coupled component. This assumption makes it easy to replace a component by another with the same interface. Once the specification of interfaces, such as communication schedules in InterComm [101], is accomplished, the framework can make runtime decisions about when data transfers will occur between different components. All that is important in such a loosely coupled framework is that data that are required by one component must be delivered in a timely manner from some other component, and the data that are provided by a component but not needed by any other components should not cause any correctness or performance problems elsewhere. For example if a component generates data and there is no other component that requires the data, the unneeded data will be discarded at the framework level and both the behavior and the performance of the entire framework will not be affected.

Second, when data imports or exports occur, rather than requiring the associated simulation timestamps to be periodic, only a monotonically nondecreasing property of the timestamps is assumed. This assumption, which enables efficient memory management in the framework, is based on 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 application components, such as solving nonlinear Schröodinger equation [60]. Visualization of scientific simulation data, for example as shown in [62] may also benefit from variable-sized time intervals. Rather than ingesting generated data at fixed intervals, the visualization program can use a loop to fetch 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 time scale problem would not fix the time when the data exchanges should be performed in advance, but allows all decisions to be made at runtime based on the current overall system state.

The following outlines the remainder of the dissertation.

1.1 Basic Architecture

To couple different components that use different simulation time scales, two approaches are described in Chapter 3. First, rather than embedding all coupling information between export and import components into the application source code, we employ a framework level configuration file, which contains information about the runtime requirements of the components and the connection information between components. Second, whenever timestamped import requests are received, a connection-wise approximate match algorithm is used to determine the matched exported data object.

1.2 Runtime-Based Optimizations

Although the basic approach offers flexibility and versatility for building and deploying large-scale multi-physics simulation systems, due to its runtime-based nature, overall performance might be a concern. We offer two methods for two causes of load imbalance.

The first approach helps to avoid unnecessary buffering in slower export processes, when data export components run more slowly than data import components. By taking advantage of the semantics of collective operations, which ensure that *all* processes in a parallel component must make the same sequence of export (or import) operations, unnecessary data buffering in slower export processes can be identified (and then be avoided) at runtime.

On the other hands, when data import components run more slowly than data export components, the second approach overcomes the available bandwidth and end-to-end latency limitations of the network by using two methods: eager transfer and distributed approximate match.

Transferring data objects that are predicted to be needed in advance, which we call eager transfer, can effectively solve the bandwidth problem. However eager transfer does not solve the latency issue because the basic runtime-based approximate match algorithm needs input data from two different sources (the import and the export components) so that latency between these two data sources always plays an important role in overall performance. We describe a distributed approximate match algorithm that helps in this situation. By piggybacking intermediate results along with the eagerly transferred data objects, no extra latency is introduced and overall performance can be improved significantly.

1.3 Control Protocol

There are two different types of operations in our framework. An *on-demand* data transfer is initiated by a data import program when a user application issues an (on-demand) timestamped data request, and an *eager* data transfer is initiated by a data export program when the user application issues an export request and pattern (of consecutive timestamps on a connection) has been observed. To support both operations simultaneously, the construction of the underlying control protocol is a challenge.

The architecture needs to be designed such that control messages can be initiated by both parties: the import and the export program on a connection. This two-sided initialization approach makes the framework flexible and efficient, but makes the control protocol very complex. To handle this issue, we describe a protocol construction method that is based on validating all possible compositions of smaller protocols for each of the modes.

1.4 Application Studies

Our framework has been applied to two real world coupled simulation scenarios, and the overall performance is described. The first scenario is for coupling of coronal and heliospheric regions around the Sun [86]. It is a one-way coupling: the boundary data, including plasma density, temperature, flow velocity, and magnetic field, are transmitted from the coronal component to the heliospheric component.

The second scenario is part of the coupling of the Lyon-Fedder-Mobarry (LFM) magnetosphere model to the thermosphere-ionosphere-nested-grid (TING) model. It is a two-way coupling: TING receives the electric potential field, the characteristic energy of precipitating electrons, and the flux of precipitation electrons from the LFM coupler component and sends the Hall and Pederson conductances back to the LFM coupler [74].

1.5 Thesis Structure

My thesis is that *it is possible to provide flexible and efficient mechanisms for control of data transfers between coupled parallel programs*, and my contributions are (1) a framework to separate coupling information from application domain computations, (2) a runtime-based approximate match algorithm that uses simulation time stamps, (3) two runtime-based optimization methods to improve overall performance under two load imbalance situations, (4) a control protocol for collective data transfers, (5) investigation of the performance of two real world application in our framework, and (6) porting strategies for advanced high performance architectures.

My dissertation is organized as follows. Chapter 2 is for the related work. Chapter 3 describes an algorithm, *approximate match*, to determine appropriate versions of exported data for requests from different time scale components. Chapter 4 presents a runtime-based optimization method for single program multiple data (SPMD) programs. Chapter 5 introduces a distributed algorithm, *distributed approximate match*, to hide the network latency for eager transfers. Chapter 6 explains the construction of all the underlying control protocols. Chapter 7 studies the behavior of two real world applications when they run in our framework. Chapter 8 outlines an enhanced architecture and investigates the framework architecture on different computational platforms, and the conclusion and future research directions are in Chapter 9.

Chapter 2

Related Work

2.1 Other Parallel Tools

Both Interoperable MPI(IMPI) [46, 47], which is a set of protocols across different MPI implementations, and MPICH-G2 [79], which is a grid-enabled implementation of the MPI v1.1 standard, allow one MPI program to run in the heterogeneous environments which are composed of different architectures and operating systems. These systems mainly focus on heterogeneous integration, and higher level coupling between application components must be handled by the participating components.

Data exchanges between distributed data structures are also offered in other software packages, including Meta-Chaos [35], InterComm [66], Parallel Application Work Space (PAWS) [10, 37], the Model Coupling Toolkit (MCT) [65, 64], Roccom [57], the Collaborative User Migration User Library for Visualization and Steering (CUMULVS) [45], and the MxN working group in the open Common Component Architecture (CCA) Forum [6, 67, 12].

Partitioned Global Array Space (PGAS) language [23, 112, 59] is a programming languages approach to support parallel environments by extending existing languages, such as Co-Array Fortran [82, 83], Unified Parallel C [16, 25, 54, 44], and Titanium [11, 52, 58, 112] (a Java dialect), or designing new ones, such as Chapel [31, 18, 32], and X10 [20, 21, 95, 3, 94, 2].

All of them target mapping of the elements of distributed data structures to the processes in an application component, as well as distributed data exchange between participating (parallel) programs, with the higher level coupling and integration left to the participating application components.

2.2 Coordination Languages

Coordination model, which creates and coordinates execution threads in distributed computing environment, was introduced by Linda [4, 17]. This model has been either spawning new languages like Delirium [73] and Strand [41, 40], or enhancing existing languages like C-Linda [8] and Fortran M [19], to name a few.

Although coordination languages have been extensively and intensively researched since the early 1990's, they has not been widely used for scientific programs. One of the reasons is that either application programmers need to learn and become familiar with a new language, or the stable supporting compilers must become widely available. In addition the inherent complexity of a generalized coordination approach makes the runtime optimization very hard.

Compared to the approaches based on coordination languages, our methodology (1) is independent of the languages the application programs are written in, since it is implemented as a runtime library, and (2) focuses on the general properties of SPMD scientific programs only (not arbitrary programs), which enables runtime optimization that can allow for high performance with low overhead.

2.3 Collective Operation

Collective operations, which must be invoked by a set of processes in a parallel program to perform an operation, are widely used in parallel libraries, such as PVM [100] and MPI [98], either for collective data movement (broadcast, scatter, gather, etc.) or for collective computation (maximum, summation, etc.).

Rather than using collective operations directly, this work takes advantage of the fact that collective operations must be performed by all processes to improve performance. Such optimizations include the functions performed by the representative threads in each participating (parallel) program, as described in Chapter 3 and the buddy-help optimization described in Chapter 4. That is, our framework takes advantage of the semantics of collective operations to decrease the overheads of performing runtime matches, but does not address the design or the implementation of collective operations.

2.4 Client-Side Caching

To reduce the data transfer time between source and sink application components, client-side caching, in which a data object might have multiple copies in the system, is another method widely used in the distributed environments. One issue with caching is coherency between multiple data copies, and various bookkeeping methods have been designed in, for example, the distributed file systems literature, including the Network File System v.4 [97], GPFS [88, 96], and Panasas [103].

Although a form of client-side caching is employed in our framework, as discussed in Chapter 5, coherence is not an issue because our algorithms only move read-only copies of the data, since once the data is exported by an application component it cannot be modified. Most important is that bookkeeping overhead can be completely avoided even when our algorithms create multiple copies of the same data object.

2.5 Protocol Construction

The SPIN [53] model checker uses its own meta-language (Protocol Metalanguage Promela) for system verification, SDL [56] and Cord Calculus [34] offer high-level abstraction for protocol design, and Protocol Composition Logic [30] focus on proving security properties of network protocols. Our protocol construction approach has two distinct properties. First simple tasks are expressed by finite state machines (FSMs), and are very close to source code implementations. The FSMs are so small that it is easy to build and implement them correctly. Second, the composition and validation step can be used to construct larger systems, and interaction between small tasks can be clearly identified.

Chapter 3

Approximate Match

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 chapter we describe an *approximate match* method to making decisions at runtime about which version of exported data are desired. Some properties of approximate match, and our experimental results that measure the overheads incurred by our approach, are presented.

3.1 Basic 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.

```
define region Sr12 | define region Sr0
define region Sr4  | ...
define region Sr5  | for(...) {
...                |   import Sr0
for(...) {         |   ...
  export Sr12      | }
  export Sr4       |
  export Sr5       |
  ...              |
}                  |

Exporter Ap0 code  | Importer Ap1 code
```

Figure 3.1: Example exporter and importer programs

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 3.1.

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/meou/bin/Ap0 2 ...
Ap1 cluster1 /home/meou/bin/Ap1 4  ...
Ap2 cluster1 /home/meou/bin/Ap2 16  ..
Ap4 cluster1 /home/meou/bin/Ap4 4  ...
#
Ap0.Sr12 Ap1.Sr0 REGL 0.05
Ap0.Sr12 Ap2.Sr0 REGU 0.1
Ap0.Sr4  Ap4.Sr0 REG  1.0
```

Figure 3.2: An example configuration file

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 3.1, 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 3.2.

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.

3.1.1 System Architecture

The main components of the runtime system that supports the matching process is shown in Figure 3.3. The configuration file discussed earlier is read in the initialization stage for each program. As shown on the left side of Figure 3.3, 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 3.3, 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 3.2.

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 3.3, 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 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).

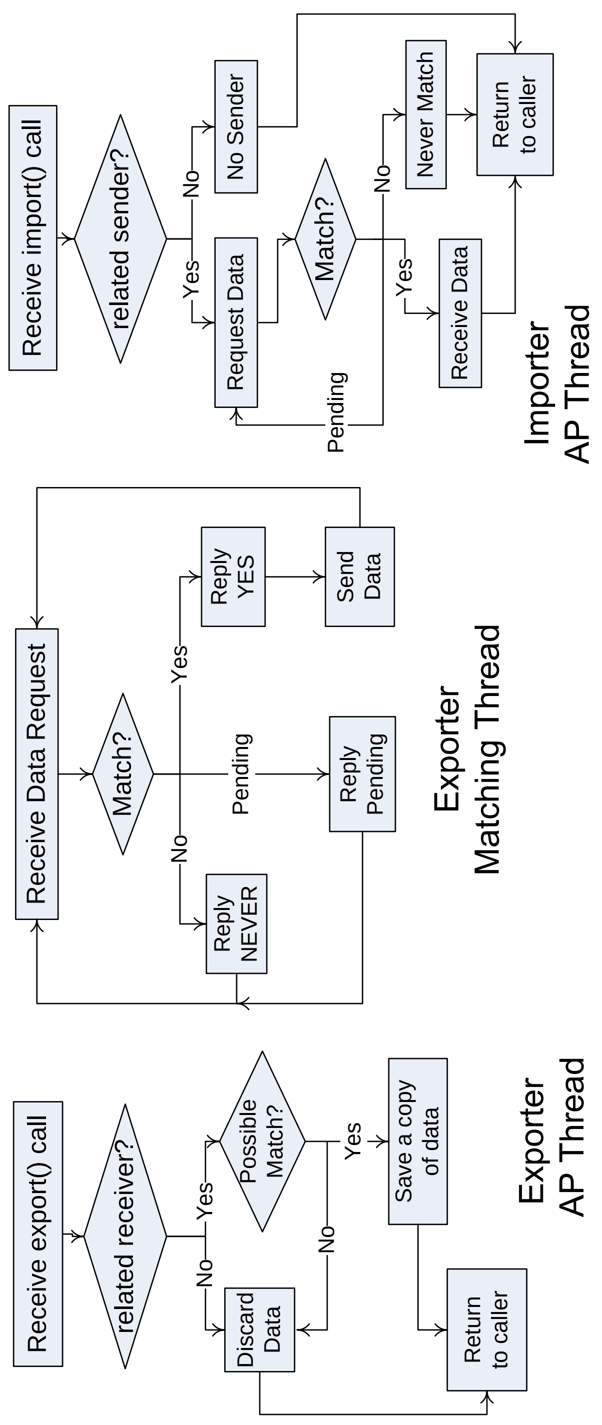


Figure 3.3: Main system components

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 [98], 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 3.3.

3.2 Matching exports to imports

As discussed in Section 3.1, 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 an 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 [7]. 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).

3.2.1 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.

3.2.2 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.

3.2.3 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.

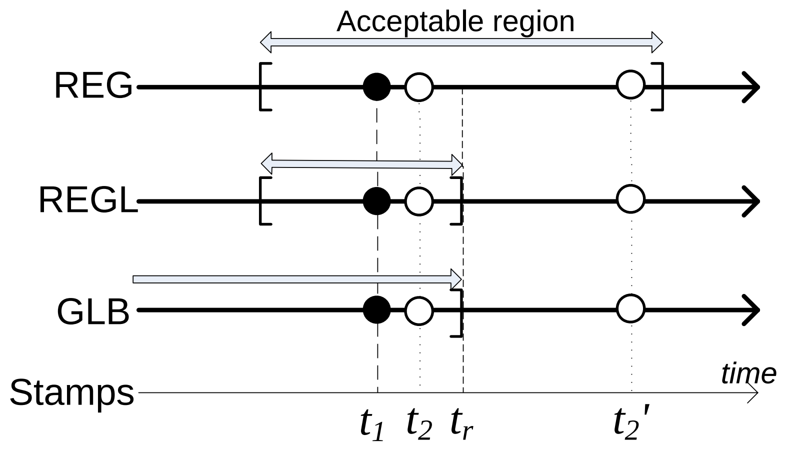


Figure 3.4: Acceptable $\neq$ Matchable

3.2.4 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 3.4. Here we denote exported stamps by filled circles and possibly future stamps by hollow circles. In this example, $\mathbf{A}@t_1$ 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 $\mathbf{A}@t_2$. However a final decision that $\mathbf{A}@t_1$ is really the match can be made if the next exported data object is $\mathbf{A}@t_2'$ rather than $\mathbf{A}@t_2$. 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 3.4.

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.

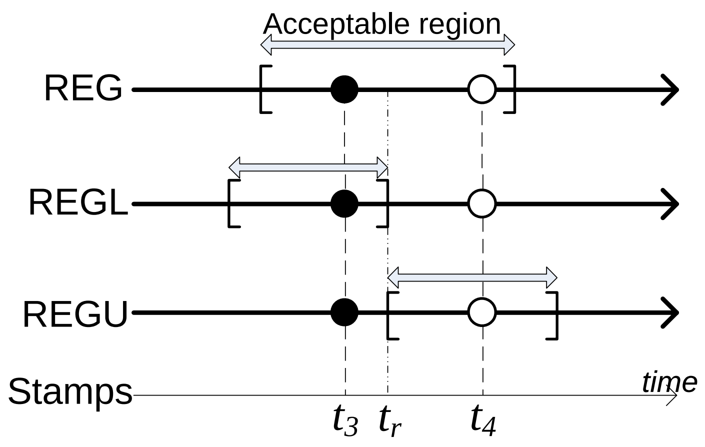


Figure 3.5: REG, REGU and REGL

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 3.5, t_r is the reference stamp and t_3 is the most current stamp. When $\mathbf{A}@t_4$ is exported, it would be the match for the REGU policy and it also assures that $\mathbf{A}@t_3$ is the match for the REG and REGL policies.

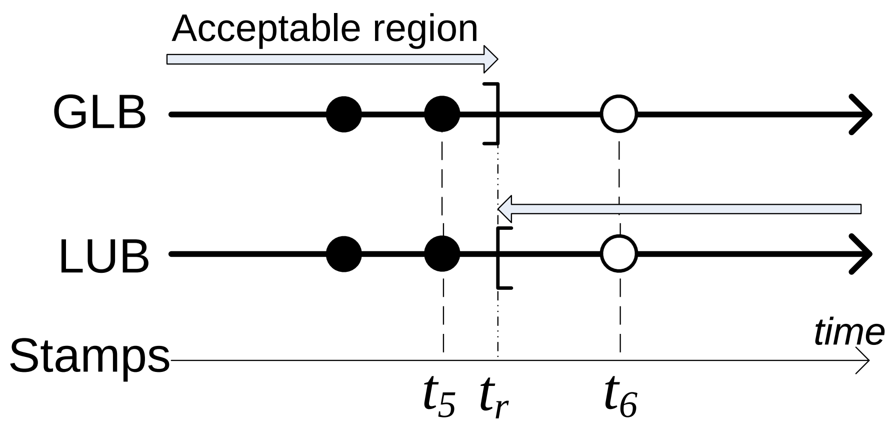


Figure 3.6: LUB and GLB

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 3.6, 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 $\mathbf{A}@t_6$ is exported. Here LUB will match $\mathbf{A}@t_6$ while GLB will match $\mathbf{A}@t_5$.

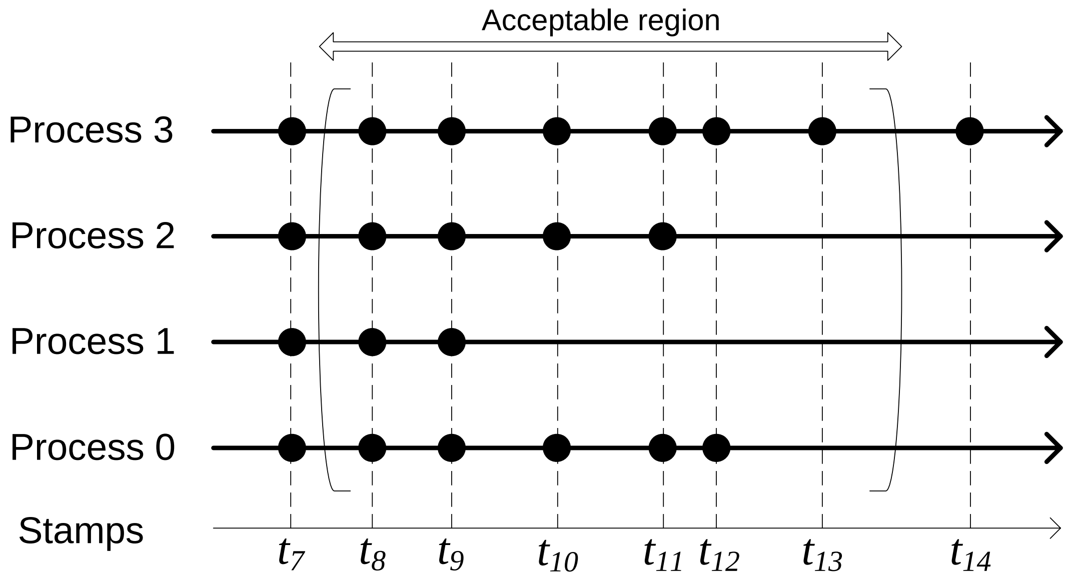


Figure 3.7: Collective correct match

3.2.5 Fast matchings properties

For all three fast type matchings (FASTR, FASTU, and FASTL), if all processes perform the match algorithm independently and no match results are exchange among each other, to keep collective correctness, the matched stamps should be the earliest stamps in acceptable regions. The reason is that, when a request being received, if only one export timestamp t_i is in the acceptable region for the process p_s, but multiple timestamps $t_i, t_{i+1}, \ldots, t_{i+k}$ are in the acceptable region for some faster process p_f, the match decision by process p_f must be the *same* as the one made by p_s, and the only choice is t_i. Figure 3.7 is a snapshot of 4 export processes when a match request is received. The fastest process p_3 has already generated data beyond the acceptable region but the slowest processor p_1 just generates $\mathbf{A}@t_9$. Although p_3 could choose any data objects, the answer from p_1 would be either $\mathbf{A}@t_8$ or $\mathbf{A}@t_9$. Besides if p_1 chooses $\mathbf{A}@t_9$ and another process $p_{very\ slow}$ (not shown) has only **A**@t_8 in its acceptable region (such that the answer from $p_{very\ slow}$ is **A**@t_8), processes p_1 and p_{very_slow} would make different match decisions. Namely no matter how many data objects are in the acceptable region, to keep the collective correctness, *all* processes should choose the earliest one. To put in another way, these *fast* type matchings are really the *earliest* type matchings from the collective correctness viewpoint.

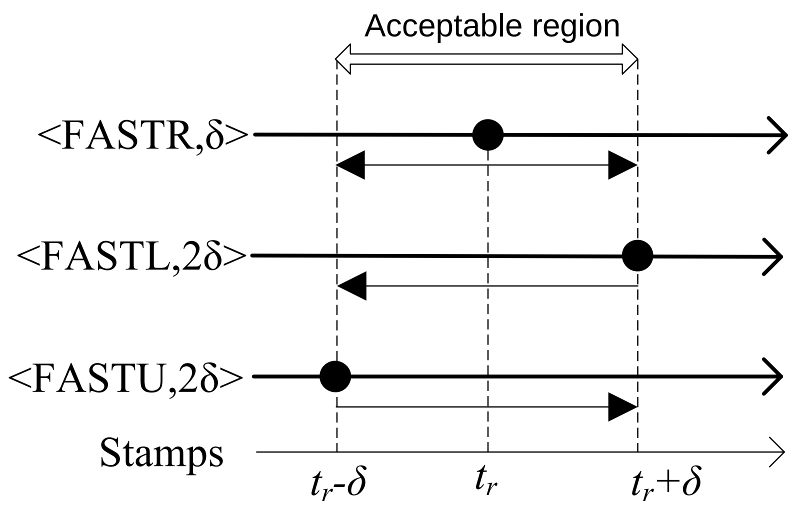


Figure 3.8: Fast matchings

The difference between all three fast type matchings is how to identify the request region based on requested stamps and precisions. For example, the following identifies the same acceptable region: $[t_r - \delta, t_r + \delta]$, as shown in Fig 3.8.

- Request **A**@t_r when the matching criterion is $\langle$FASTR, $\delta\rangle$.
- Request **A**@$t_r + \delta$ when the matching criterion is $\langle$FASTL, $2\delta\rangle$
- Request **A**@$t_r - \delta$ when the matching criterion is $\langle$FASTU, $2\delta\rangle$

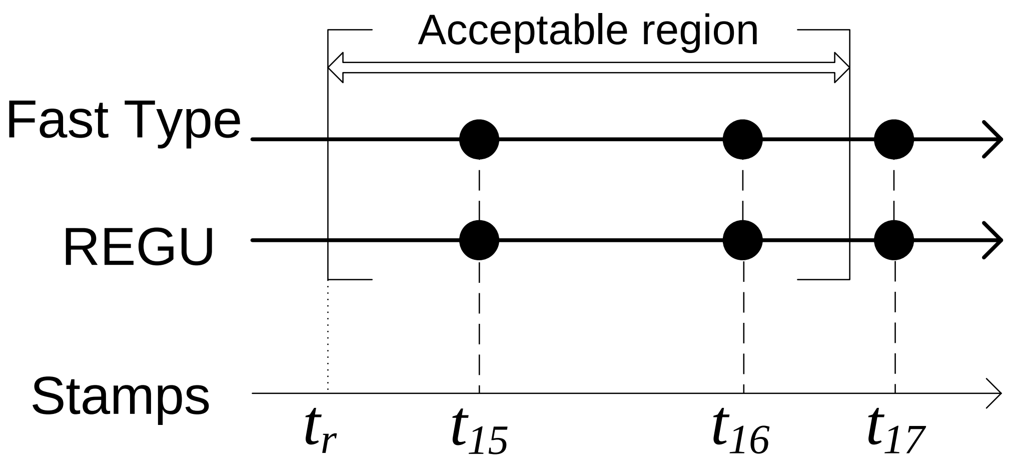


Figure 3.9: Relation between Fast & REGU

There is an interesting relation between those fast type matchings and REGU. If both have the same acceptable region, the same answer will be chosen. As shown in Fig 3.9 **A**@t_{15} will be chosen by both because it is the closest one to base stamp t_r for REGU and is also the earliest stamp in the acceptable region for fast type matchings.

3.3 Experiment 1

Our runtime system is implemented using C++, the C++ Standard Template Library, and Pthreads, and uses the InterComm library [101] to perform the parallel data transfers. To investigate the behavior and performance of the matching techniques, we first 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

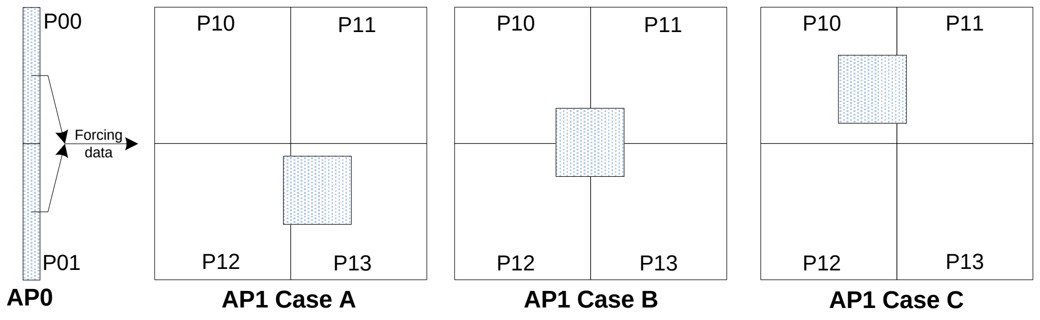


Figure 3.10: Experimental environment

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 32x256 local array, half of the total 32x512 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 $\langle$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 3.10, and measure 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 Figure 3.1, 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 3.3. 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:

```
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).

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 3.1, 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 3.2. From the standard deviations seen in Table 3.1, 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 3.1, 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 is introduced by our techniques. As shown in Table 3.2, although 13% (944μs) 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

	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 3.1: Average execution time (standard deviation), in seconds

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

Table 3.2: Overhead in the slowest process

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 944μs 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 4394μs, as shown in Table 3.2) 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 3.1, 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 P10[1]. As shown in Table 3.2 the slowest process also makes local match requests, and the expensive remote request to the exporter, taking 3468μs, is again hidden.

Interestingly, the overhead for case A (944μs) is greater than that for case B (708μs). In case A, the only work for P10 is as the representative process, but in case B, P10 has 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 3.1, P10 is the slowest process. As shown in Table 3.2, the overhead for the remote request (3703μs) is again hidden behind the work done in the slowest process, which makes a fast local request. In this case, the overhead

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

is 535μs. If we compare the overhead of case B (708μs) against that of case C (535μs), 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.

3.4 Experiment 2

We use the following experiment, by using (1) different sizes of the array, and (2) different ratios between the number of generated data objects and the number of required data objects, to see the execution time under different coupling approaches.

- 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.

- Ap0, running with two processes, is the exporting program for the forcing function (t,x,y). It generates data at every basic time unit (the time units can be arbitrary). Ap1, running with four processes, is the importing program for $u(t,x,y)$. The matching criterion specified in the configuration file is $\langle$REGL,0.05$\rangle$.

- Both programs run on the same cluster of 2.8GHz Pentium 4 machines connected via Gigabit Ethernet, and the underlying operating system is Linux 2.4.21-37.

- Three different sizes (192 *x* 6 , 256 *x* 8, and 512 *x* 16) of array are transferred to program Ap1. In each case, the array is evenly distributed among four processes.

Array u	Coupling Method	Ratio (|Generated| : |Required|)		
		1:1	10:1	20:1
192 x 6	Direct	12.3ms	12.6ms	13.1ms
	Stamp-Based	13.2ms	13.0ms	13.4ms
256 x 8	Direct	22.8ms	23.4ms	23.6ms
	Stamp-Based	24.0ms	23.8ms	23.1ms
512 x 16	Direct	87.2ms	88.3ms	89.7ms
	Stamp-Based	88.9ms	91.3ms	91.3ms

Table 3.3: Average execution time

- Two different coupling approaches (direct coupling and the stamp-based coupling) are used. Also three different ratios (1:1, 10:1, 20:1) between the copies of generated data (from $f(t,x,y)$) and the copies of requested data (for $u(t,x,y)$) are considered. In each case, the stamp-based coupling approach only transfers the requested data; however, the direct coupling method transfers *all* of the exported data but only uses the desired one to perform the computation.

- For the direct coupling, 100, 1000, and 2000 copies of exported data are transferred when the ratios are 1:1, 10:1, and 20:1 respectively. For the stamped-based coupling, 100 copies of exported data are transferred for all three ratios.

- Each configuration was run three times, 100 copies of exported data are used to perform the computation, and the average execution time per iteration from the slowest process in program Ap1 is shown in Table 3.3.

We make the following observations. First, for the direct coupling, the execution time increases very little when the ratio goes from 1:1 to 20:1 for all different array sizes. The difference among them is the time to transfer unrequested data objects under Gigabit Ethernet. For example, to transfer 19 copies of 512 *x* 16 arrays, only 2.5ms (89.7 - 87.2) is needed. Also, for the stamp-based coupling, the execution time is almost the same for all different array sizes. In fact, comparing two different coupling approaches, we claim the stamp-based approach does not introduce lots overhead for the execution time for *all* scenarios.

Second, in this experiment, the execution time is dominated by the domain data computation for *all* scenarios. For example, if the ratio is 20:1 and the direct coupling is used, the execution time is increased from 23.6ms to 89.7ms when the the array size is increased from 256 *x* 8 to 512 *x* 16. By combining with previous result (transfer 19 copies of 512 *x* 16 arrays need 2.5ms), we know most of the time is spent on the computation.

Chapter 4

Collective Optimization (Buddy Help)

The loosely coupled framework described in Chapter 3 offers flexibility and versatility for building and deploying large-scale multi-physics simulation systems. It describes a temporal consistency model in which each exported data object must be buffered by the runtime system implementing the model, until there is no possibility that an object will be requested by an importing component. That can be confirmed by determining that either there is no importing component for objects of that type or because importer requests that have already been processed can be shown to ensure that the object in question cannot be requested.

Although this approach ensures the correctness of the data exchange mechanism, overall system performance may suffer from unnecessary buffering, when one process in a data exporting (parallel) component performs the collective export operation early relative to the other processes (i.e. it is the first process to execute the export runtime library call). In that case, other processes can use the information produced in resolving the call in one process for later calls in other processes in the exporting parallel component.

We focus on the the temporal consistency issue here by taking advantage of the semantics of collective operations, which ensure that *all* processes in a parallel component must make the same sequence of export (or import) operations (similar to the required behavior of parallel programs that use MPI collective operations [98]).

Collective operations are commonly used in many single program multiple data (SPMD) parallel program implementations, and require that (1) the same code is running on all processes, (2) the dataset is partitioned across the multiple processes, and (3) each process performs computation on the part of the data object it owns. Moreover, collective operations, such as broadcast, barrier, reduce, etc., require *all* processes in the *same* program to execute the same function with appropriately matching parameters. These operations are well supported in popular parallel libraries such as PVM [100] and MPI [98], and play important roles in SPMD programs. Performance studies have shown that some parallel programs spend more than 80% of their interprocessor communication time in collective operations [89].

Data exchange between shared or overlapping regions in different coupled simulation components can be viewed as a collective operation, where the data to be transferred spans both multiple processes in a single component and the processes in two separate components. That is because the exchange is not complete until *all* involved processes transfer their share of the data (however it does not require that all processes transfer data at the same time, meaning no barrier synchronization is required). In addition, the collective operation semantics guarantee that all processes in the same exporting component must make the *same* decision about which copies of the generated data should (and should not) be transferred to the corresponding importing program(s). When some of the processes in the data exporting component run more slowly than other others, perhaps because of imperfect load balancing within the component or for other application-specific reasons, those slower processes can be sped up if the decision about which transfers to make are performed by the fastest process in the component (the one that executes the export call first).

4.1 Collective Semantics

Compared to traditional collective operations, such as broadcast (copying data from one to a group of processes) and reduce (aggregating with some binary operation data supplied by a group of processes) in PVM [100] and MPI [98], data transfers in our framework also exhibit collective properties. This means that *all* processes in the same program must execute the same export (or import) operations in the same order (but not necessarily at the same time), with appropriately matching parameters. Formally the following property must always hold in our framework:

> **Collective Property** If one process transfers (exports or imports) data with timestamps $t_1, \ldots, t_n$ during execution, all other processes in the same program must also transfer data with those timestamps, in the same order.

To support and monitor collective behavior at runtime, our framework implementation still employs an extra process in each program, called the representative (or *rep* for short), to act as a low-overhead control gateway [111]. For example when the *rep* in an exporting program receives a request from an importing program, it (1) forwards the request to all processes in the exporting program, (2) collects the responses from all processes, (3) combines all responses to produce the final answer to the request, and (4) sends back the final answer to the requester (to the *rep* of the importing program).

The legal set of responses from all the processes aggregate into one of the following five cases: all `MATCH`, all `NO MATCH`, all `PENDING`, a mixture of `PENDING` and `MATCH`, or a mixture of `PENDING` and `NO MATCH`. Additionally when all or only some responses are `MATCH`, all the matched timestamps must be the same.

It is incorrect for some of the responses to be `MATCH` and some to be `NO MATCH` for the same request, because only those processes whose responses are `MATCH` try to transfer data, which is a clear violation of collective property. It is also incorrect if the matched timestamps from those `MATCH` responses are not the same, because those processes will try to transfer data with different timestamps and collective property would not hold again.

The collective property is maintained if all responses are the same. Interestingly, it is still legal if the collective responses are a mixture of `PENDING` and `MATCH` or a mixture of `PENDING` and `NO MATCH`. This situation means that some processes are running more slowly than others (e.g., either because of load imbalance or because of other application-specific properties), such that when receiving forwarded requests the *best* match cannot yet be decided (so their responses are `PENDING`s.) Based on the guarantee (because of the collective nature of export and import operations) that those slower processes will make the same decisions as their faster peers, the answer sent by the *rep* is `MATCH` if the collective responses are a mixture of `PENDING` and `MATCH` and is `NO MATCH` if the collective responses are a mixture of `PENDING` and `NO MATCH`.

4.2 Collective Optimization (Buddy-Help)

When the collected responses are a mixture of `PENDING` and `MATCH`, more can be done than just determining the *rep* answer for the `MATCH`. If the *rep* then sends the final answer (`MATCH` in this case) back to the slower processes in that program, those processes then know whether or not a data object they will export in the *future* should be buffered by the framework (buffered only in the case that it is a match), even *before* the data is exported by that process.

Because the overall model requires that timestamps for requests form an increasing sequence, as in many timestep-based numerical algorithms, the generated data objects are buffered only if the framework cannot decide whether the data objects are needed or not – either because they already have passed the latest timestamp in the acceptable region, or because the best match still cannot be decided.

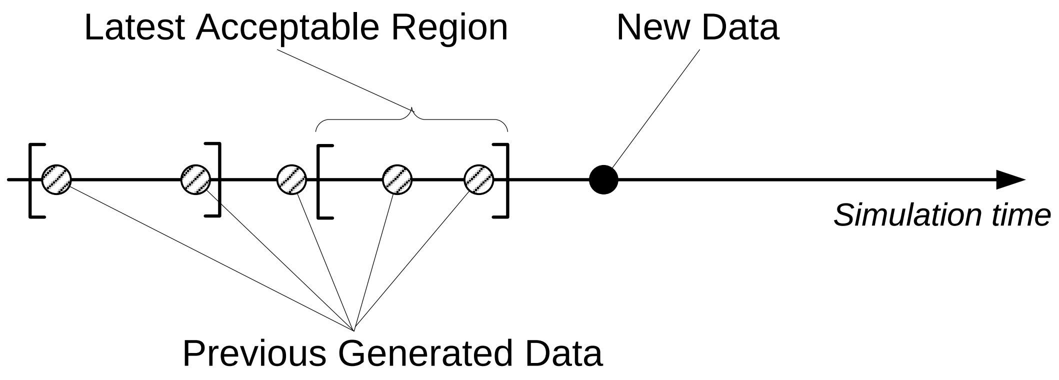


Figure 4.1: Slower Importer

When the data importing program runs more slowly than the related exporting counterpart, as shown in Figure 4.1, the timestamp of a newly generated data object will *pass* the latest acceptable region which is identified by the last request timestamp and a user-defined tolerance. Buffering of this generated data object is necessary because it might be a match for future requests. Although the buffering operation may be time-consuming when the data size is large, the overall application performance will not be affected much because the data exporting program is not the slowest component in the whole system.

When the data exporting program runs more slowly than the related importing counterpart, the buffering of newly generated data is a performance concern. If the timestamp of the newly generated data object is outside *all* of the acceptable regions, buffering is not needed because it is beyond the user-defined tolerance.

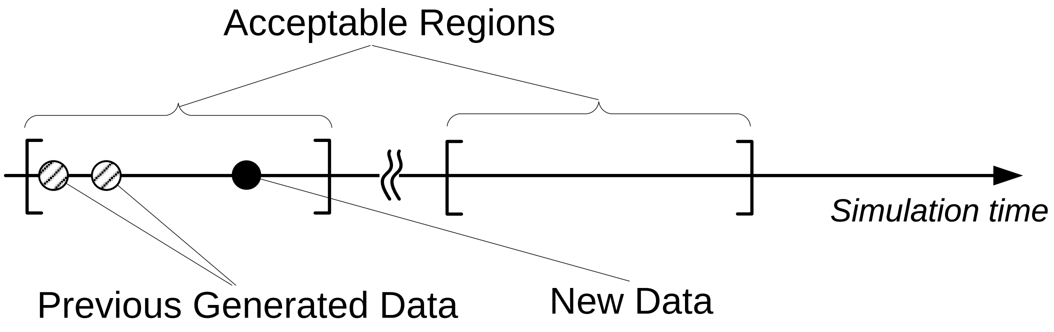


Figure 4.2: Slower Exporter

However if the new generated data object, call it $\mathbf{A}@t$, (a distributed array $\mathbf{A}$ with simulation timestamp t), is in one of acceptable regions R, as shown in Figure 4.2, in general buffering is necessary because $\mathbf{A}@t$ *might* be the match. If the *next* generated data object $\mathbf{A}@t'$ is outside region R, then $\mathbf{A}@t$ is confirmed as the match for this region R, and the buffering step was indeed required. However, if $\mathbf{A}@t'$ is also located in region R, $\mathbf{A}@t'$ would be a better match, and the system could free the buffer for object $\mathbf{A}@t$.

It is no surprise that much unnecessary buffering can occur in the framework if multiple objects are exported that fall in one acceptable region – which can easily occur in coupling physical simulation components that act on different time scales. Formally, if data objects $O_1, \ldots, O_{n(i)}$ are located in the acceptable region R_i, and the time for buffering (and freeing) object O_j is t_j, the time T_i spent on that unnecessary buffering in region R_i is:

$$T_i = \sum_{k=1}^{n(i)-1} t_k \tag{4.1}$$

If a total of N requests are received (so that the acceptable regions are $R_1, \ldots, R_N$) during the program execution, and all acceptable regions are mutually disjoint ($R_i \cap R_j = \emptyset$, $i \neq j$), the total time T_{ub} spent on unnecessary buffering is:

$$T_{ub} = \sum_{i=1}^{N} T_i = \sum_{i=1}^{N} \sum_{k=1}^{n(i)-1} t_k \tag{4.2}$$

Compared to currently used ad-hoc tightly coupled approaches, approximate matching and buffering of generated data are two extra tasks that our framework must perform, and it is clear that T_{ub} plays an important role in overall performance when the data exporting program runs more slowly than the related importing counterpart.

One way to decrease T_{ub} is taking advantage of collective property described previously. More precisely, if for a given request the collective responses in the *rep* are a mixture of `MATCH` (or `NO MATCH`) and `PENDING`, the rep not only sends the final answer (which is `MATCH` or `NO MATCH`) to the requester, but also sends it to those processes whose responses are `PENDING` (we call this *buddy-help*.) In this way those slower processes can know the *right* match for this request, and avoid unnecessary buffering of data objects that cannot possibly be a match, even *before* the data objects are generated by export operations in those slower processes.. Decreasing T_i (and therefore T_{ub}) in those slower processes matters for overall performance, because the processes that benefit from buddy-help are the slowest processes in the slower program – and therefore are the performance limiting factor for that pair of coupled programs.

One interesting side effect of the buddy-help optimization is that if each time-step iteration in a data exporting programs performs computational tasks and a slower process p_s starts to get buddy-help during the jth request, T_k in process p_s will form a non-increasing sequence for $k \geq j$. We use a micro-benchmark described in the next section to explain that behavior more completely.

4.3 Experiment

The complexity of the framework makes it difficult to measure the benefits from the optimization methods we have just described for general scientific programs, so we have designed a micro-benchmark to measure the potential performance improvements from the optimizations. The benchmark configuration is as follows:

- Solve $u_{tt} = u_{xx} + u_{yy} + f(t,x,y)$, a two dimensional diffusion equation with a forcing function $f(t,x,y)$ which can be viewed as the external input for $u(t,x,y)$.
- Program U, which computes $u(t,x,y)$, owns a 1024 x 1024 array which is evenly distributed among the participating processes.
- Four configurations are considered. Program U has either 4, 8, 16, or 32 processes.
- Program F, which computes $f(t,x,y)$ has four processes p_1, p_2, p_3, and p_s, each of which is responsible for a 512 x 512 array.
- There is no data exchange between process p_s and p_i with i=1,2, and 3.
- Data of size 1024 x 1024 is transferred from $f(t,x,y)$ to $u(t,x,y)$ with matching policy REGL and precision 2.5. Program F is the exporter program and program U is the corresponding importer program.
- Process p_s performs extra computational work to make it the slowest process in program F, and it may also run more slowly than any of processes in program U

(with respect to matching export/import calls).

- Processes p_i in F with i=1,2, and 3 run faster than any of the processes in program U for all four configurations of U.

The experiment was performed on a cluster of Pentium 4 2.8GHz machines connected via Gigabit Ethernet. The execution times for exporting data in the slowest process p_s of program F are shown in Figures 4.3, 4.4, 4.5, and 4.8. Here each run performed 1001 data exports, and to simulate multi-resolution coupling, one out of every twenty exported data objects end up being transferred to program U (those are the ones that matched). The results are from six runs for each configuration. The time for exporting data was measured because it shows the effectiveness of the buddy-help optimization.

Figure 4.3 shows the case when the importer program U has only 4 processes and is running more slowly than the exporter program F. In this case every exported data object will be saved in the framework buffer because there is no way to know which exported data objects might be needed for a match – therefore the execution time for all 1001 data exports should be similar and Figure 4.3 confirms that, except for early iterations (where the time is 8% greater) and after 600 iterations (where the time is 4% less). The extra 8% is a result of initialization of the framework and its underlying data structures. The 4% decrease in later iterations is likely the result of less congestion in the network and a lighter workload in the framework, because the times shown in Figure 4.3 are from the slowest process p_s in program F, and after 600 iterations all other processes p_i, i = 1, 2, and 3, in program F have already completed.

By keeping the size of the distributed data array fixed (1024 x 1024), the program U runs faster as the number of importer processes increases – because less computation is performed for each importing process. Figure 4.4 shows the case when the importer

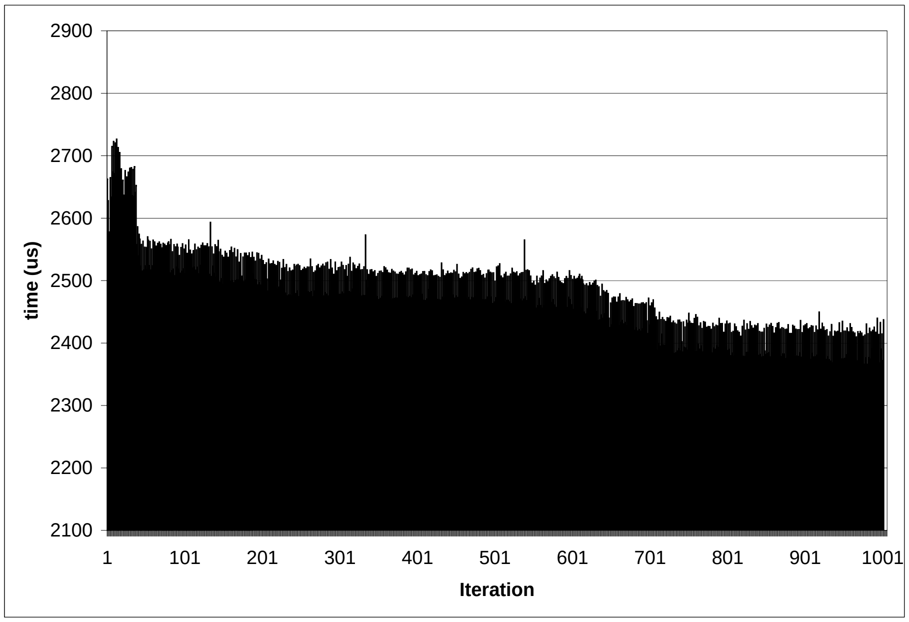


Figure 4.3: Coupled with 4 Importer Processes

program U has 8 processes, but it is still running slower than the exporter program F. The result is very similar to the case of 4 processes.

The results start to become more interesting when program U has 16 processes, as shown in Figure 4.5. Here U catches up to process p_s in program F, and a typical scenario is shown in Figures 4.6 and 4.7, where **D**@t denotes the distributed data **D** at the simulation time t. Process p_s receives the first data request **D**@20 after exporting 14 copies of **D** in line 5. Because the matching policy is REGL (described in Section 3.2.3) and the tolerance is 2.5, the acceptable region is [17.5, 20] and the reply from process p_s is {**D**@20, `PENDING`, **D**@14.6}, which means that for the request **D**@20, the answer is `PENDING` and the current latest exported data is **D**@14.6. Once the answer is generated,

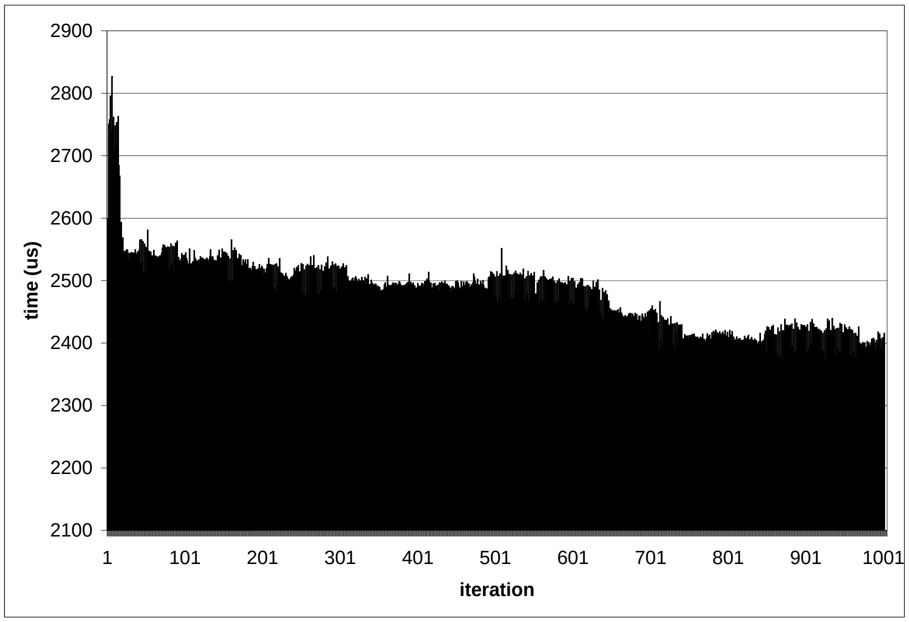


Figure 4.4: Coupled with 8 Importer Processes

process p_s knows immediately that any version of **D** exported with a timestamp less than 17.5, as in lines 10-11, can be discarded it is not in the acceptable region.

Process p_s then receives the buddy-help message in line 8, which is the reply `MATCH` and the match **D**@19.6, for the earlier request from the fastest process in F. Once the match **D**@19.6 has been determined, even though the export with that timestamp has not been occurred in process p_s yet, any future data exported with a timestamp less than 19.6, as in lines 10-13, can be discarded. This shows the benefit of buddy-help.

This pattern occurs again after the match **D**@19.6 is produced by process p_s. Because extra memory allocations and deallocations `memcpy`s are performed by process p_s, as shown in the beginning part of Figure 4.5, the processes in program U have a chance

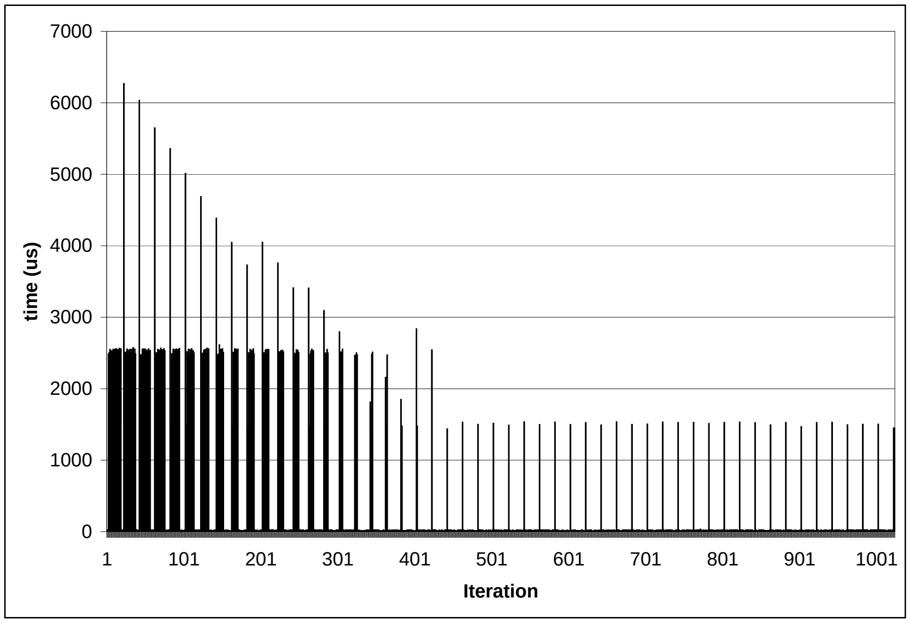


Figure 4.5: Coupled with 16 Importer Processes

to catch up so that between successive data transfers new data requests will show up earlier, and the number of skipped data copies increases (so T_i starts to decrease.) For example 4 `memcpy`s are skipped in lines 10-13 and then 7 `memcpy`s are skipped in lines 26-29 of Figures 4.6 and 4.7. Eventually the optimal state, as shown in Figure 4.9, is reached and maintained, where only the matched data are buffered in the framework. The optimal state has the following characteristics:

- For each matched and then transferred data object, a corresponding buddy-help message will be received early enough by a slow exporter process. (In the example, that is process p_s.)

1	export **D**@1.6, call memcpy.
2	export **D**@2.6, call memcpy.
3	⋮
4	export **D**@14.6, call memcpy.
5	receive request for **D**@20,
6	reply {**D**@20, `PENDING`, **D**@14.6}.
7	remove **D**@1.6, ⋯, **D**@14.6.
8	receive buddy-help {**D**@20, `YES`, **D**@19.6}.
9	remove **D**@16.6.
10	export **D**@15.6, *skip* memcpy.
11	export **D**@16.6, *skip* memcpy.
12	export **D**@17.6, *skip* memcpy.
13	export **D**@18.6, *skip* memcpy.
14	export **D**@19.6,
15	call memcpy,
16	send **D**@19.6 out.
17	export **D**@20.6, call memcpy.

Figure 4.6: A Typical Buddy-Help Scenario

```
18  export D@21.6, call memcpy.
19              ⋮
20  export D@31.6, call memcpy.
21  receive request for D@40,
22      reply {D@40, PENDING, D@31.6}.
23      remove D@19.6, ···, D@30.6.
24  receive buddy-help {D@40, YES, D@39.6}.
25      remove D@31.6.
26  export D@32.6, skip memcpy.
27  export D@33.6, skip memcpy.
28              ⋮
29  export D@38.6, skip memcpy.
30  export D@39.6,
31      call memcpy,
32      send D@39.6 out.
33  export D@40.6, call memcpy.
34              ⋮
```

Figure 4.7: A Typical Buddy-Help Scenario

- For slow exporter processes, the framework can determine which versions (timestamps) of exported data objects will be requested by the corresponding importer program even before those data are exported, and only the matched data objects are buffered in the framework.
- The remaining exported data that is not a match will not be saved by the framework. Namely T_i is equal to 0 once the optimal state is entered.

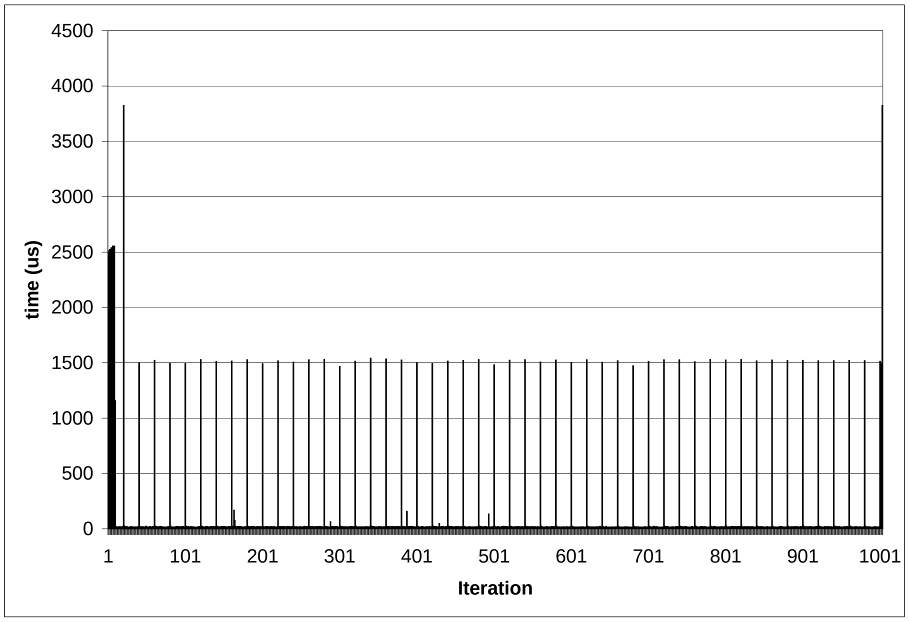


Figure 4.8: Coupled with 32 Importer Processes

By keeping the exporter program and its participating processes fixed, the exporter program can reach the optimal state earlier if the importer program is running faster. The reason is that when the importer program is running faster, the related exporter processes

```
              ⋮
export $\mathbf{D}@t_\alpha$,
    call memcpy,
    send $\mathbf{D}@t_\alpha$ out.
export $\mathbf{D}@t_\beta$, *skip* memcpy.
              ⋮
export $\mathbf{D}@t_\gamma$, *skip* memcpy.
export $\mathbf{D}@t_\delta$,
    call memcpy,
    send $\mathbf{D}@t_\delta$ out.
              ⋮
```

Figure 4.9: Optimal State

will receive the data requests earlier, and based on the information provided by buddy-help, unmatchable data can be identified earlier and more `memcpy`s can be skipped (so T_i starts to decrease.) This claim can be validated from the data in Figures 4.5 and 4.8. Both configurations have the exact same exporter program and around 400 iterations are needed to reach the optimal state when the importer program U has 16 processes, but only around 25 iterations are needed to reach the optimal state when the importer program U has 32 processes.

The performance benefits of avoiding unnecessary buffering from the buddy-help optimization depend on the ratio of the size of the acceptable region to the inter-arrival time between successive importer match requests. Consider the following example. If the matching policy is REGL and the precision is 5.0, the result *with* buddy-help is shown in Figure 4.10. After receiving the request for $\mathbf{D}@10.0$, the acceptable region would be identified as [5.0, 10.0], and the exported data object $\mathbf{D}@4.6$ in line 8 will not saved because it is outside of the acceptable region. However all exported data in lines 9-11, which are within the acceptable region, are not saved either because they are not the match, **D**@9.6, which became known via the buddy-help mechanism. Figure 4.11 shows a different result *without* buddy-help for the same configuration. In that example, whenever *acceptable* data is exported, as shown in lines 9-18, the new exported data object *is* the best current candidate for a match, so must be saved, and the previous best candidate can be safely deleted. The final match will be identified only after a data object is exported *outside* the acceptable region, which is **D**@10.6 in lines 19-21.

The buddy-help message, which is the answer from the fastest process in an exporter program, plays an important part here – the farther the fastest process progresses, the more help the slowest process can get. However the processes in most scientific data-parallel programs will not usually get out of sync by too much, because data exchanges between the processes within the program occur relatively frequently, loosely synchronizing the processes. But if (1) at least one of the processes p_f acts as a data source, which receives external data and performs its computation without using data from its peer processes, and (2) non-blocking data transfers (such as MPI_Isend) or advanced facilities such as Remote Direct Memory Access (RDMA) over InfiniBand [72] are used for intra-program communication such that multiple copies of the computed data objects (with different timestamps) can be kept in the same program, then the fastest process has an opportunity to stay ahead and to help other, slower peer processes.

```
1   export D@1.6, call memcpy.
2   export D@2.6, call memcpy.
3   export D@3.6, call memcpy.
4   receive request for D@10.0,
5       reply {D@10.0, PENDING, D@3.6}.
6       remove D@1.6, ···, D@3.6.
7   receive buddy-help {D@10.0, YES, D@9.6}.
8   export D@4.6, skip memcpy
9   export D@5.6, skip memcpy.
10              ⋮
11  export D@8.6, skip memcpy.
12  export D@9.6,
13      call memcpy.
14      send D@9.6 out.
15  export D@10.6
16              ⋮
```

Figure 4.10: With Buddy-Help

1	export **D**@1.6, call memcpy.
2	export **D**@2.6, call memcpy.
3	export **D**@3.6, call memcpy.
4	receive request for **D**@10.0,
5	reply {**D**@10.0, `PENDING`, **D**@3.6}.
6	remove **D**@1.6, $\cdots$, **D**@3.6.
7	export **D**@4.6, *skip* memcpy.
8	export **D**@5.6, call memcpy.
9	export **D**@6.6,
10	call memcpy,
11	remove **D**@5.6.
12	export **D**@7.6,
13	call memcpy,
14	remove **D**@6.6.
15	$\vdots$
16	export **D**@9.6,
17	call memcpy,
18	remove **D**@8.6.
19	export **D**@10.6,
20	call memcpy,
21	send **D**@9.6 out.
22	export **D**@11.6
23	$\vdots$

Figure 4.11: Without Buddy-Help

Chapter 5

Eager Transfer and Distributed Approximate Match

Earlier we have described a loosely coupled component-based framework that offers great flexibility and versatility for building and deploying large-scale multi-physics simulation systems. However, the performance of that framework is very sensitive to the available bandwidth and end-to-end latency of the network connecting the various components of a coupled simulation.

In this chapter, we describe and analyze two methods, eager transfer and distributed approximate match, to deal with the network bandwidth and latency. Transferring predicted data in advance, which we call eager transfer, can effectively solve the bandwidth problem. Our analysis and experiments show that, on average, the lower the available network bandwidth, the more time is saved by eager transfer.

However, eager transfer does not solve the network latency problem. The reason is that, although the data transfer time is hidden behind the applications computation, a round-trip time between components for control messages, to decide which data object is the one to be transferred, is still needed. In short network latency still plays an important role in overall performance, even with eager transfer.

The origin of this problem is that the approximate match algorithm has two input sources – one is from the data export component (history information about previous satisfied requests from an import component) and the other from the data import component (the current request). The approximate match algorithm in Section 3.2 runs only in the data export component, and a round-trip message exchange is needed for each data request, even if the matched data object has *already* been eagerly transferred to the requester.

Running the algorithm only in the import component is not a good approach either, because whenever a new copy of a data object is exported, a message needs to be sent to the data import component even if this copy of data is not required. That is, no matter where the algorithm runs, compared to transferring the requested data as in the direct coupled approach, extra network latency is always introduced.

Here we introduce another method, distributed approximate match, which is a distributed version of our previously described approximate match algorithm, to handle the latency issue. It contains two parts: inverse approximate match and range check. The inverse approximate match runs in the component that supplies the data, and the range check runs in the component that consumes the data. The new distributed algorithm has the same functionality as our previous approximate match algorithm, but does not introduce any extra latency to transfer the required data. Our experimental results show that by combining both methods, the overall performance of the framework can be improved significantly.

5.1 Approximate Match-based Architecture

It is often the case that some coupled components run faster than others, and overall system performance is determined by the slowest component. When data import components run more slowly than the corresponding exporters, the runtime performance of our existing framework is very sensitive to both network bandwidth and latency between participating components.

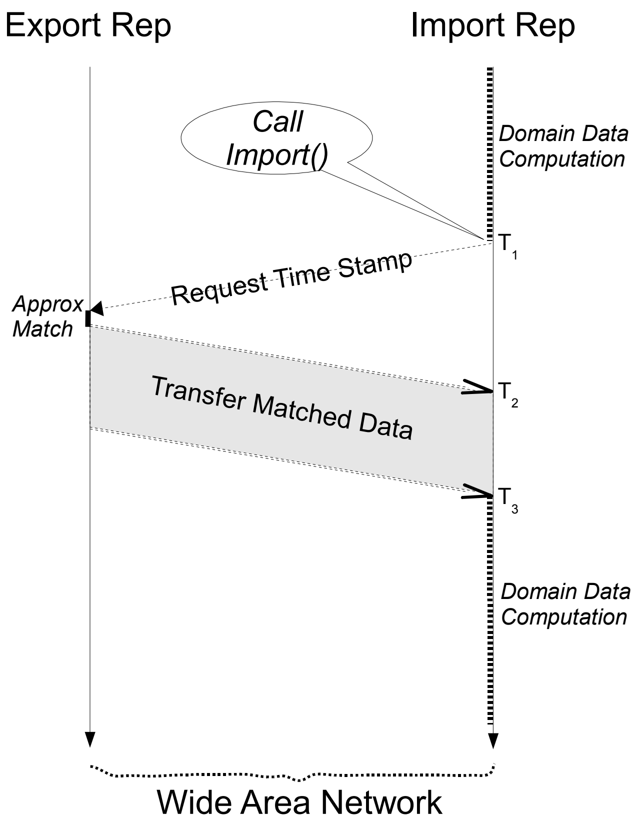


Figure 5.1: A Scenario for a Slow Data Import Component, Original Approach

A typical scenario for the representative (*rep*) threads is shown in Figure 5.1, in which the data import component needs to wait at least one network round-trip time $(T_2 - T_1)$ before starting to receive the required matching data. More formally, if T_{export}, δ_{ie}, δ_{ei}, M and BW are the time spent by the export program, the latency between import and export components, the latency between export and import components, the size of the matched data object, and average network bandwidth, respectively, the data import time T_{import} can be expressed as follows,

$$T_{import} = T_{export} + \delta_{ie} + \delta_{ei} + \frac{M}{BW} \tag{5.1}$$

Normally the time $\delta_{ei} + \frac{M}{BW}$ is required to transfer *any* data object of size M from the data export component to the import one via a network with bandwidth BW, and since the approximate match algorithm runs on the data export components, the extra δ_{ie} must be included in Equation 5.1. It is a natural choice to run the algorithm in the data export component, because the timestamps of multiple export data objects may be required to make correct matching decisions. On the other hand, if the algorithm runs at the data import component, whenever a new copy of a data object is exported, its timestamp needs to be forwarded to the import component, even if the data with this timestamp might never be requested. When the latency between components is relatively high, as for components running at two different sites connected via a wide-area network (e.g., a Grid computing scenario), the cost could be very high.

More formally, when a data import component runs more slowly than an export component on the other end of a connection, if $T_{compute}(i)$, $T_{import}(i)$, $T_{export}(i)$, $\delta_{ie}(i)$, $\delta_{ei}(i)$, $M(i)$ and $BW(i)$ are the user application computation time, data import time, the time spent by the export program, the latency between import and export components, the latency between export and import components, the size of matched data objects, and average network bandwidth, respectively, the total execution time of the data import component, T_{total}, can be expressed as follows, for each iteration i of the total N iterations.

$$\begin{aligned} T_{total} &= \sum_{i=1}^{N} (T_{compute}(i) + T_{import}(i)) && (5.2) \\ &= \sum_{i=1}^{N} T_{compute}(i) + \sum_{i=1}^{N} T_{import}(i) && (5.3) \end{aligned}$$

$$= \sum_{i=1}^{N} T_{compute}(i) + \sum_{i=1}^{N} (T_{export}(i) + \delta_{ie}(i) + \delta_{ei}(i) + \frac{M(i)}{BW(i)}) \quad (5.4)$$

$$= \sum_{i=1}^{N} T_{compute}(i) + \sum_{i=1}^{N} T_{export}(i) + \sum_{i=1}^{N} (\delta_{ie}(i) + \delta_{ei}(i)) + \sum_{i=1}^{N} \frac{M(i)}{BW(i)} \quad (5.5)$$

$$\geq \sum_{i=1}^{N} T_{compute}(i) \quad (5.6)$$

As shown in Equation 5.5, when the data import component runs more slowly than the export component, the total execution time T_{total} is composed of four parts (1): application computation time, (2) the time for performing approximate match in export programs, (3) round trip delay for the match, and (4) network transfer time for the data. In addition, as shown in Equation 5.6, T_{total} is bounded from below by the application computation time, and equality holds for an *unattainable* network between import and export components that has infinite bandwidth and zero delay. To approach this theoretical lower bound in a *real* environment, our framework is enhanced as described in the following sections.

5.1.1 Eager Transfer Approach

One way to deal with finite network bandwidth is to transfer data objects in advance, if they can be predicted to be needed (based on the matches already made for that connection), which is what we call an *eager transfer*, and Figure 5.2 shows a typical scenario for this approach.

1. When a new copy of an object with a timestamp that is predicted to be needed is exported at time T_{01} (in the export component), a request-response protocol is used, between both the the reps in the export and import components, as well as between the import rep and all the processes in the import component, to approve the eager transfer and allocate memory space in all processes of the import

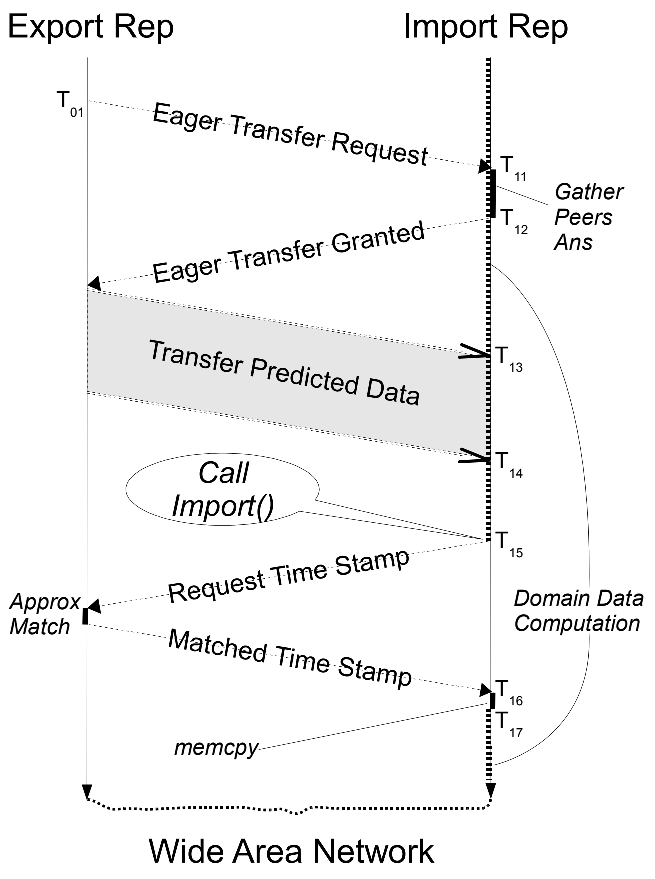


Figure 5.2: A Scenario for a Slow Data Import Component, Eager Transfer Approach

component.

2. If a data import request is received by the import rep *between* T_{11} and T_{12}, the eager transfer request might be denied or cancelled (not shown in the figure). However, if an import request is received *after* T_{12}, because the grant signal has been sent by the import component, the transfer for the granted eager request data will still be performed. Additional details about the collective safety properties and related protocols for this situation can be found in Chapter 6.

3. The predicted data object is transferred to the import component after the grant signal is received by the export component.

4. When the import application executes a later data import request, at time T_{15}, the request timestamp is sent to the export component, and the approximate match is performed to *identify* the exported data object with the *matched* timestamp.

5. If the matched data object has already been transferred, only the confirmation signal is sent back to the export component, and memory copies are performed locally from previously allocated memory to the user application in all import processes. However, if the matched data are still in the export component, the matched data object will then be transferred, as shown in Figure 5.1.

6. Our framework now uses a mono-periodic pattern predictor, which claims the future request timestamps are periodic with the period of p if the interval p between successive received request timestamps are kept the same for certain times, and our framework can support plug-in application-specific predictors.

If we compare Figures 5.1 and 5.2, the time between user application computations has been reduced from one round-trip delay plus the time for the data transfer to only the round-trip delay, when the eager transfer prediction is correct. The savings can be attributed to overlapping of computation with the background data transfers, a common approach in many parallel and distributed system architectures. More formally, when a data import component runs more slowly than an export component so that the eager transfer method can be used, the data import time T^{E}_{import} and the total execution time T^{E}_{total} can be expressed by the following, assuming all predictions are correct and the required data have been transferred to the import program when executing data import operations. Here the user application computation time, the time spent by the export program (including to perform the approximate match in each export process and to collect all answers in the export rep,) and the local memory copy time in each import component process are denoted by $T^E_{compute}(i)$, T_{export}, and $T^E_{memcpy}(i)$, respectively, for each of the N iterations.

$$
\begin{aligned}
T^E_{import} &= T_{export} + \delta_{ie} + \delta_{ei} + T^E_{memcpy} && (5.7)\\
T^E_{total} &= \sum_{i=1}^{N} (T^E_{compute}(i) + T^E_{import}(i)) && (5.8)\\
&= \sum_{i=1}^{N} T^E_{compute}(i) + \sum_{i=1}^{N} T^E_{import}(i) && (5.9)\\
&= \sum_{i=1}^{N} T^E_{compute}(i) + \sum_{i=1}^{N} T_{export} + \sum_{i=1}^{N} (\delta_{ie}(i) + \delta_{ei}(i)) + \sum_{i=1}^{N} T^E_{memcpy}(i) && (5.10)
\end{aligned}
$$

Although the user computation time $T^E_{compute}$ may be greater than the original $T_{compute}$ because of the extra work needed to perform eager data transfers, based on the experiments we will show in Section 5.3, the times are almost the same. Namely the following equation holds:

$$
T^E_{total} \approx \sum_{i=1}^{N} T_{compute}(i) + \sum_{i=1}^{N} (\delta_{ie}(i) + \delta_{ei}(i)) + \sum_{i=1}^{N} T_{export}(i) + \sum_{i=1}^{N} T^E_{memcpy}(i) \qquad (5.11)
$$

We observe that, when a data import component runs more slowly than the export component on a connection, and the prediction is correct for earlier eager transferred data, (1) the total execution time is *independent* of the network bandwidth, as shown in Equation 5.11, and (2) by comparing Equations 5.5 and 5.11, the architecture based on eager transfer reduces the total execution time by T^E_{save}, which is computed in Equation 5.12. The effectiveness of eager transfer relies on the available network bandwidth between components. T^E_{save}, the time saved, and the network bandwidth BW are inversely and linearly related, meaning that the lower the network bandwidth is, the more time is needed to transfer the matched data, so that more time is saved from using an eager transfer.

$$T_{save}^{E} = T_{total} - T_{total}^{E} = \sum_{i=1}^{N} \frac{M(i)}{BW(i)} - \sum_{i=1}^{N} T_{memcpy}^{E}(i) \tag{5.12}$$

5.2 Distributed Approximate Match

Although the eager transfer approach can effectively hide network bandwidth costs, network latency between components is still a concern, especially when those components run on different machines at different locations (such as in a Grid computing scenario).

To run the approximate match algorithm, both the current requested timestamp from the data *import* component, and the exported timestamps, from the data *export* component, are needed. The distinct nature of these input data sources makes the performance of the match architecture vulnerable to network latency costs, no matter where the algorithm runs. For example, running the approximate match algorithm in the data export component for eager transfers requires that each data import (Step 4 in Section 5.1.1) spend time $\delta_{ei} + \delta_{ie}$ to compute the *matched timestamp* even if the *matched data object* has already been transferred to the data import component. This is shown as the time interval T_{15} to T_{16} in Figure 5.2. On the other hand, if the match algorithm runs in the data export component, round-trip control messages can be avoided for each data *import*, but the cost incurred for a data *export* would be very high, since to collect the information necessary to perform the approximate match algorithm the exported timestamp needs to be sent to *each* import component that has a connection to that data object in the export component, whenever a new data object is exported.

Therefore, rather than running the *whole* approximate match in either component, we have designed a distributed approximate match algorithm. The algorithm has two parts: an *inverse approximate match* that runs in the export component and a *range check* that runs in the import component. By attaching the results of the inverse approximate match part of the algorithm to the data objects that are eagerly transferred, the achieved total execution time can be close to the lower bound shown in Equation 5.6, for the case when the import component runs more slowly than the export component and the correct predictions are made for the eager transfers. We now describe the two parts of the distributed approximate match algorithm in more detail.

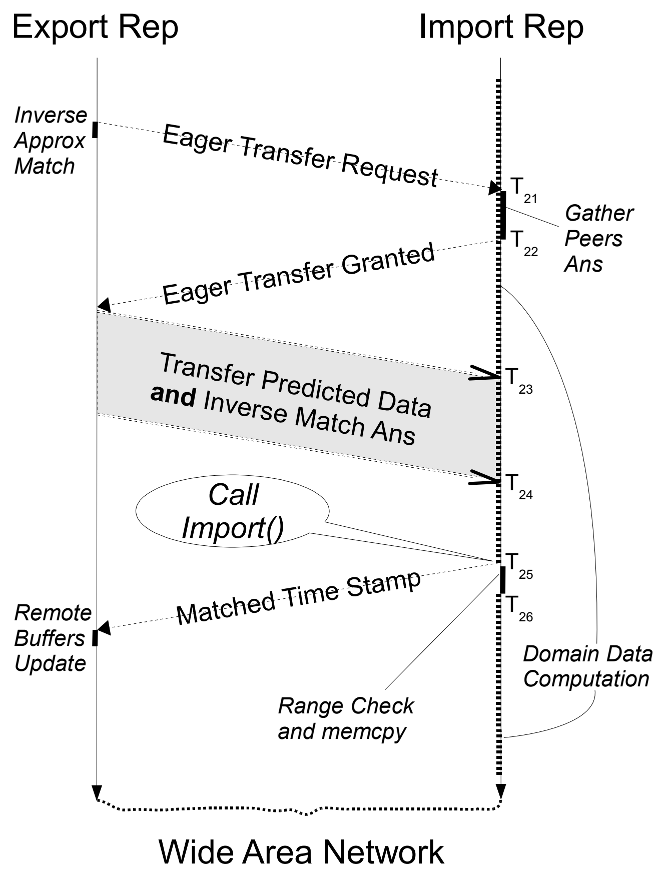


Figure 5.3: A Scenario for a Slow Data Import Component, Distributed Match Approach

5.2.1 Inverse Approximate Match

Given (1) a matching policy f, which is either a user-defined method or one of the pre-defined policies in Section 3.2, (2) a user-defined, connection specific tolerance value p, (3) an ordered set of the export timestamps on the connection, $S_T = \{t_{e_1}, t_{e_2}, ..., t_{e_k}\}$, and (4) a predicted requested timestamp t_p, which is the result from either a user-defined prediction function or a pre-defined function, for example based on simple differences between previously seen timestamps, we say that an export data object with timestamp t_{pm}, denoted by $\mathbf{D}@t_{pm}$, is a predicted data object if t_{pm} is the approximate match answer for a predicted timestamp t_p, or more formally,

$$f(S_T, p, t_p) = t_{pm} \text{ (or more tersely, } f(t_p) = t_{pm}) \tag{5.13}$$

Additionally the inverse approximate match G is defined to return a range $R_{t_{pm}}$ around t_{pm} such that $R_{t_{pm}}$ is a collection of elements whose approximate match answers are all the same, t_{pm}, or more formally

$$G(f, p, t_{pm}) = R_{t_{pm}} = \{t_x | f(t_x) = t_{pm}\} \tag{5.14}$$

We observe the following:

- $t_{pm} \in S_T$, because t_{pm} is also an export timestamp. Namely $\exists\, m$, s.t. $t_{pm} = t_{e_m}, 1 \leq m \leq k$.
- $t_p \in R_{t_{pm}}$, because $f(t_p) = t_{pm}$; that is to say, as long as the predicted timestamp t_p can be identified, $R_{t_{pm}}$ is not empty,
- *No* information from the *export* component is required to evaluate G, when the prediction algorithm runs in the export component.

While seeming complicated, the inverse approximate match G is straightforward to implement for the pre-defined match policies. The key is to find an appropriate range around t_{e_m} such that any element in that range returns the approximate match value t_{e_m}. For example, if the approximate match f is **REGU**, as defined in Section 3.2, the inverse approximate match can be evaluated in the following way:

$$G(REGU, p, t_{e_m}) = R_{t_{em}}^{REGL} = \begin{cases} [t_{e_m}, t_{e_m} + p] & \text{if } t_{e_m} + p < t_{e_{m+1}} \\ [t_{e_m}, t_{e_{m+1}}) & \text{otherwise} \end{cases} \tag{5.15}$$

When a data import component runs more slowly than the export component for a connection, a typical scenario for the inverse approximate match is shown in Figure 5.3. As soon as a new copy of the data object $\mathbf{D}@t_{e_j}$ is ready, the inverse approximate match is executed and, similarly to the eager transfer protocol from Chapter 5.1.1, two-way control signals are sent to get approval and allocate memory space from all processes in the import component for the data to be transferred. After receiving the grant signal from the import component, both the range $R_{t_{e_j}}$ and the predicted data object $\mathbf{D}@t_{e_j}$ are sent to and stored in the data import component. The range check algorithm, described in the next section, is performed whenever a data import request is performed, using the range data sent for the inverse approximate match.

5.2.2 Range Check

The range check part of the approximate match algorithm runs in the data import component, and complements the inverse approximate match algorithm. Given a requested timestamp t_r on a connection, a previously received $R_{t_{e_i}}$, and the corresponding cached data object $\mathbf{D}@t_{e_i}$, the range check H, identifies whether t_r is in the range $R_{t_{e_i}}$, or more formally,

$$H(t_r, R_{t_{e_i}}) = \begin{cases} 1 & \text{if } t_r \in R_{t_{e_i}} \\ 0 & \text{otherwise} \end{cases} \tag{5.16}$$

The range check H is simple to implement, runs quickly, and can be evaluated locally in the data import component. We also note the following observations.

Lemma 5.2.1 *Given a requested timestamp t_r, if $H(t_r, R_{t_{e_i}}) = 1$, then $\mathbf{D}@t_{e_i}$ is the approximate match result for t_r; otherwise $H(t_r, R_{t_{e_i}}) = 0$, and $\mathbf{D}@t_{e_i}$ is not the match result.*

Proof If $H(t_r, R_{t_{e_i}}) = 1$, then t_r is in $R_{t_{e_i}}$. By the definition of $R_{t_{e_i}}$ in Equation 5.14, we know $f(t_r) = t_{e_i}$. Namely the corresponding data object $\mathbf{D}@t_{e_i}$ is the approximate match result. A similar argument can be applied to the second part. $\blacksquare$

Lemma 5.2.2 *Given a requested timestamp t_r, if ranges $R_{T_1}, R_{T_2}, \ldots, R_{T_n}$, and their corresponding data objects $\mathbf{D}@T_1$,$\mathbf{D}@T_2$,…,$\mathbf{D}@T_n$, are stored in the import component, the range check H can evaluate to 1 for at most one range R_{T_i}.*

Proof By contradiction.

Assume $R_{T_i} \neq R_{T_j}$ and $H(t_r, R_{T_i}) = H(t_r, R_{T_j}) = 1$. By Lemma 5.2.1 both $\mathbf{D}@T_i$ and $\mathbf{D}@T_j$ are approximate match results for t_r, or $f(t_r) = T_i$ and $f(t_r) = T_j$. Because f cannot have two different values for the one input, $T_i = T_j$. Therefore $R_{T_i} = R_{T_j}$. This is a contradiction. $\blacksquare$

Lemma 5.2.1 implies that the approximate match results can be directly obtained from executing the range check H locally in the data import component, when the predictions for eager transfers are correct and the corresponding ranges and data objects are buffered in the data import component, or more formally,

$$f(S_T, p, t_p) = t_{pm} \iff H(t_p, G(f, p, t_{pm})) = 1. \tag{5.17}$$

Lemma 5.2.2 says there is no need to evaluate the remaining ranges if the range check returns 1 already for some range R_{T_k}, This implies that for each data import request it is possible to execute the range check H only once, if the predictions for eager transfers are correct and the received ranges and data objects are buffered in sorted order (by the ranges).

Equation 5.17 hints at an important point: as long as the timestamp for the predicted eager transfer is *close* to the requested timestamp by the data import component, *perfect* prediction is not necessary.

Lemma 5.2.3 *If the requested timestamp t_r is different from the predicted timestamp t_p, as long as both are close enough to have the same approximate match results, the predicted data object $\mathbf{D}@t_p$ will be the match for the request t_r.*

Proof Given $f(S_T, p, t_r) = f(S_T, p, t_p) = t_{pm}$, we know $H(t_r, G(f, p, t_{pm})) = 1$, by Equation 5.17. ▮

Figure 5.3 shows a typical scenario for the distributed match algorithm. When a data import component runs more slowly than the export component on a connection, some ranges and corresponding data objects will be buffered in the data import component. If the predictions for the eager transfers are good enough, the data import operations that occur between times T_{25} and T_{26} can be transformed into (1) range checks H to identify the *matched* data object, (2) a memory copy for the matched data object from the framework buffer to the user buffer, and (3) sending a control message to the export component for remote buffer management and other control mechanisms in the export component. So as soon as the first two operations are performed locally in each import process and the control messages are sent to the export component, the data import function can return to the application, and the execution time for a data import operation is no longer bounded by the latency of the network. More formally, if $T_{rc}(i)$, $T^D_{memcpy}(i)$, $T^D_{import}(i)$, and $T^D_{compute}(i)$ denote the time to execute the range check, the time to perform the local memory copy, the data import time, and the user application computation time, respectively, for each of the i out of the total N iterations, the total execution time T^D_{total} for a distributed approximate match is:

$$\begin{aligned}
T^D_{import} &= T_{rc} + T^D_{memcpy} && (5.18)\\
&\approx T^D_{memcpy} && (5.19)\\
T^D_{total} &= \sum_{i=1}^{N} (T^D_{compute}(i) + T^D_{import}(i)) && (5.20)\\
&= \sum_{i=1}^{N} T^D_{compute}(i) + \sum_{i=1}^{N} T^D_{import}(i) && (5.21)\\
&= \sum_{i=1}^{N} T^D_{compute}(i) + \sum_{i=1}^{N} T_{rc}(i) + \sum_{i=1}^{N} T^D_{memcpy}(i) && (5.22)\\
&\approx \sum_{i=1}^{N} T^D_{compute}(i) + \sum_{i=1}^{N} T^D_{memcpy}(i) && (5.23)
\end{aligned}$$

Equations 5.23 and 5.19 hold because the range check H, which is independent of (1) the user application complexity and (2) the size of the matched data objects, takes much less time than the other term(s).

As for the eager transfers described in Section 5.1.1, although the user computation time $T^D_{compute}$ may be greater than the original $T_{compute}$, because of the the eager data transfers and inverse approximate matches begin performed in the background, the experimental results in Section 5.3 show that the additional costs are negligible. Namely, the following equation holds

$$T^D_{total} \approx \sum_{i=1}^{N} T_{compute}(i) + \sum_{i=1}^{N} T^D_{memcpy}(i) \tag{5.24}$$

By comparing Equations 5.24 and 5.6 from Section 5.1, we see that the execution time of the distributed algorithm is only higher than the lower bound by $\sum_{i=1}^{N} T^D_{memcpy}(i)$, which is usually very small compared to the user application computation time, which is $\sum_{i=1}^{N} T_{compute}(i)$. The distributed approach reduces the total execution time by T^D_{save}, as shown in Equation 5.25. T^D_{save} shows that the network costs between coupled components, both for bandwidth and latency, have been eliminated in the distributed approach, with the additional expense of local memory copies.

$$T^D_{save} = T_{total} - T^D_{total} = \sum_{i=1}^{N} \frac{M(i)}{BW(i)} + \sum_{i=1}^{N} T_{export}(i) + \sum_{i=1}^{N} (\delta_{ie}(i) + \delta_{ei}(i)) - \sum_{i=1}^{N} T^D_{memcpy}(i) \tag{5.25}$$

5.3 Experiment 1

To measure the effectiveness of the proposed architecture, we set up a micro-benchmark as follows.

- Solve $u_{tt} = u_{xx} + u_{yy} + f(t,x,y)$, the two dimensional diffusion equation, with a forcing function $f(t,x,y)$, which can be viewed as the external input for $u(t,x,y)$.
- Program U_e computes $u(t,x,y)$ and has four processes, each of which is responsible for a 512×512 array. Program F_e computes $f(t,x,y)$ and owns a 1024×1024 array that is evenly distributed among the participating processes.
- Four configurations are considered. Program F_e has either 4, 8, 16, or 32 processes. Three framework architectures, the original one [111] (called on-demand (**OD**)), the one using eager transfers only (called **ET**), and the one using both eager transfers and distributed approximate match (called **ET+DM**) are measured.
- Data of size 1024×1024 is transferred from F_e to U_e with matching policy REGU and precision 0.2. Namely program F_e is the data export component program and U_e is the corresponding data import component.
- Both Program U_e and Program F_e run on the same cluster of 2.8GHz Pentium 4

machines connected via Gigabit Ethernet, and the underlying operating system is Linux 2.4.21-37.

- To measure the performance of the three architectures with different network latencies between components, delays varying from 0 to 200ms are artificially introduced, in each direction, whenever control messages or data transfers are done between the two programs. The longer network delays have been chosen based on measurements of the round trip time from the University of Maryland to a university in Taiwan, to model a Grid computing scenario, and the shorter delays model the delay between clusters inside the University of Maryland.

- For each architecture, each configuration was run three times, and 110 data transfers were done in each run. In addition, the experiment was designed so that the requested timestamp in the import component and the predicted timestamp produces the same approximate match result, so the predictions are correct.

- The import program U_e, which is the slower one in this experiment, has a structure similar to the Import Program P1 in Figure 3.1, and `gettimeofday` was used to measure the times for the finite difference computations and the data import operations.

Delay	0	0.2	2	20	200
OD	876.5 (1.4)	876.5 (1.2)	876.9 (2.0)	876.6 (1.4)	876.4 (1.2)
ET	881.6 (3.8)	881.8 (3.6)	881.6 (3.5)	882.0 (3.8)	877.5 (3.2)
ET+DM	881.4 (3.1)	881.5 (2.8)	882.1 (3.3)	881.9 (3.6)	881.4 (3.8)

Table 5.1: Average Time for App Computation (with StDev), for 32 Data Import Processes, in ms

Table 5.1 shows the computation time for all three architectures (OD, ET, ET+DM), when the data export program F_e uses 32 processes and the delays vary from 0 to 200ms. When F_e uses 4, 8, or 16 processes, the results are similar. Clearly, in all three architectures, the network delay between components does not affect the computation time much. In addition, the computation time for the ET and ET+DM architectures is slightly higher, in both average and standard deviation, than the computation time for the OD architecture. The reason is that, during the application computations, even if ET and ET+DM need to transfer predicted data in the background (which OD does not), the incurred overhead is very small (at most 0.6%) in our machines, which have standard configurations, due to low overheads in Linux thread scheduling between the application thread and the framework threads. Equations 5.11 and 5.24 are derived based on these results.

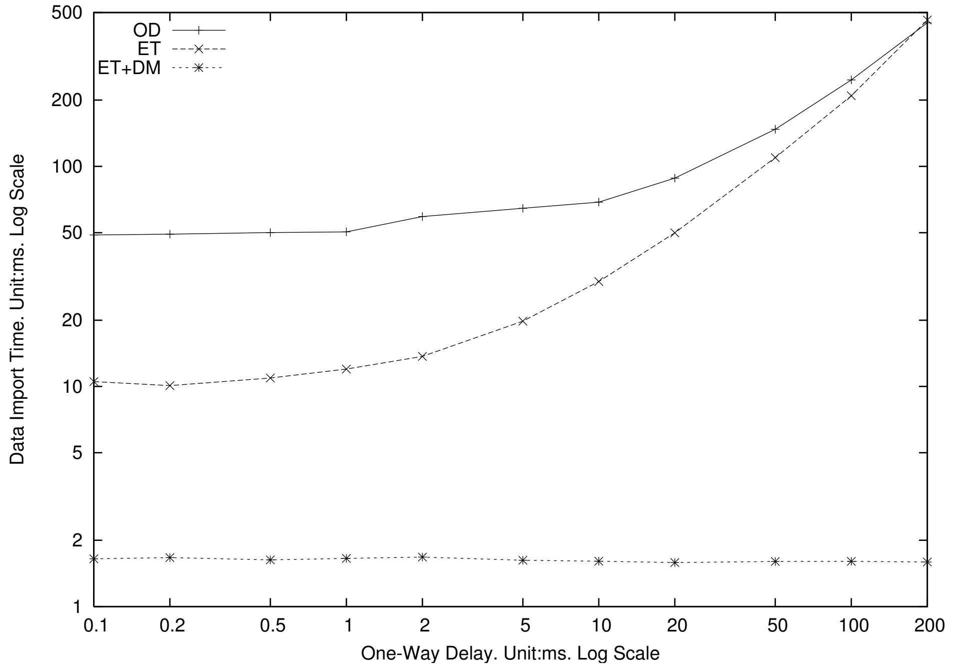


Figure 5.4: Import Time for OD, ET, and ET+DM

Figure 5.4 shows the execution time for data import operations in all three architectures when F uses 32 processes (note that both axes are log scale), and the round trip delays and data import time for the OD and ET architectures are shown in Figure 5.5 (where both axes are linear scale).

First, as shown in Figure 5.4, the ET+DM architecture needs the least time to import user data, and the required time (around 1.6ms) is independent of the network delay between components. This result confirms our earlier analysis from Equation 5.23. That is, when predictions are good enough and the predicted data objects have been transferred to the import program, the data import operation in each import process is essentially a local memory copy and its execution time T^D_{memcpy} is independent of the latencies between components.

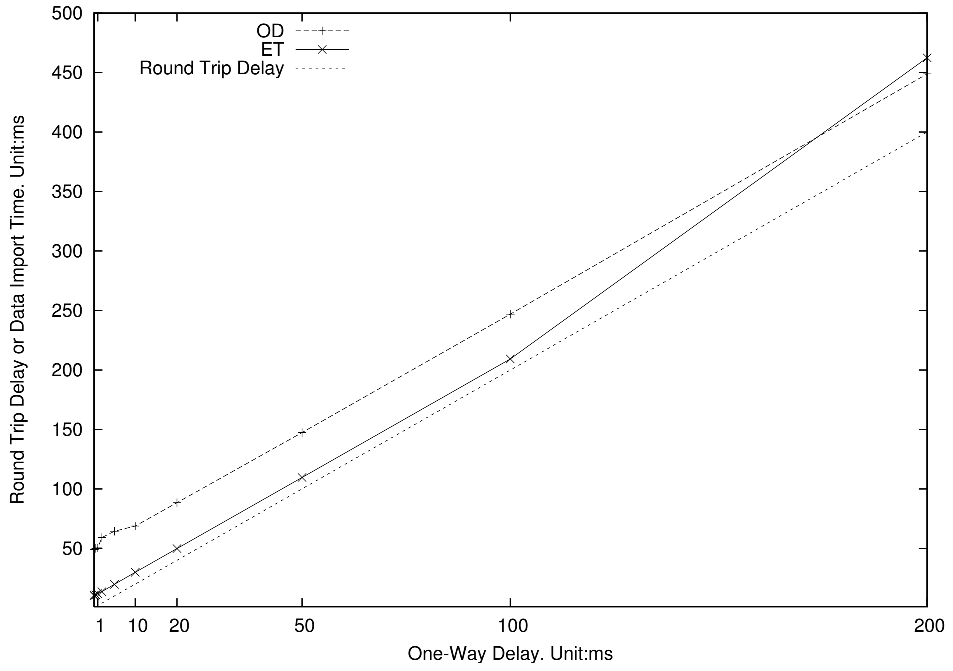


Figure 5.5: Round Trip Delay and Import Time for OD & ET

Second, as shown in Figure 5.5, the OD architecture needs the most time to import user data, and the data import time is always around 50ms more than the round trip network delay. Comparing the result with Equation 5.1, the 50ms is the sum of T_{export}, the data export time, and $\frac{M}{BW}$, the time to transfer size M data objects over a 0-latency, BW bandwidth network.

Third, Figure 5.4 shows that the ET architecture only outperforms the OD architecture in measuring data import time when the latencies are small. In fact, as seen clearly in Figure 5.5, when the latency is 200ms the ET architecture requires more time to import data than the OD architecture does. Figure 5.5 also shows that when the network latency is smaller than 100ms, the line for the data import time in ET architecture is parallel to the line for the round trip delays (RTD). The difference between the two lines is around 10ms, which can be observed when the one-way delay is 0.1ms in Figure 5.4.

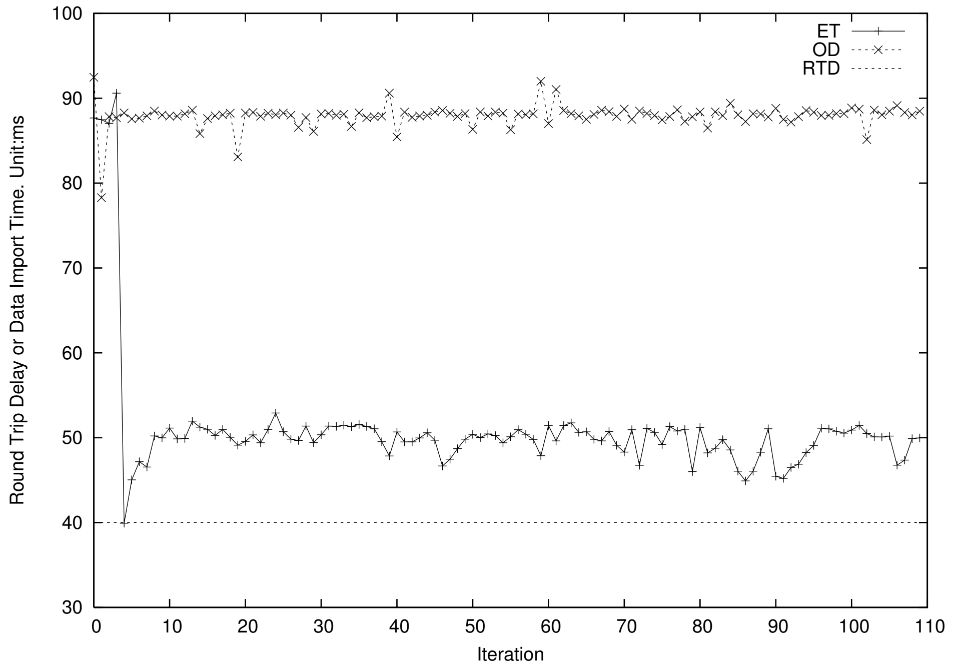


Figure 5.6: One-Way Delay = 20ms

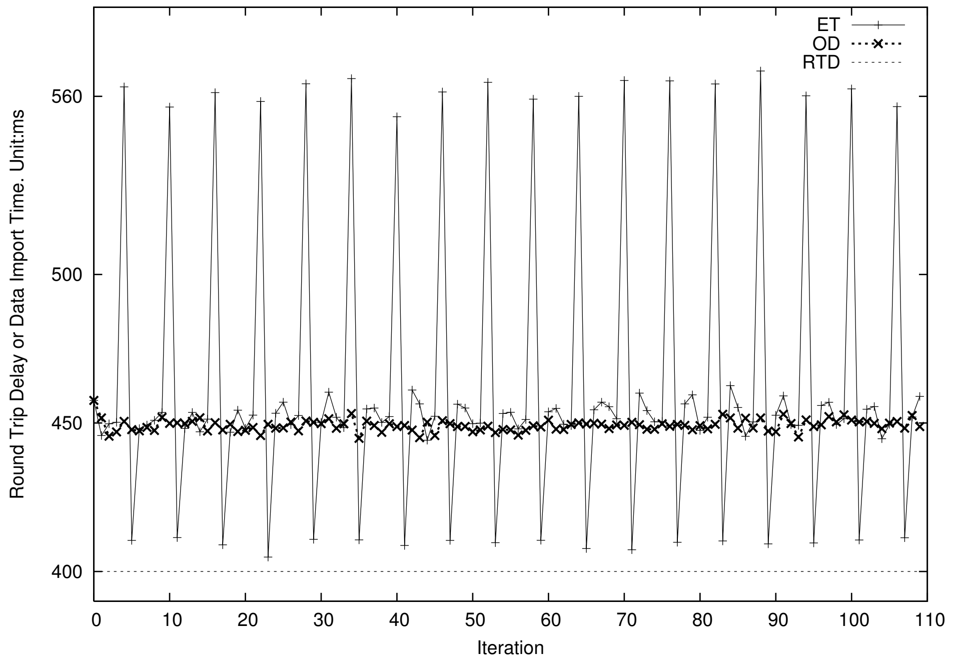


Figure 5.7: One-Way Delay = 200ms

To explain the behavior when the delay is 200ms, experiment traces and related RTD are shown in Figures 5.6 and 5.7 when the one-way latency is 20ms and 200ms, respectively. Figure 5.6 shows that, after the prediction pattern is learned, the ET architecture needs around 50ms to perform data import operations, which is around 10ms more than the RTD. The experimental results for other configurations (F uses 4, 8, or 16 processes) and other delays (except 200ms) are similar.

If Equation 5.7 holds, the 10ms difference would be the sum of T^{E}_{memcpy}, the local memory copy time for the ET architecture, and T^{E}_{export}, the time spent on the export program for the approximate match. Assuming T^{E}_{memcpy} is similar to T^{D}_{memcpy} (1.6ms), the local memory copy time T^{D}_{memcpy} for the ET+DM architecture, T^{E}_{export} would be around 8.4ms.

Figure 5.7 exhibits different behavior for a very high network latency, showing the cost of maintaining collective safety (Step 2 discussed in Section 5.1.1) – the eager transfer request might be denied or cancelled if a new data import request shows up *before* the operations for granting earlier eager requests complete. Depending on the timing of events, the cancellation might happen in one of the import processes, the import rep, one of the export processes, or the export rep. Details for all possible cases can be found in Chapter 6. However, if the new import request is received *after* the grant signal (for an earlier received eager request) has been sent, the transfer for the granted eager request data will be performed, and if the new request has a matched data object that is different from the one for the eager request, the transfer for the matched object will not start until the data transfer for the eager request finishes.

Figure 5.7 shows that the OD architecture takes about 450ms to perform data import operations, and most of the data import operations for the ET architecture also take about 450ms. This is because many of the eager transfer requests in this scenario (network latency 200ms, F uses 32 processes) are cancelled, so the data import operations for the ET architecture act just like the data import operations for the OD architecture.

A more interesting situation is that sometimes the ET architecture takes around 560ms to perform an import operation $Import_i$, which is around 110ms more than the 450ms needed for import operations in the OD architecture, but then the next import operation $Import_{i+1}$ takes only around 410ms, which is around 10ms more than the RTD time of 400ms. The reason is that unlike most import operations in the ET architecture, the operation $Import_i$ is received by the import rep *after* the grant signal (for an earlier received eager request) has been sent. Therefore the operation $Import_i$ takes extra time, to wait for the eager request data to be transferred. Because extra time is spent for the operation $Import_i$, the data for $Import_{i+1}$ has a chance to be buffered in the import process before $Import_{i+1}$ is issued. Therefore only the RTD time, the time for approximate match, and the local memory copy time T^{D}_{memcpy} are needed to perform $Import_{i+1}$. After that, the buffer for the eager transfer data is empty again, and the next import operation $Import_{i+2}$ will be executed as for the OD architecture.

The above scenario is caused mainly by eager transfer cancellation. The more time the data import operation needs, the more likely it is that an earlier eager request will be cancelled. That is why this behavior does not occur for lower network latencies, as seen in Figure 5.6. This behavior indirectly shows the true benefits of the ET+DM architecture. The data import time is so small that eager requests are rarely cancelled, for the scenarios shown in the experiments.

5.4 Experiment 2

We use the following experiment, by introducing extra workload for domain data computations, to see the execution time under two different scenarios.

- 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.

- Ap0, running with four processes, is the exporting program for the forcing function (t,x,y). It generates data at every basic time unit (the time units can be arbitrary). Ap1, running with sixteen processes, is the importing program for $u(t,x,y)$. The matching criterion specified in the configuration file is $\langle$REGL,0.05$\rangle$. Both eager transfers and distributed approximate match (called **ET+DM**) are used.

- Both programs run on the same cluster of 2.8GHz Pentium 4 machines connected via Gigabit Ethernet, and the underlying operating system is Linux 2.4.21-37.

Extra workload	*u* : 1024 *x* 1024 delay = 1ms	*u*: 128 *x* 128 delay = 200ms
0	86ms	428ms
1s	1.08s	1.05s
3s	3.08s	3.05s

Table 5.2: Average execution time

- Two scenarios are considered. The first one is similar to the data exchange in local area networks: the size of transferred data (*u*) is large (1024 *x* 1024) but the end-to-end delay between programs is small (1ms). The second one is similar to the data exchange in wide area networks: the size of transferred data (*u*) is small (128 *x* 128) but the end-to-end delay between programs is large (200ms). In each scenario, three different workloads (origin workload, 1s extra, and 3s extra) are considered.

- Each configuration was run three times, 200 copies of data objects are exported, half of them are imported, and the average execution time per iteration from the slowest process in program Ap1 is shown in Table 5.2.

We make the following observations. First, when *u* is a 1024 *x* 1024 array and the end-to-end delay is 1ms, the execution time is dominated by the workload, which is composed of the domain computation and the extra workload. By comparing all three different workloads, we know the time for domain computation is 86ms per iteration, and the execution time per iteration would be *Y*s + 86ms if *Y*s extra workload is introduced.

Second, when *u* is a 128 *x* 128 array and end-to-end delay is 200ms, the experimental result shows the effectiveness of **ET+DM**. When the extra workload is not introduced, the total execution time per iteration is 428ms, which is composed of two end-to-end delays (400ms) and domain data computation (28ms). However, when 1s (or 3s) is introduced as the extra workload, the combined workload (1.048s or 3.048s) is larger than the round trip delay (400ms). In this situation, **ET+DM** starts to work such that the round trip delay disappears completely and the execution time per iteration is nearly the same as the time for the combined workload, rather than the sum of the combined workload time and the round trip delay.

Chapter 6

Collective Control Protocol

Earlier we suggested a runtime-based framework to control data transfer between loosely coupled application components. One of important features of this framework is that, whenever a data object is exported or imported, its associated (simulation) timestamp is also required to passed to the framework. Those timestamps are keys for the underlying priority-based control protocol.

Whenever user applications issue (on-demand) timestamped data requests, rather than directly performing collective communications, as in a traditional tightly coupled framework [101], control messages are first exchanged between data exporters and importers. If the desired data can neither be found nor be generated, the request will be denied. However, if requests can be satisfied (by the approximate match algorithm), matched export timestamps will be used as tags to perform associated collective communication operations.

Eager transfer (with distributed approximate match), as shown earlier in Chapter 5, improves the runtime performance by a large amount. When the effect of network bandwidth and latency can not be ignored. That is a common situation when the components run on different machines at different locations (as in Grid computing), for various reasons, including concerns about user privileges, security or that the runtime requirements for a component are for special hardware or software packages.

To support both on-demand and eager transfer operations, the underlying architecture is designed such that control messages can be initiated by either parties – control message for on-demand data requests are initiated by data import components and for eager transfer are by data export components.

This two-sided initialization approach make the framework flexible and efficient, at the price of a complicated control protocol, especially for the coverage of all possible interactions between the two modes. To handle this difficulty, we will describe a protocol construction method, which is based on validating all possible compositions of smaller protocols for each of the modes.

6.1 System Outline

Our system has several interesting characteristics. First, we target parallel user programs that use the single program multiple data (SPMD) model. Application data objects (mainly arrays) may be distributed across multiple processes, and a data decomposition (e.g., by blocks in each array dimension) is applied to map distributed data elements to the various processes. Communication between processes is achieved through one of several methods:

1. Ad-hoc approaches, such as traditional message passing (i.e. send/receive pairs using the Message Passing Interface (MPI) [98]) between processes, perhaps using Cartesian topology operations to organize the processes for efficient communication (e.g., MPI_Cart_xxx).
2. Programming language extensions. Partitioned Global Array Space (PGAS) language [23, 112, 59] is a programming languages approach to support parallel en-

vironments by extending existing languages,such as Co-Array Fortran [82, 83], Unified Parallel C [16, 25, 54, 44], and Titanium [11, 52, 58, 112] (a Java dialect), or designing new ones, such as Chapel [31, 18, 32], and X10 [20, 21, 95, 3, 94, 2].

3. Language-independent libraries, including InterComm [101], Parallel Application Work Space (PAWS) [37], and the Model Coupling Toolkit (MCT) [64], for data exchange between multiple parallel programs.

Second, systems may be able to buffer multiple copies of distributed data objects **D**, with each copy associated with an unique tag, or timestamp, t and denoted by $\mathbf{D}@t$. For example, in Figure 6.1 4 copies of a distributed data objects each span 12 processes. All tags for the same distributed data object **D** form an increasing tag sequence, with $t_i < t_j$ if the copy $\mathbf{D}@t_i$ is generated (or requested) before $\mathbf{D}@t_j$. Although seemingly arbitrary, such tags can be naturally found in target user applications. For example, they can be the simulation timestamps in scientific simulations that solve time dependent partial differential equations [111], or the timestamps in HTTP 1.1 [39] header fields.

Third, not all distributed data objects that are generated by one program are required to be transferred to another program, but if such transfers do happen, they must be *collectively consistent*. Namely if distributed data object **D** spans n processes in program P_1, and one of the n processes p_k transfers its own part of $\mathbf{D}@t_i$, all other processes of the same program must transfer their part of $\mathbf{D}@t_i$, although those transfers do not need to happen at exactly the same time. Formally, the distributed data transfer must be *collective*, but not necessarily *synchronous*. Moreover, if during program execution process p_k transfers $\mathbf{D}@t_{i_1}$, $\mathbf{D}@t_{i_2}$, ..., $\mathbf{D}@t_{i_m}$, all other processes in the same program **P1** must also transfer their own parts of $\mathbf{D}@t_{i_1}, \mathbf{D}@t_{i_2}, \ldots, \mathbf{D}@t_{i_m}$, and in the same order. That is, the *tag transfer history*, which is t_{i_1}, t_{i_2}, ..., t_{i_m} for our example, must be the same for all processes in the same program.

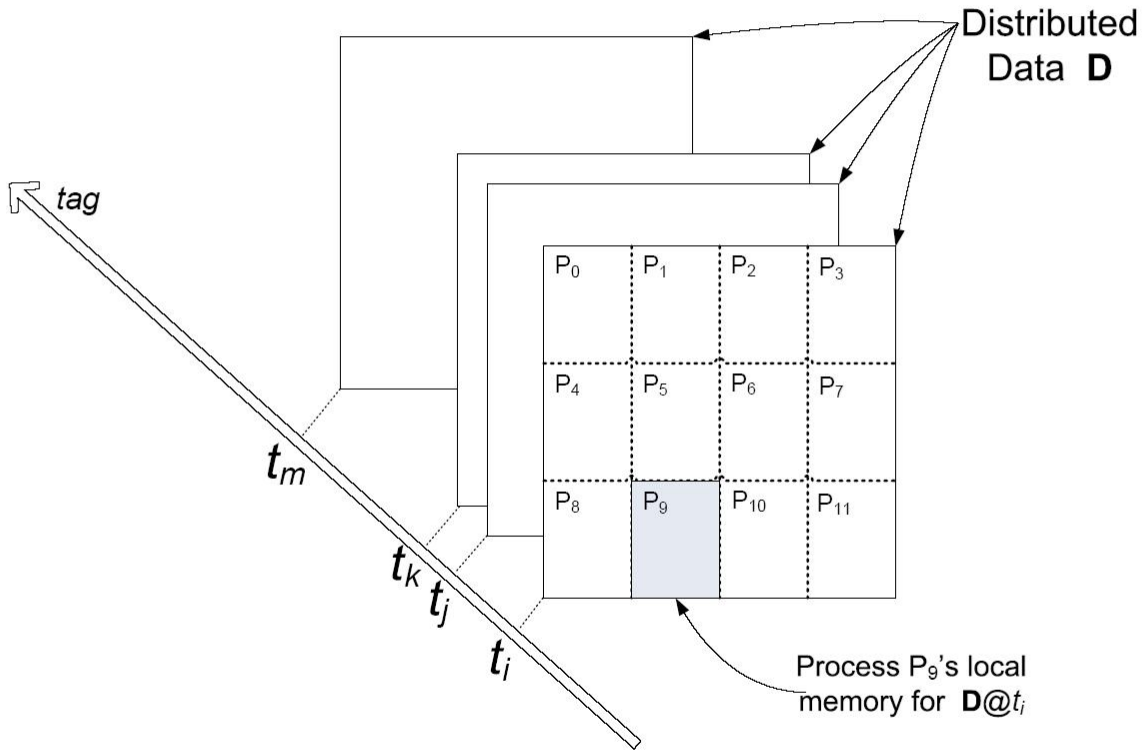


Figure 6.1: Copies of Distributed Data

Fourth, data transfers can be initiated by either data export programs or data import programs for any given transfer. Normally, the control message for a data transfer is initiated by a data import request from an application program. In this case the data objects are *pulled* by the data import program, and we call that an *on-demand* data transfer. Additionally when the system observes a pattern in the requests made by a data import program, the system can then decide to have the data export program *push* data objects to the data import program, if the system predicts that they will be (eventually) be requested, and we call the push approach an *eager* data transfer. If there are simultaneous on-demand and eager data transfer requests, the on-demand one has a higher priority since the data import program is waiting for the requested data.

Finally, a special execution entity, called the *representative*, or *rep* for short, is used as a control message gateway for each program. The *rep* combines, forwards, and caches control messages between all processes in a program, and also communicates with all relevant *rep*s in other programs. The representative is the central control point for collective messages and has the following two advantages over a more decentralized mechanism: (1) collective consistency for all data transfers can be maintained in a relatively straightforward way, and (2) priority-based transfers, on-demand over eager, can be supported correctly.

6.2 Protocol Operations

In a tightly coupled system distributed data objects are directly transferred whenever a new copy is generated or requested. In this approach, the data import and export operations must be carefully matched by the system integrator so that the expected data can always be generated and requested as required. This approach does not work very well for a large-scale coupled environment in which each component might be separately developed, and the participating components might be changed.

Rather than transferring data objects immediately after a new copy is generated or requested, in our system design the participating process issues a control message instead. The control message is used by the system to decide whether a data transfer should happen or not, and this approach effectively handles data availability issues in tightly coupled systems. The following explains the basic operations of our control protocol. Its construction method and correctness will be mentioned in Section 6.3.

6.2.1 Control Flow

In our system design each program is composed of multiple processes, and the processes run on either a shared memory machine or one or more nodes of a compute cluster. Each process contains one application execution unit and several system-level units. Currently each execution unit is implemented as a POSIX thread (Pthread) [80, 15, 36]. In addition, one extra thread running in one process, called the *representative*, acts as the control message gateway for all processes in a program.

The system starts in on-demand transfer mode, in which a control message is always sent to the representative (*rep*) by the data import process whenever the user thread executes a data import operation, each of which consists of a tag and the memory address for the data to be imported. The tag must be unique for each data import operation inside each process, but because of the requirement for collective consistency, the tag must be the same for *all* processes in the same data import program. More specifically, all processes in a data import program must have exactly the same tag transfer history,

When receiving tagged import requests, the data import representative executes one of the following two operations:

1. If the received tag is new, a control message requesting a data transfer is sent to the data export *rep*. When the data export *rep* receives the request, it forwards the request to all processes in the data export program. Then an approximate match operation [111] is performed by each data export process, and the result is sent back to the requesting data import process via the data import representative, as shown in Figure 6.2. In addition, the result of the match operation is cached in the data import *rep*.

2. If the received tag has been seen before, the *rep* sends back the cached result.

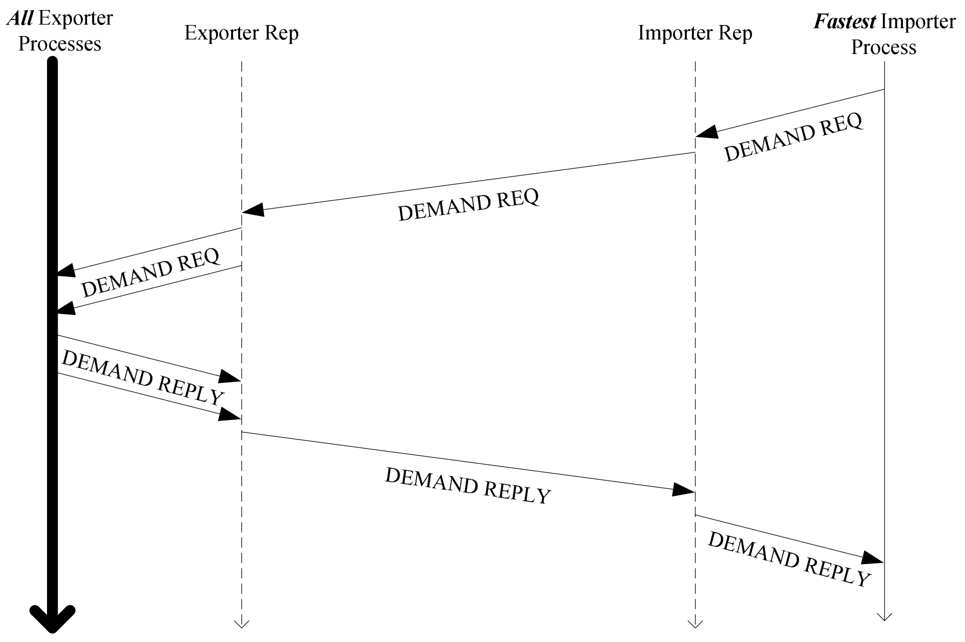


Figure 6.2: On-Demand Operation

The cached result is no longer needed once the tag has been requested by all data import processes.

Eager transfer mode is initiated when a pattern of tag requests is identified by the data export program. In this case an eager request control message will be sent from the data export process to the data export *rep*, whenever a data object with a tag that is predicted to be needed by the data import is generated, but the data export *rep* will not forward the request to the data import *rep* until it receives request messages with that tag from *all* data export processes. The data import *rep* then forwards the eager request message to all data import processes, and each data import process will grant the request if adequate memory space is available in that process. If any data import process has insufficient space available, the eager request is denied. The data import *rep* sends the combined reply back to all data export processes via the data export *rep*. The overall

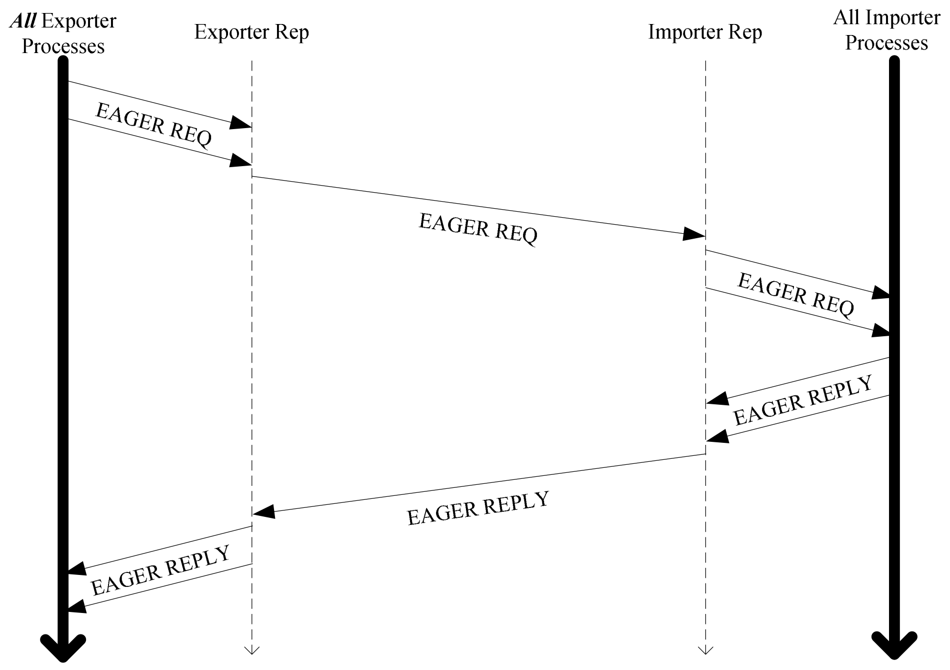


Figure 6.3: Eager Transfer Operation

control flow for eager transfers is shown in Figure 6.3.

Initiation of control messages differs in the two transfer modes (the data import program initiates in on-demand mode and the data export program initiates in eager transfer mode), and when those control messages are interleaved in time the ones for on-demand transfers have higher priority, because those requests require the data import program to block. For example, as shown in Figure 6.4, if the data export *rep* receives an on-demand request from the data import *rep* after an eager request has been made but before its outcome has been determined, the eager request is terminated and each data export process will deal with the on-demand request. Similarly, an eager request received by the data import *rep* will be ignored if an on-demand transfer request is ongoing.

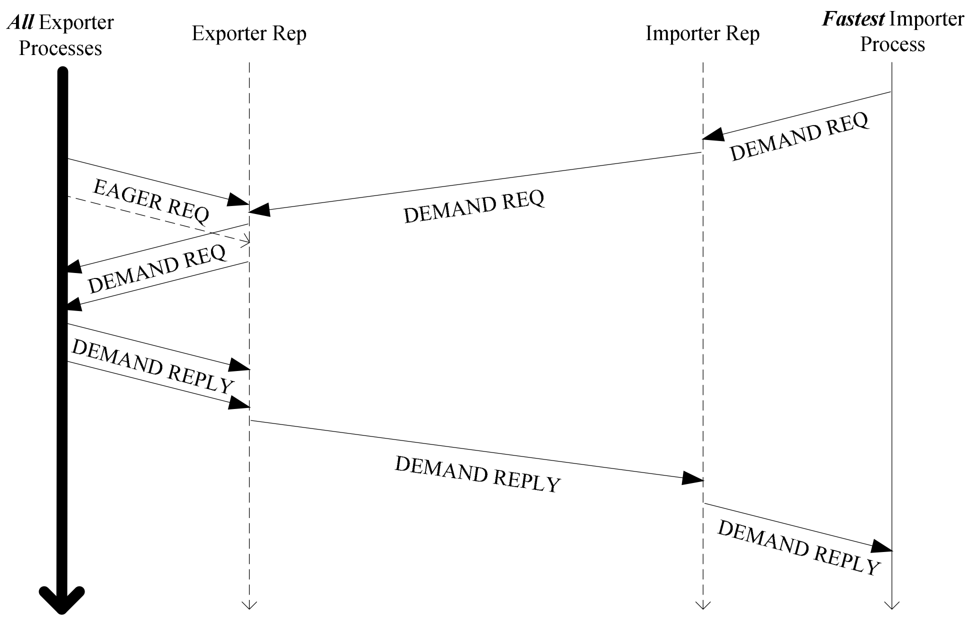


Figure 6.4: Priority-based Operation

6.3 Protocol Construction : Composition Approach

One of the challenges in designing the multi-threaded control protocol is *completeness* – determining the correctness of all possible states. The well-known finite state machine (FSM) or extended finite state machine (EFSM) approaches can suffer from state space explosion, and some high-level approaches such as SDL [56] and its extensions do not have direct mappings from modeling to implementation. We, however, can take advantage of two properties of our system. First, both the on-demand and eager transfer modes are not too complex in isolation, so it is easy to construct a correct FSM for each mode. Second, it is the interaction between the two modes that makes the protocol complicated, both in the state spaces and in the state transitions. To have a closer connection between modeling and implementation, as well as to take advantage of the above properties we have constructed the overall protocol by composing small FSMs,

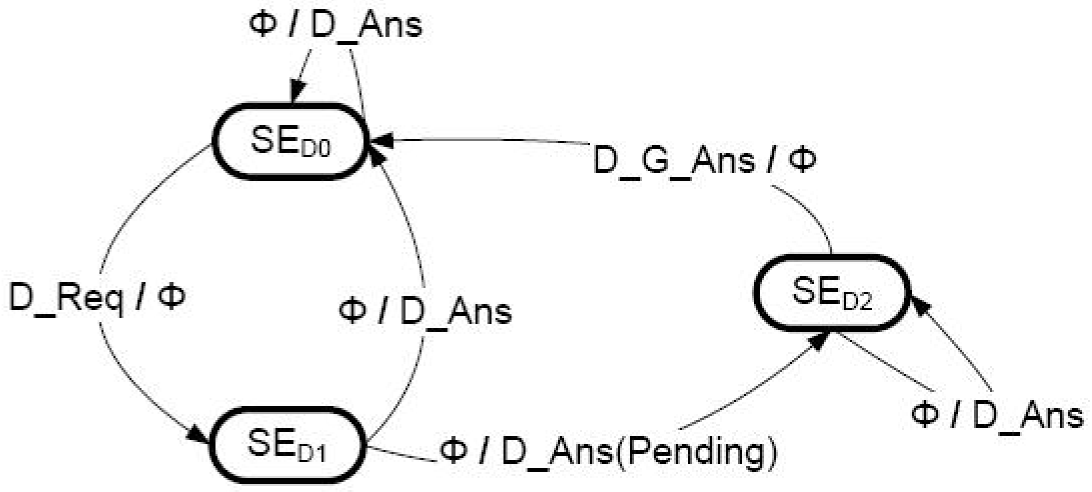
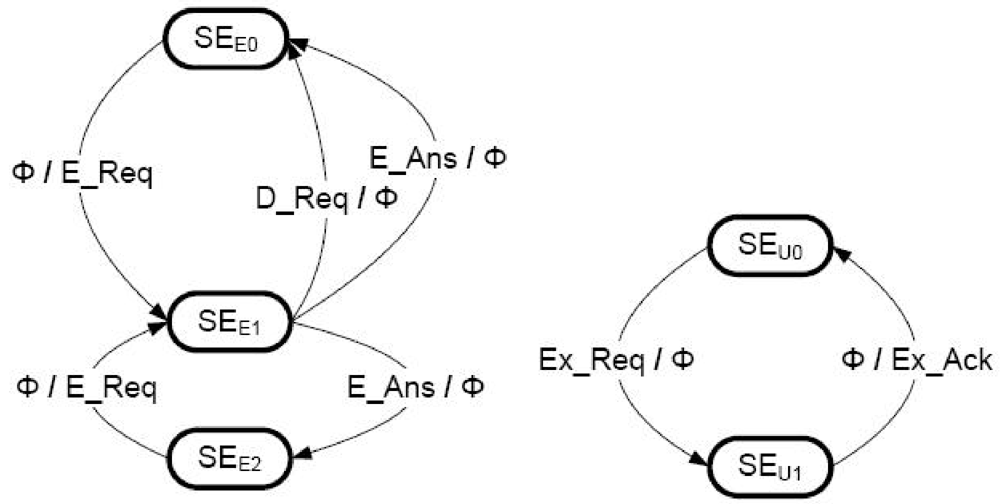


Figure 6.5: On-Demand Operation

each of which is clearly a correct model for the underlying operations. More specifically, the simple FSMs are constructed first, then *all* combinations of states and events between those FSMs are considered, and only the ones that are valid are included in the combined FSM.

For example, Figure 6.5 shows the FSM (a Mealy machine) in each producer process for an on-demand transfer, in which there are only three states (labeled as SE_{D0}, SE_{D1}, and SE_{D2} with SE_{Di} meaning *S*tate for *E*xporting data, *D*emand mode operation # *i*, *i* = 0..2), and six transitions (labeled as $Input\ Signal/Output\ Signal$ pairs). Here the signal X_Y means mode X (*D*emand mode, *E*ager transfer mode, or user *Ex*port) for operation Y (*Req*uest, *Ans*wer or *Ack*noledge). As shown, this FSM is quite simple and its correctness can be validated easily. Additionally the mapping from this FSM to the related multi-threaded implementation is straightforward. Figures 6.6(a) and 6.6(b) shows the FSMs for the eager transfer operation (SE_{Ei} meaning *S*tate for *E*xporting data, *E*ager transfer mode operation # *i*, *i* = 0..2) and the user application (SE_{Ui} meaning *S*tate for *E*xporting data, *U*ser application mode operation # *i*, *i* = 0..1), respectively.



(a) Eager Operation (b) Foreground Application Thread

Figure 6.6: Two Small FSMs

The states in the combined FSM can be constructed by validating *all* of the composition of states in the small FSMs, as shown in Table 6.1. In our system, 10 of the 18 composite states are invalid in the context of the system operations, 5 states are transient states, and the remaining 3 persistent states have special meanings. State SE_{D0} SE_{E0} SE_{U0} denotes computation in the application – it is an idle state from the viewpoint of the control messages. State SE_{D0} SE_{E1} SE_{U0} is waiting for a reply from an eager transfer request, and state SE_{D2} SE_{E1} SE_{U0} is for a self-tuning optimization operation in which slower processes can avoid some unnecessary memory copies with the help of the fastest process in the same program as mentioned in Chapter 5.

Once the states have been identified, the overall FSM can be constructed by adding the transitions between states, which are the validated compositions of the transitions in each small FSM, as shown in Figure 6.7. The overall FSM for data import processes can be constructed in the similar way, as shown in Figure 6.8.

One of the advantages of the composition approach is that the relationship between different modes can be covered systematically. For example, the transition *D_Req*,Φ,Φ / Φ,Φ,Φ, from state SE_{D0} SE_E1 SE_{U0} to state SE_{D1} SE_{E0} SE_{U0}, shows that the on-demand operation has higher priority, as mentioned in Section 6.1 – an eager transfer request will be canceled whenever an on-demand request arrives.

The composition approach can also help to design an FSM involving multiple different tasks, even if the tasks are implemented with only one thread. For example Figures 6.9 and 6.10 show the eager transfer FSM and the on-demand FSM for the *rep* process for a data import program, respectively. The overall FSM for data import rep, constructed using the method described previously, is shown in Figure 6.11. Likewise individual FSMs and the overall FSM for data export rep are shown in Figure 6.12.

6.3.1 Correctness

An important concern for the proposed protocol is its collective correctness. Formally, the tag transfer histories of all processes in the same program must be the same.

Although the protocol is run by multiple threads and in multiple processes, the *rep* in each program acts as the control message gateway; data transfers in each process cannot start until confirmation messages containing the confirmed tag from the *rep* are received. Each process maintains two ordered tag sets: T_c for confirmed tags and T_{uc} for pending tags, and operates as follows:

1. Tags are supplied by user programs whenever data object requests are made or data objects are generated, as shown in Figure 3.1, and each tag must be unique across all such operations.

2. A tag t_u is added to T_{uc} whenever a user program requests a data object, or when an eager transfer request arrives because the application generated a new data object.

The tags in T_{uc} do not trigger data transfers. Additionally a control message with tag t_u will be sent to the *rep*.

3. Based on the tags it receive, the *rep* selects a unique tag, and a confirmation message with the selected tag will be sent to the requesting processes. If the *rep* cannot determine the final tag due to misbehavior by the user program (such as inconsistent arguments provided by different processes in the same program,) runtime tag errors will occur that may not be able to be detected or avoided by the framework.

4. If one process receives a confirmation message with tag t_r from its *rep*, all other processes in the same program will receive the same confirmation message with tag t_r, but not necessarily at the same time.

5. Whenever a confirmation message is received from the *rep*, each process puts the received tag t_r into T_c, and removes t_r from T_{uc} if it is there. A distributed data transfer will then be started with tag t_r.

We claim that all processes will have the same T_c, which is the tag transfer history previously noted, if a runtime tag error did not happen during execution. If $T_c(p_i)$ and $T_c(p_j)$ differ after program execution ($T_c(p)$ denotes the tag transfer history of process p), at least one tag t_k would exist in either $T_c(p_i)$ or $T_c(p_j)$, but not in both. That implies that during execution the tag t_k is received only by p_i or p_j, but not both. This is a contradiction.

Overall States	Validity	Note
SE_{D0} SE_{E0} SE_{U0}	valid	Application Computation
SE_{D0} SE_{E0} SE_{U1}	transit	Distributed Data Generated
SE_{D0} SE_{E1} SE_{U0}	valid	Wait for Eager Request Ans
SE_{D0} SE_{E1} SE_{U1}	transit	Receive Data Request
SE_{D0} SE_{E2} SE_{U0}	transit	Process Next Eager Request
SE_{D0} SE_{E2} SE_{U1}	invalid	
SE_{D1} SE_{E0} SE_{U0}	transit	Receive On-Demand Request
SE_{D1} SE_{E0} SE_{U1}	invalid	
SE_{D1} SE_{E1} SE_{U0}	invalid	
SE_{D1} SE_{E1} SE_{U1}	invalid	
SE_{D1} SE_{E2} SE_{U0}	invalid	
SE_{D1} SE_{E2} SE_{U1}	invalid	
SE_{D2} SE_{E0} SE_{U0}	valid	Wait for Group On-DemandAns
SE_{D2} SE_{E0} SE_{U1}	transit	
SE_{D2} SE_{E1} SE_{U0}	invalid	
SE_{D2} SE_{E1} SE_{U1}	invalid	
SE_{D2} SE_{E2} SE_{U0}	invalid	
SE_{D2} SE_{E2} SE_{U1}	invalid	

Table 6.1: Validation of State Composition

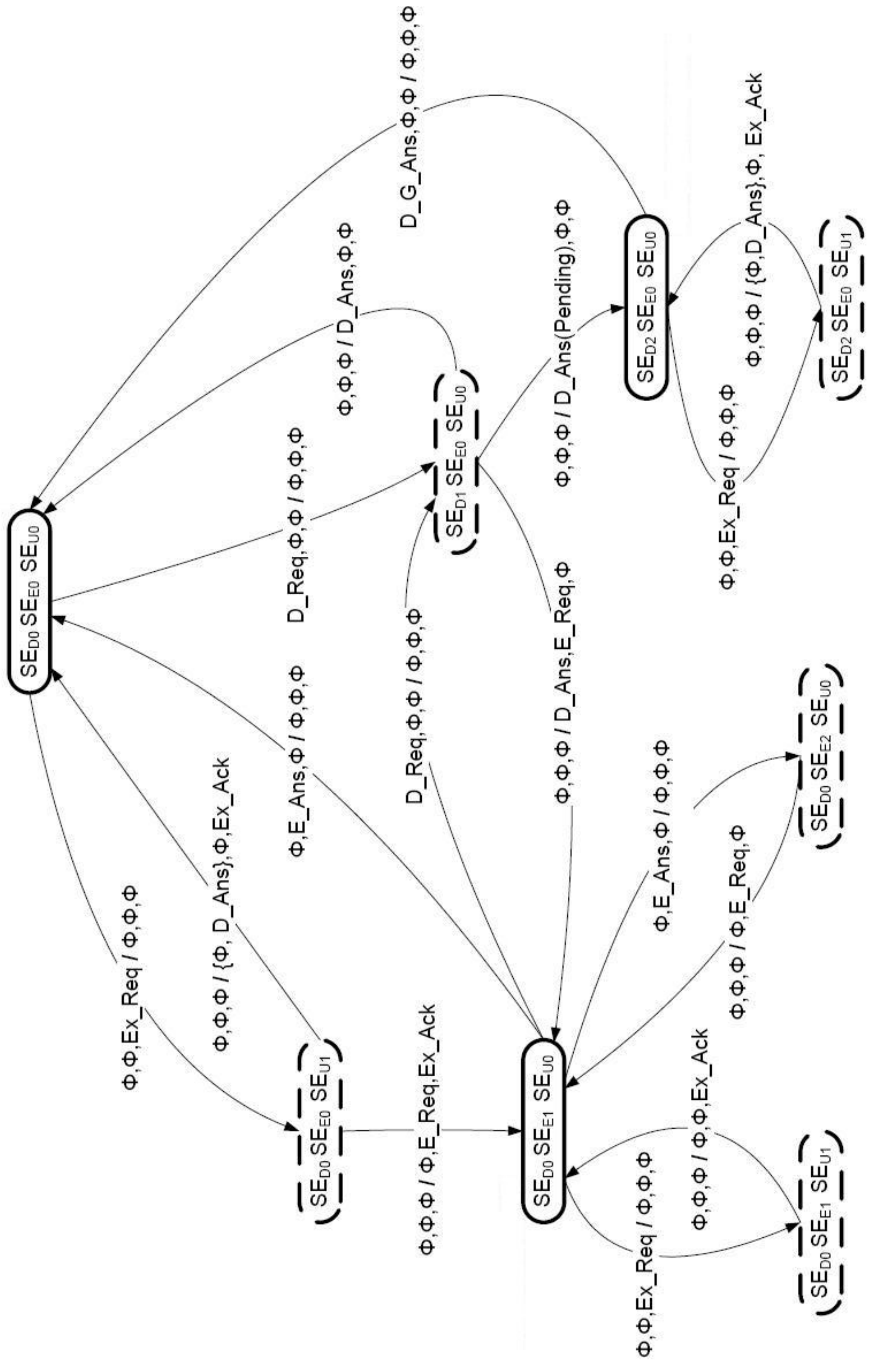


Figure 6.7: States for Data Export Processes

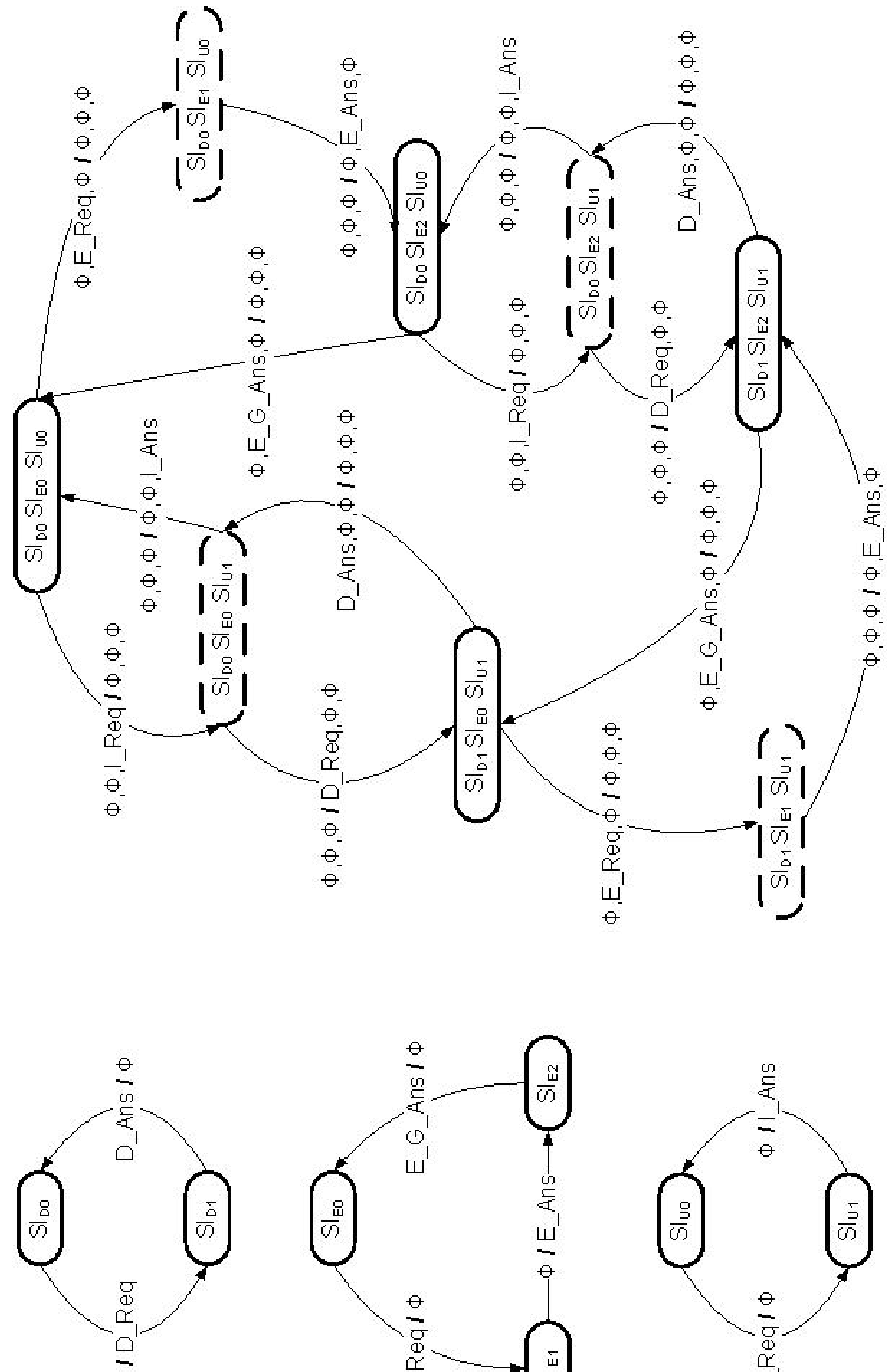


Figure 6.8: States for Data Import Process

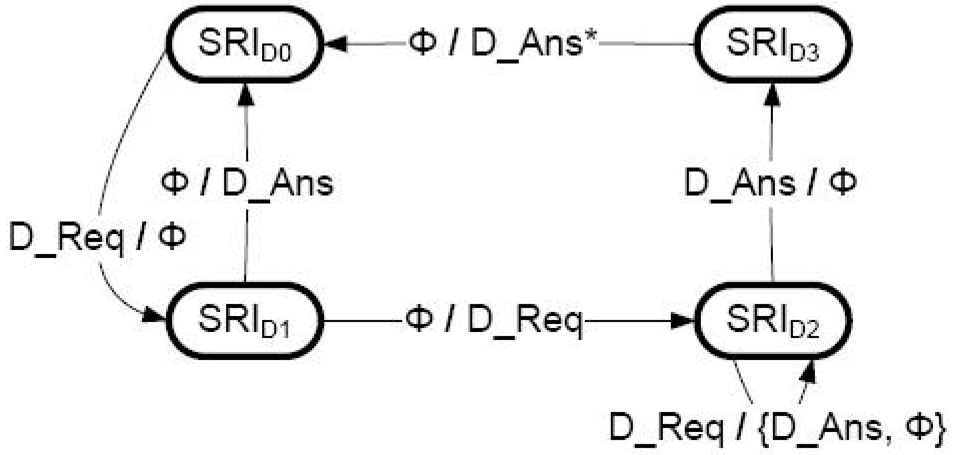


Figure 6.9: On-Demand Operation in Data Importer Rep

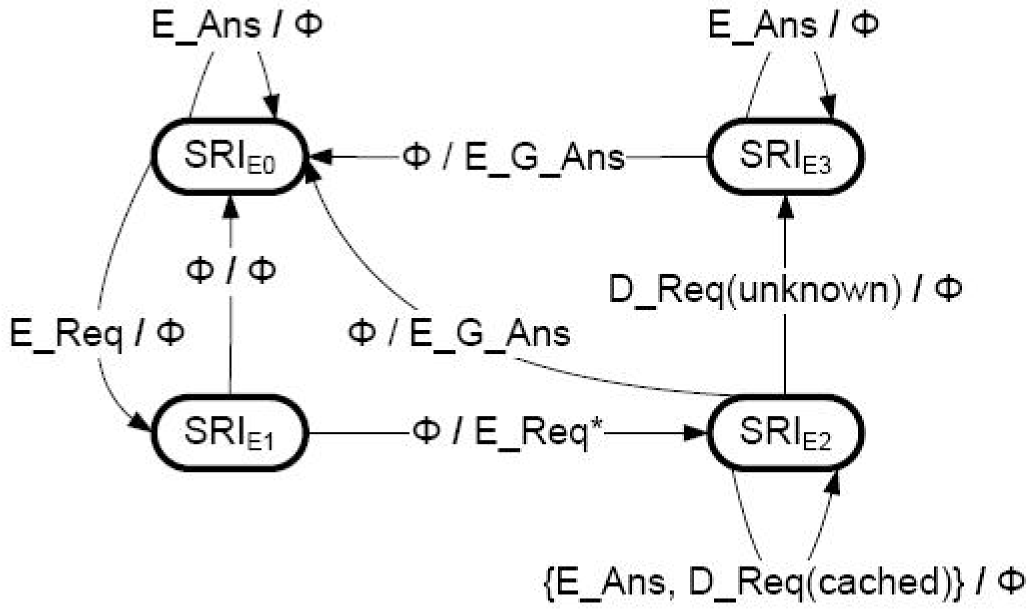


Figure 6.10: Eager Operation in Data Importer Rep

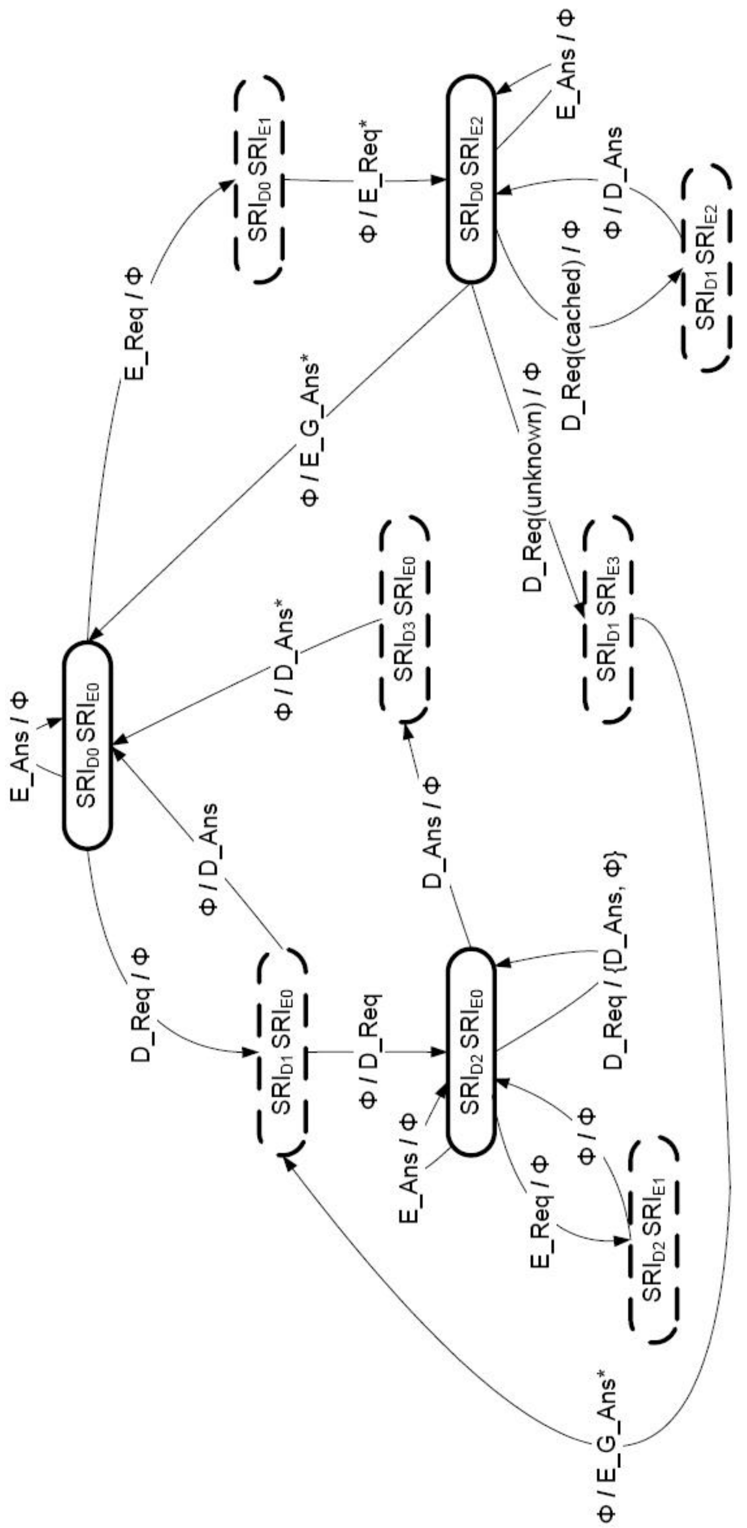

$\text{SRI}_{D0}\,\text{SRI}_{E0}$
$\text{SRI}_{D0}\,\text{SRI}_{E1}$
$\text{SRI}_{D0}\,\text{SRI}_{E2}$
$\text{SRI}_{D1}\,\text{SRI}_{E2}$
$\text{SRI}_{D3}\,\text{SRI}_{E0}$
$\text{SRI}_{D1}\,\text{SRI}_{E3}$
$\text{SRI}_{D1}\,\text{SRI}_{E0}$
$\text{SRI}_{D2}\,\text{SRI}_{E0}$
$\text{SRI}_{D2}\,\text{SRI}_{E1}$
E_Ans / Φ
E_Req / Φ
Φ / E_Req*
E_Ans / Φ
Φ / E_G_Ans*
D_Req(cached) / Φ
Φ / D_Ans
D_Req(unknown) / Φ
Φ / D_Ans*
D_Req / Φ
Φ / D_Ans
D_Ans / Φ
Φ / D_Req
D_Req / {D_Ans, Φ}
E_Ans / Φ
E_Req / Φ
Φ / Φ
Φ / E_G_Ans*

Figure 6.11: States for Data Importer Rep

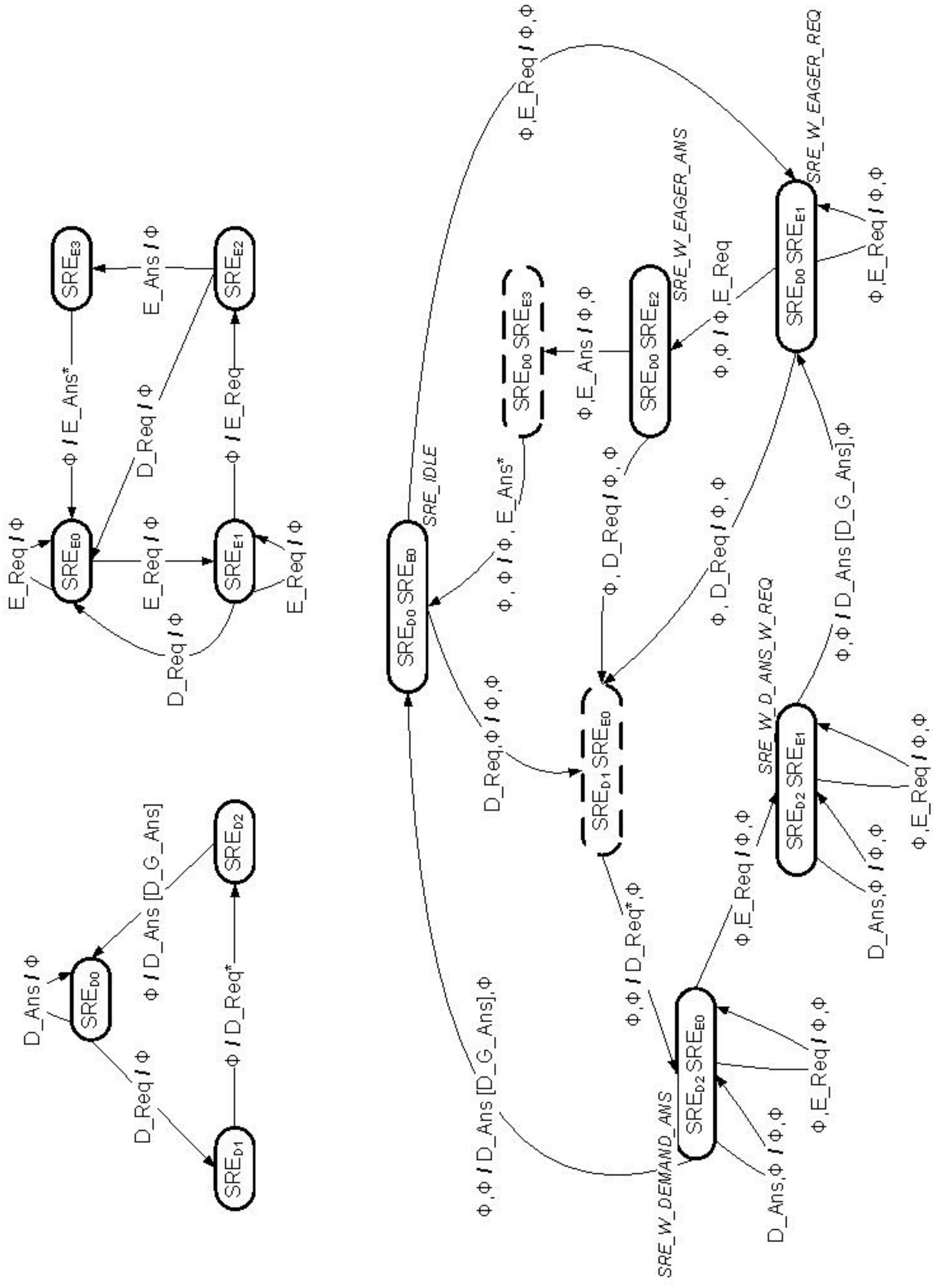


Figure 6.12: States for Data Export Rep

Chapter 7

Applications Study

Our framework has been applied to two real world coupled simulations, and the deployment and overall performance is studied in this chapter. The first is for coupling of coronal and heliospheric regions around the Sun [86]. It is an one-way coupling: the boundary data, including plasma density, temperature, flow velocity, and magnetic field, are transmitted from the coronal region to the heliospheric region. The second is part of coupling of Lyon-Fedder-Mobarry (LFM) magnetosphere model and the thermosphere-ionosphere-nested-grid (TING). It is a two-way coupling: the TING receives the electric potential field, the characteristic energy of precipitating electrons, and flux of precipitation electrons from LFM coupler and sends the Hall and Pederson conductances back to LFM coupler [74]. Both codes come from the Center for Integrated Space Weather Modeling (CISM), an NSF Science and Technology Center, and are part of a larger set of coupled models for completely characterizing the effects of solar radiation on the Earth's magnetic field.

7.1 MAS and ENLIL

The physical phenomena occurring in the solar photosphere, corona, and interplanetary space involves quite different spatial and temporal scales [86], and to have a better understanding of the underlying physics, the whole system is traditionally dissected into small pieces and each of them is modeled and investigated separately. However, to get a whole picture of those phenomena, an integrated approach, which couples related models together, is needed.

In this section, we compare different simulation approaches for coupling the coronal region and the heliospheric region. Their interface is located in the super-critical flow region, usually between 18 and 30 solar radii from the Sun, and the time-dependent data, including plasma density, temperature, flow velocity, and magnetic field, are transmitted only one way: from the coronal model to the heliospheric model. Besides the coronal model needs to simulate more complex physical processes over finer spatial and temporal scales while heliospheric model can use simpler approximation over coarser scales.

The coronal region is modeled by the Magnetohydrodynamics Around a Sphere (MAS) code from Science Applications International Corporation (SAIC) which is based on the resistive magnetohydrodynamics (MHD) equations that are solved by a semi-implicit finite-difference scheme using staggered mesh [69, 68, 70, 71, 77, 76]. The heliospheric region is modeled by the ENLIL code from National Oceanic and Atmospheric Administration (NOAA) which is based on the ideal MHD equations that are solved by an explicit finite-difference Total Variation Diminishing Lax-Friedrichs (TVDLF) scheme using cell-centered values [85, 105, 84].

7.1.1 Direct Coupling

The first approach is direct coupling. The MAS code computes 32 copies of time-varied boundary data (one for each simulation time step), and only 6 copies of them are transmitted to ENLIL code via InterComm [101]. (This is a simplified versoin.) Their pseudo codes for both models are in Table 7.1.

`for ts = 0.01 to 0.32 step 0.01`	`ts = 0.0`
`  compute domain data`	
`  if ts == 0.01 * 2 ** k`	`while(ts != 0.32)`
`    send ts to ENLIL`	`  recv ts from MAS`
`    send boundary data to ENLIL`	`  recv boundary data from MAS`
`  end if`	`  compute domain data`
`end for`	`end while`
The MAS Pseudo Code	The ENLIL Pseudo Code

Table 7.1: Direct Coupling for the MAS and the ENLIL

In this approach, the code for modeling the coronal region (the MAS) is designed to work with the code for heliospheric region (the ENLIL). Because the ENLIL, which runs on coarser scales, only needs the boundary data at certain simulation time steps, only 6 (`ts` = 0.01, 0.02, 0.04, 0.08, 0.16, or 0.32) of 32 copies of boundary data are transferred from the MAS to the ENLIL, and the decision logic for the time steps is hard-coded (the `if` statement) in the MAS source code.

Even if this direct coupling method is very efficient (because no extra data buffering and data movements are required), this method is very inflexible. If the ENLIL code needs the boundary data at different simulation time stamps, the MAS source code needs to be changed and those new time stamps (or their patterns) must be known by the MAS in advance. When the code size is over 20,000 lines, changing the source code all the time is not a good solution.

7.1.2 Timestamp-Based Coupling

The second approach is timestamp-based coupling. By using the import and export functions from our framework and writing a configuration file, rather than executing data transfers directly as in Table 7.1, both the MAS and the ENLIL can perform the same simulation without completely specifying the exact data transfers to be performed. Tables 7.2 and 7.3 show the pseudo codes and the related configuration file respectively.

In this approach, the ENLIL explicitly identifies the requested simulation time stamps and the related boundary data by calling the import function; similarly the MAS calls the export function when a new copy of boundary data are ready. It is the framework's job, based on the time stamps and the configuration file, to perform the match and the possible data transfers between export and import requests.

`for ts = 0.01 to 0.32 step 0.01`	`for ts in {0.01,0.02,0.04,`
	`            0.08,0.16,0.32}`
`  compute domain data`	`  import (ts, boundary data)`
`  export (ts, boundary data)`	`  compute domain data`
`end for`	`end for`
The MAS Pseudo Code	The ENLIL Pseudo Code

Table 7.2: Stamped-Based Coupling for the MAS and the ENLIL

Still only 6 copies of boundary data are transferred from the MAS to the ENLIL, but in this method those 6 time stamps are identified in the ENLIL source code (the `for ts` statement). It means that if the ENLIL needs the boundary data at the different

```
mas   cluster0 /home/meou/bin/mas   1

enlil cluster0 /home/meou/bin/enlil 1


mas.bt_out  enlil.br_in   FASTR   0.0001

mas.bt_out  enlil.bt_in   FASTR   0.0001

mas.bp_out  enlil.bp_in   FASTR   0.0001

mas.vr_out  enlil.vr_in   FASTR   0.0001

mas.vt_out  enlil.vt_in   FASTR   0.0001

mas.vp_out  enlil.vp_in   FASTR   0.0001

mas.rho_out enlil.de_in   FASTR   0.0001

mas.te_out  enlil.te_in   FASTR   0.0001
```

Table 7.3: The Configuration File for the MAS and the ENLIL

simulation time stamps, or if other heliospheric model is used, there is no need to change the MAS source code.

One overhead of our framework is the buffering of exported data. As mention in Sections 3.1 and 4.2, if the export data object *might* be requested in the future, it will be buffered in the framework, and if the export data are outside the known acceptable region, which was identified by an earlier received import request, it will be discarded.

In this experiment, no extra buffering is performed because the MAS runs much more slowly than the ENLIL such that the import request always happens before the matched export data are generated. It means, for the MAS, the acceptable regions always show up quite early and most mis-matched export data can be identified and be safely discarded when they are exported.

7.1.3 Experiments

To compare both coupling approaches, an experimental configuration is set as follows:

- The MAS is a sequential Fortran code. The original source code and a data input file (shown in Table 7.5) is provided by Dr. Zoran Mikić and Dr. Jon A. Linker in SAIC in San Diego California. The compiler we use is Intel Fortran compiler version 9.1.040. A parallel MPI version has been developed. It will replace the sequential version in the near future once its correctness is validated. The sequential version and the MPI version has the same export calls.

- The ENLIL is also a sequential Fortran code. Its input data is from the MAS, and the original source code is provided by Dušan Odstrčil in NOAA in Boulder Colorado. The compiler we use is GNU G95 version 0.91.

- Eight 300 *x* 1 arrays are transferred from the MAS to the ENLIL in each data transfer.

- Two Pentium 4 2.8GHz machines are used. One for the MAS, and the one for the ENLIL. Both machines are connected via Gigabit Ethernet.

- The match policy is FASTR, and the precision is 0.0001.

- Each coupling configuration runs 12 times, and the execution time of the MAS is shown in Table 7.4. (The ENLIL is around 10 times faster than the MAS.)

The execution time is very similar for both coupling approaches, although the timestamp-based one is a little bit faster in average and higher in the standard deviation. The reason is obvious. First, as mentioned earlier, the buffering overhead of timestamp-based coupling approach does not exist here due to the fact that the MAS runs much more slowly than the ENLIL, therefore the only overhead introduced by our framework is from those background utility threads. Those background overhead is quite low in general. Second, because in our multithreaded framework, one thread is for blocking send/receive and the user code runs in the foreground thread, the timestamp-based approach's performance could be better sometimes than the single thread, blocking send/receive, the direct coupling approach.

Coupling Approach	Direct	Stamped-Based
MAS Time (in second)	46.48 / 0.189	45.16 / 1.38

Table 7.4: The MAS Execution Time (Average/Standard deviation)

7.2 LFM-Coupler and TING

The Earth's thermosphere and ionosphere are a dynamically coupled system [108]. This coupling involves many physical and chemical processes of quite different spatial and temporal scales, and also heavily interacts with other parts of our atmosphere and interplanetary space. Among them, the magnetosphere plays a significant energy and momentum source for the thermosphere-ionosphere system, such as the geomagnetic storms [61].

Traditionally the thermosphere, the ionosphere, and the magnetosphere all were studied separately, and an empirical model [92] or statistical models from observations [109, 91] are used for the required input data. However to get more accurate explanations of certain phenomena, an integrated approach of those three regions is the way to go.

The Coupled Magnetosphere Ionosphere Thermosphere (CMIT) model [50, 110] combines the Thermosphere-Ionosphere Nested Grid (TING) [107] and Lyon-Fedder-

Mobarry (LFM) global magnetohydrodynamics (MHD) code [38, 75] into a two-way coupled simulation system. The TING is a high resolution, three-dimensional, time dependent model for the coupled thermosphere-ionosphere system. It is an extension of the Thermosphere/Ionosphere General Circulation Model (TIGCM) [93] from the National Center for Atmospheric Research (NCAR). The LFM global magnetosphere model solves the ideal MHD equations into a large region around the Earth, and has a non-uniform, distorted spherical grid that allows better resolution. Due to the complexity of the LFM, a separate model, LFM-Coupler, has been designed to perform necessary transformations between the LFM and the TING [74].

The LFM-Coupler and the TING is still a two-way coupling: the LFM-Coupler sends the electric potential field, the characteristic energy of precipitating electrons, as well as flux of precipitation electrons to the TING, which uses those data to compute the Hall and Pederson conductances and sends them back to the LFM-Coupler, and two different coupling approaches are considered in this section.

7.2.1 Direct Coupling

The first approach, as shown in the Table 7.6, is the direct coupling. In this approach, the simulation time is embedded in the source code (the `for ts = 1 to 10 step 1` statements in both codes) such that during iteration *i* the LFM-Coupler sends the data with the simulation time step *i* to the TING. Similarly the TING assumes the *i*th copies of received data having the simulation time step *i*.

Similarly to the coupling for the MAS and the ENLIL in Section 7.1.1, even if this method is very efficient, embedding the simulation time step into the loop index makes both codes hard to maintain and very inflexible. For example, lots efforts need to be spent in changing the LFM-Coupler source code if users want to use other models for the thermosphere-ionosphere system.

7.2.2 Timestamp-Based Coupling

The second approach, as shown in Table 7.7, is a timestamp-based coupling, which binds the simulation time step and its related data together during data exchanges between different models. By using the import and export functions from our framework and using a configuration file (shown in Table 7.8), both models still perform the same functionality but their simulation time stamps can be untied from the loop iteration indexes. This method makes it easy to compare different models for the same region. For example, fewer changes are needed in the LFM-Coupler source code if the TING model is replaced by the TIGCM model from NCAR [93].

7.2.3 Experiments

To compare both coupling approaches, an experimental configuration is set as follows:

- Both the LFM-Coupler and the TING are A++/P++ C++ code and use Overture, an object-oriented code framework for solving partial differential equation [9], from the Lawrence Livermore National Laboratory (LLNL).
- To run the LFM-Coupler and the TING without the LFM, the output from the LFM are used as data files, which are used as the input for the LFM-Coupler. The original source codes for the LFM-Coupler and the TING, and the LFM output data file are provided by Dr. Viacheslav G. Merkin in Boston University.
- Three 25 *x* 33 arrays are transferred from the LFM-Coupler to the TING and one 2 *x* 25 *x* 33 array is transferred from the TING to the LFM-Coupler in each iteration.

- Two Pentium 4 2.8GHz machines are used. One for the LFM-Coupler, and the one for the TING. Both machines are connected via Gigabit Ethernet.
- The match policy is FASTR, and the precision is 0.0001.
- Each coupling configuration runs 11 times, and each run has 10 iterations.

Table 7.9 shows total execution time for both the LFM-Coupler and the TING, and the Table 7.10 shows the computation time and data transfer time for the LFM-Coupler. The following things can be observed. First, as shown in Table 7.9, the LFM-Coupler runs slightly more slowly in both coupling approaches, and the timestamp-based approach has around 20% overhead in both the LFM-Coupler and the TING. Second, Table 7.10 shows the source of the overhead for executing the LFM-Coupler under timestamp-based coupling: the computation time is increased by 7% but the data transfer time is increased by 28%. Because this experiment is a two-way coupling and most of the LFM-Coupler execution time is for data transfers, unlike the situation in Section 7.1.3. the multithreaded non-blocking export can not help very much. In fact, the extra background service threads in our framework affect the foreground computation performance (by 7%) in this experiment, and the control message exchanges between two codes really make the foreground thread wait (block) longer for sending or receiving data.

```
$invars
option='streamer' fldtype='potential' bingauss=.false. bnfile='br.offaxis1.dat'
eqtype='parker' np1d=130 onedfile='parker1.8.pw' b0=0. rhor0=1.
bcr0type='1dchar' bcr1type='1dchar' tmax=800. ntmax=32 dtmax=.01 dtmin=.005
ifideal=0 slund=1.e5 visc=.002 rsifile=' ' rl=29. g0=.823 ifrho=1 iftemp=1
ifvdgv=1 ifpc=1 rfrac=.073,.667,1. drratio=8.,15.,1. nfrmesh=5 tfrac=.5,.67,1.
dtratio=.1,1.,7. nftmesh=5 mmodes=0 ipltxint=0 tpltxint=1.25 ihistint=5 ifprec=1
trsdump=25. upwindv=1. cfl=.4 isitype=1 dformat='hdf'
plotlist='vr','vt','vp','br','bt','bp','jr','jt','jp','p','rho','t','ap'
tnode=300.,310.,600.,610. vnode=0.,.004,.004,0. ishearprof=3 dthmax=.15 th0=1.878
ihst=9 jhst=126 khst=1 parchar=.false. ubzero=.true. nfiltub=2 he_frac=0.
radloss=0. tcond=0. ifaw=0 tnode_ch=0.,500. q0phys_ch=0.,0. tbc0=1.8e6 tbc1=0.
upwinda=1. emgflux=.true. tnode_ef=650.,700. brfile_ef='br.offaxis1.dat',
'br.offaxis5.dat' ncgmax=1500
$end
;
; Run of MAS on a 2D (axisymmetric) streamer.
```

Table 7.5: The Input File for MAS

`for ts = 1 to 10 step 1`	
`read input data from LFM`	`for ts = 1 to 10 step 1`
`perform transformations`	`recv data from LFM-C`
`send data to TING`	`compute conductances`
`recv data from TING`	`send data to LFM-C`
`send data back to LFM`	`end for`
`end for`	
The LFM-Coupler Pseudo Code	The TING Pseudo Code

Table 7.6: Direct Coupling for the LFM-Coupler and the TING

`while (not finish)`	
`import (ts, data) from LFM`	`while (not finish)`
`perform transformations`	`import (ts, data) from LFM-C`
`export (ts, data) to TING`	`compute conductances`
`import (ts, data) from TING`	`export (ts, data) to LFM-C`
`export (ts, data) to LFM`	`end while`
`end while`	
The LFM-Coupler Pseudo Code	The TING Pseudo Code

Table 7.7: Stamped-Based Coupling for the LFM-Coupler and the TING

```
LFM-C cluster0 /home/meou/bin/jpara_wrapper   1

TING  cluster0 /home/meou/bin/ionosphere      1


LFM-C.current_out   TING.current_in  FASTR   0.0001

LFM-C.density_out   TING.density_in  FASTR   0.0001

LFM-C.sspeed_out    TING.sspeed_in   FASTR   0.0001

TING.conducts_out   LFM.conducts_in  FASTR   0.0001
```

Table 7.8: The Configuration File for the LFM-Coupler and the TING

Codes	Direct Coupling	Stamped-Based Coupling
LFM-Coupler	3.63s / 0.033s	4.34s / 0.038s
TING	3.58s / 0.010s	4.29s / 0.014s

Table 7.9: The Execution Time (Average/Standard deviation)

Coupling Approach	Total Time	Computation Time	Data Transfer Time
Direct Coupling	3.63s / 0.033s	1.58s / 0.033s	2.05s / 0.023s
Stamped-Based	4.34s / 0.038s	1.70s / 0.040s	2.63s / 0.014s

Table 7.10: Dissection of LFM-Coupler Execution Time (Average/Standard deviation)

Chapter 8

Enhanced Architecture and Porting

In this chapter, the architecture of our framework will be explained first, and its implementation in various platforms, including multi-core processors, Cray XT3/XT4, Cray XMT, and IBM Blue Gene, will be discussed next.

8.1 Enhanced Architecture

Our framework, which is implemented using C++/STL, TCP sockets, as well as POSIX thread library, and the parallel data transfers between components are performed by the InterComm [101], has the following components.

Figure 8.1 shows the flowchart for executing data exports. For each data export call, if related data importer can not be found, the function will return to the user application immediately. This is an effective approach for the following issue: the data generate component can export its interface data whenever a consistent version is ready without worrying about whether the exported data object will be needed.

If related data importers exist, the framework will perform one of the following: If no pending requests are in the framework, the pattern for eager request will be checked. If the timestamp of this export data object fits the pattern, a control message for eager requests will be issued, and if the eager request is granted later, the predicted data will be sent to the request process.

However if earlier pending requests exist in the framework, they will be re-evaluated (re-approximate match), and if the match answers are Yes or Never, corresponding actions will perform. If the match answers are still Pending, they will be pushed back to the pending buffer. Obviously, for those slow data export process, the same pending request will be re-evaluated again and again, and the buddy-help method, mentioned in Chapter 4, is an effective approach for this situation.

The approximate match control thread, which is shown in Figure 8.2 and runs on *each* data export process, handles data request events. Approximate match will be performed for each received data requests, based on the connection-wise match policy and tolerance in a framework-level configuration file, and there are three possible answers: *Never*, *Yes*,and *Pending*.

If the answer is Never, it means, based on user-defined match policy and the related tolerance, the matched timestamp can neither be found from already exported timestamps now nor be generated in the future. In this case, no data transfer will happen and it is the requester's responsibility to handle this situation. If the answer is Pending, it means, based on currently exported timestamps, the *matched* one can not be decided, such as the example in Figure 3.4 of Section 3.2. In this case, the requested timestamp will be saved in the framework and no data transfer will be triggered. If the answer is Yes, the matched timestamp, will be sent to the request process via a reply control message, and the matched data object will be transferred to the request process by calling IC_Send, which is the function supplied by InterComm for parallel data transfers. Additionally, if the pattern for eager requests has not be formed, the matched stamp will be used as an input of learning. If a pattern shows up and a predicted timestamp has already been exported, a control message for eager requests will be issued. (If the eager request is granted later, the predicted data will be sent to the request process.)

Figure 8.3 shows the flowchart for executing data import. During the execution of data import functions, if requested data can not be found, either because of no related import component, or because of receiving Never answer, no data transfer will happen and the control will be returned to the user application. However if the requested data can be found, the data will be copies to user space either from framework buffer (if earlier predictions are right and the data object have been transferred), or from the data export component. (for the on-demand data transfer)

The import control thread is also shown in Figure 8.3. Compared to the Approximate Match Control Thread in Figure 8.2, the import control thread is much simpler. After receiving the eager request, it replies *Granted* if the available memory can be allocated, and *Not Granted* if can not.

To support this collective property at runtime, we employ an extra thread in each program, called the representative (or *rep* for short), to act as a low-overhead control gateway. Not only does the rep forwards, reduces, and caches control messages between the processes in the same program and the reps in other programs, it also participates the buddy-help optimization mentioned in Chapter 4. The state diagrams of data import rep and data export rep are already shown in Figures 6.11 and 6.12 respectively.

8.2 Porting to Other Architectures

Our framework, which is currently implemented for Linux clusters, needs support for POSIX thread and user-level server sockets from operating systems. Precisely each process is a multi-threaded process, (the user application is the foreground thread and all control threads mentioned earlier run in the background.) and a user-level server port is used to accept incoming control messages. Besides, for each program, one process has to run the representative thread and an extra user-level server port is needed for the work done by the representative.

The porting strategy for other environments, including x86-based multicore processors, Cray XT4, Cray XMT, and IBM BlueGene, are discuss as follows.

8.2.1 Multicore x86

The multi-core processor (MCP) is the design choice by Intel, AMD, and many others for better performance, energy efficiency, and production reliability [14, 51]. In this architecture, one machine can have multiple processor chips, and one chip can have more than one computation cores. Besides three layers of cache memory is possible: the L1 cache is inside the computation core, the L2 cache is outside the cores, but on the chip, and the L3 cache is between the chips and the main memory. Both AMD and Intel have quad-core x86 architecture in 2007, and a brief comparison is as shown in Table 8.1. Main differences are (1) the support for different SIMD instruction sets, (2) the size of L1 cache, (3) whether the L2 cache is shared or not, and (4) whether L3 cache exists or not.

One of the challenges for MCP now is how to use those multiple cores. Intel's solution is Threading Building Blocks (TBB) [90], which is a mix of shared memory approach and Pthread library. Our multi-threaded framework is a natural fit for those multi-core processors. For x86 MCP architecture, control messages can be executed on one or two cores and the user application computation can run on remaining cores. However, to improve the runtime performance, the mapping between the hardware cores and the software threads is an issue should be investigated further, but is outside the

Processors	AMD Opteron 2300 Series	Intel Xeon 5300
SMP Capabilities	Up to 2 Sockets/8 Cores	Up to 2 Sockets/8 Cores
L1 size per core(max)	64KB (D) + 64KB (I)	32KB (D) + 32KB (I)
L2 size (max)	512KB , per core	4MB (shared) x 2
L3 size (max)	2MB (shared)	–
SIMD Set Support	SSE, SSE2, SSE3, SSE4A	SSE, SSE2, SSE3

Table 8.1: AMD and Intel Multi-core Processors

scope of this dissertation.

8.2.2 Cray XT4

The Cray XT4 [28] is a distributed memory massively parallel supercomputer designed by Cray Inc. The XT4 is comprised of between 548 and 30508 processing elements (PEs), where each PE is comprised of one 2.6GHz AMD 64-bit Opteron processor (single, dual, or quad core) coupled with a custom SeaStar2 communications chip, and between 1 and 8 GB of RAM. The PowerPC 440-based SeaStar2 device provides a 6.4 gigabyte per second connection to the processor across HyperTransport, as well as six 7.6 Gigabyte per second links to neighboring PEs. The PEs are arranged in a 3-dimensional torus topology, with 548 PEs in 6 cabinets. The performance of each XT4 model will vary with the speed and number of processors installed, and the Cray datasheets describes a 320 cabinet model as providing 318 teraflops of peak performance.

The XT4 runs an operating system called UNICOS/lc, which has two components: a full-featured Linux for the service PEs and the Catamount microkernel for compute PEs. The service PE run a full-featured Linux, and can be configured to provide login, I/O, system, or network services. The Catamount microkernel in compute PEs is a light-weight computational environment, which minimizes the system overhead and the following features are not supported [29]:

- Pipes, sockets, remote procedure calls, or other TCP/IP communications.
- Dynamic process control (such as `exec()`, `popen()`, and `fork()`).
- Dynamic loading of executable images.
- Threading.
- The proc files such as cpuinfo and meminfo.
- The `ptrace()` system call.
- The `mmap()` function.
- The `profil()` function.
- Any of the `getpwd()` family of library calls.
- Terminal control.
- Any functions that requires a daemon.
- Any functions that requires a database, such as `ndb()`.
- Limited support for signals and `ioctl()`.

To port our framework to Cray XT4, the approach used by Dart [33] can be applied. In this approach, (1) both compute PEs and network service PEs are required, and (2) our framework is split into two parts: the part for domain computations and the part for runtime message exchanges. All domain computations are performed by allocated compute PEs, all control message exchanges are performed by allocated network service

PEs, and data exchange to/from compute PEs to other programs are through network service PEs. However the *right* ratio between the number of compute PEs and the number of network service PEs is important and would have to be experimented with a real machine.

8.2.3 Cray XMT

The Cray XMT [26, 81] supercomputing system, the third generation of the Cray MTA supercomputer architecture originally developed by Tera, is a scalable massively multithreaded platform with a shared memory architecture for large-scale data analysis. It scales from 24 to over 8000 processors providing over one million simultaneous threads and 128 terabytes of shared memory, and an early model has been shipped to Pacific Northwest National Laboratory on Sep 19 2007 [27].

The design of Cray XMT is based on AMD Torrenza technology to populate the AMD Opteron sockets with custom Cray Threadstorm chips developed for multithreaded processing. A single Cray Threadstorm processor can sustain 128 simultaneous threads and is connected with up to 16 GB of memory that is globally accessible by any other Cray Threadstorm processor in the system. Each Cray Threadstorm processor is directly connected to a dedicated Cray SeaStar2 chip, which is also used in Cray XT4. Besides the Cray XMT platform includes separate AMD Opteron-based service blades that can be configured for I/O, login, network or system functions.

Support for multi-threading makes the porting our framework to Cray XMT easier – each process can perform both application computation and control message exchanges; however the effective usage of threads for domain computation would be an important issue.

8.2.4 BlueGene Architecture

The Blue Gene/L (BG/L) supercomputer [43, 106, 87, 1, 48, 78, 22, 99] is a massively parallel system developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). It is designed to reach operating speeds in the PFLOPS (petaFLOPS) range, and currently (November 2007) reaching peak speeds over 596 TFLOPS (Tera-FLOPS) on the Top500 List [104].

The Blue Gene/L supercomputer is unique in the following aspects [13]:

- Trading the speed of processors for lower power consumption.
- Dual processors are in each compute node. It can operate at one of the two modes: co-processor (1 user process/node: computation and communication work is shared by two processors) or virtual node (2 user processes/node)
- System-on-a-chip design.
- A large number of nodes (scalable in increments of 1024 up to at least 212,992).
- Three-dimensional torus interconnect with auxiliary networks for global communications, I/O, and management.
- Lightweight OS per node for minimum system overhead.

Each Compute or I/O node is an application specific integrated circuit (ASIC) with associated DRAM memory chips. The ASIC is composed of (1)two 700 MHz PowerPC 440 embedded processors, each of which has a double-pipeline-double-precision Floating Point Unit (FPU), and (2) a cache sub-system with DRAM controller and the communication logic sub-systems. The dual FPUs give each BlueGene/L node a theoretical peak performance of 5.6 GFLOPS (Giga-FLOPS), and node CPUs are not cache coherent with one another.

Two compute nodes are packaged in each compute card, and each node board has 16 compute cards with up to 2 I/O nodes. There are 32 node boards in each cabinet, and up to 1024 compute nodes can be put in the standard 19" cabinet. Each Blue Gene/L node is attached to three parallel communications networks: a 3D toroidal network for peer-to-peer communication, a collective network for collective communication, and barrier operations are performed via a global interrupt network.

The operating system in I/O nodes is Linux, which provides communication with the world via an Ethernet network. The file system operations of the compute nodes is also performed by the I/O nodes. The compute node in Blue Gene/L runs a minimal operating system, which can only run one process at a time with limited support of POSIX system calls. (POSIX threads are not supported.) To run multiple programs concurrently in a Blue Gene/L system, the system must be partitioned into electronically isolated sets of nodes. The number of nodes in a partition must be an integer power of 2, and must contain at least 32 nodes.

To port our framework to Blue Gene/L, the method for Cray XT4 can be used. In this approach, (1) both compute nodes and I/O nodes are required, and (2) our framework can be split into two parts: the part for domain computations and the part for runtime message exchanges. All domain computations are performed by allocated compute nodes, all control message exchanges are performed by allocated I/O nodes, and data exchange to/from compute nodes to other programs are through I/O nodes.

IBM unveiled Blue Gene/P, the second generation of the Blue Gene supercomputer, on June 26 2007. It is designed to run continuously at 1 PFLOPS (petaFLOPS) and can be configured to reach speeds in excess of 3 PFLOPS. Each Blue Gene/P chip consists of four 850 MHz PowerPC 450 processors, and the can be scaled to an 884,736-processor, 216-rack cluster to achieve 3-PFLOPS performance. Currently (November 2007) the

Jugene in Forschungszentrum Juelich has 65,536 processors and its peak performance is over 222 TFLOPS [104]. Besides, Blue Gene/P has limited supports for POSIX threads (Blue Gene/L does not). Each Blue Gene/P chip can have up to four Pthreads [42].

The support for POSIX threads in Blue Gene/P makes the porting of our framework easier. To use our framework in Blue Gene/P, (1) both compute nodes and I/O nodes are required, (2) each program runs its rep process on one of the allocated compute nodes in which one thread is used for the rep process, one thread is used for control message exchanges, and the other two threads perform domain computations, (3) all other allocated compute nodes use up to 3 threads to run domain computations, and the remaining one thread is used to exchange control messages, and (4) data exchange to/from compute nodes to other programs are through I/O nodes.

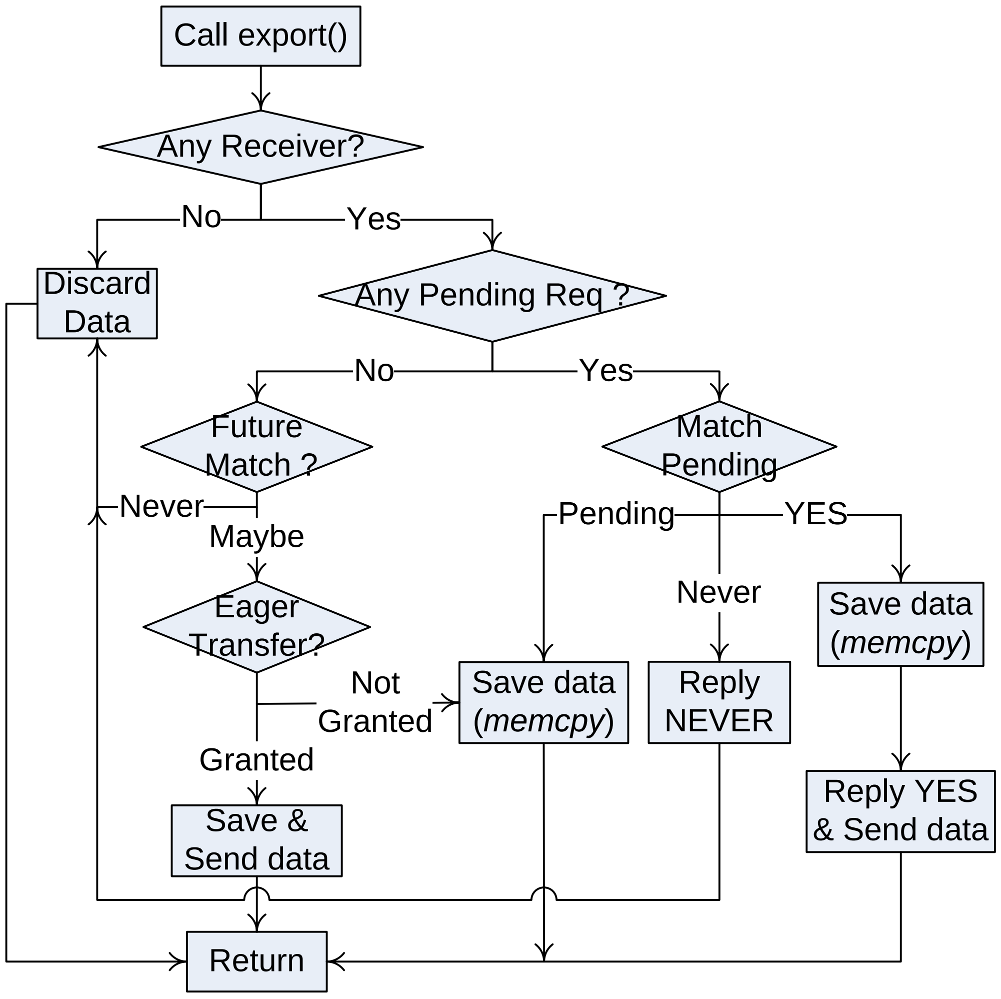


Figure 8.1: Data Export Function

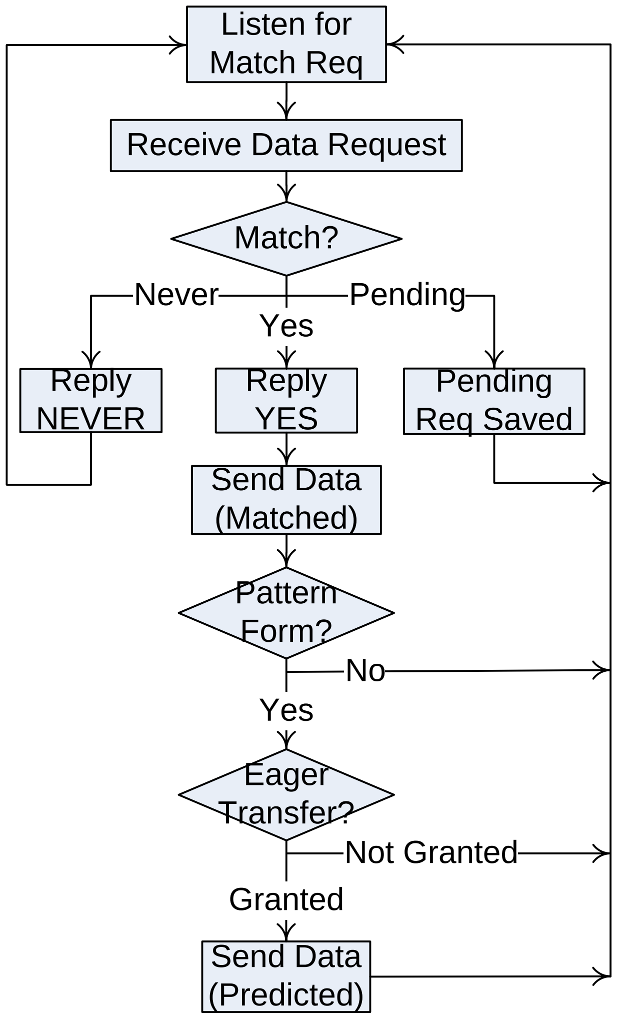


Figure 8.2: Approximate Match Control Thread

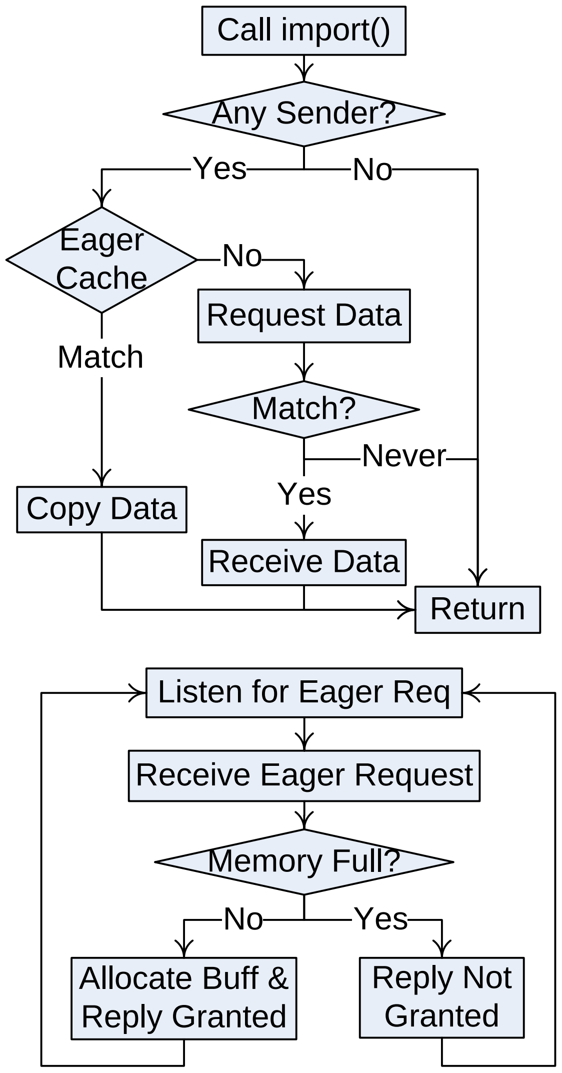


Figure 8.3: Data Import Function/Import Control Thread

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this dissertation, I supported the following thesis: *it is possible to provide flexible and efficient mechanisms for control of data transfers between coupled parallel programs*, and conclude the associated work as follows.

9.1.1 Basic Algorithm and Architecture

First, we suggest (1) a runtime-based approximate match algorithm, and (2) an approach to separating coupling information from applications domain computation, to support data exchanges between different simulation time scales components.

Once the coupling information is extracted from application source codes, data generating component can export desired interface data whenever a consistent version of the data across the parallel component is produced. The component does not have to concern itself about which and how many components will receive the data, or even whether data transfers will actually occur. Similarly, data consuming components can prepare to import a new version of required interface data whenever they are needed without knowing anything about the corresponding exporters.

The connection-wise approximate match algorithm is performed by data export components whenever timestamped data import requests are received. Based on the related match criteria, which are composed of match policies and tolerances, approximate matched timestamps can be decided if exact ones can not be found. Currently eight different match policies are supported in the framework and it is possible to plug-in user-defined ones. Experimental results shows that the incurred overhead by approximate match algorithm is very low.

9.1.2 Collective Optimization

Even the basic framework described earlier offers flexibility and versatility for building and deploying large-scale multi-physics simulation systems, the overall performance of this runtime-based approach might be a concern.

Basically the basic framework describes a temporal consistency model in which each exported data object must be buffered by the runtime system implementing the model, until there is no possibility that an object will be requested by an importing component.

Although this approach ensures the correctness of the data exchange mechanism, overall system performance may suffer from unnecessary buffering, when one process in a data exporting (parallel) component performs the collective export operation early relative to the other processes (i.e. it is the first process to execute the export runtime library call). In that case, other processes can use the information, which is the approximate match answer produced by other faster processes in the same exporting parallel components, to avoid some unnecessary memory copies. In the optimal situation, the slower processes can only save the requested export data objects and skip the memory copies for all other unrequested data objects.

9.1.3 Eager Transfer and Distributed Approximate Match

The performance of our basic framework is very sensitive to the available bandwidth and end-to-end latency of the network connecting the various components of a coupled simulation. To cope with those issues, we describe and analyze two methods, eager transfer and distributed approximate match, to deal with the network bandwidth and latency.

Transferring predicted data in advance, which we call eager transfer, can effectively solve the bandwidth problem. Our analysis and experiments suggested that, on average, the lower the available network bandwidth, the more time can be saved by eager transfer.

However, eager transfer does not solve the network latency problem. The reason is that, although the data transfer time is hidden behind the applications computation, a round-trip time between components for control messages, to decide which data object is the one to be transferred, is still needed. That is, even with eager transfer, network latency still plays an important role in overall performance,

Here we introduce another algorithm, distributed approximate match, which is a distributed version of our basic approximate match algorithm, to handle the latency issue. It contains two parts: inverse approximate match, running in the component that supplies the data, and the range check, running in the component that consumes the data. The new distributed algorithm has the same functionality as our previous approximate match algorithm, and its performance can be independent to the network latency in the best case. Our experimental results show that by combining both methods, the overall performance of the framework can be improved significantly.

9.1.4 Control Protocol

One of important features for the above framework is that, whenever a data object is exported or imported, its associated (simulation) timestamp is also required to passed to the framework. Those timestamps are keys for the underlining priority-based control protocol.

Whenever user applications issue (on-demand) timestamped data requests, rather than directly performing collective communications, as in traditional tightly coupled frameworks, control messages are first exchanged between data exporters and importers, and if requests can be satisfied, matched export timestamps will be used as tags to perform associated collective communication operations. Eager transfer (with distributed approximate match), improves the runtime performance lots when the effect by network bandwidth and latency can not be ignored. That is a common situation when the components run on different machines at different locations (as in Grid computing).

To support both on-demand and eager transfer operations, the underlining architecture is designed such that control messages can be initiated either by both parties – control message for on-demand data requests are initiated by data import components and for eager transfer are by data export components. This two-sided initialization approach make the framework flexible and efficient, by the price of a complicated control protocol, especially for the coverage of all possible interactions between two the modes. To handle this difficulty, we will describe a protocol construction method, which is based on validating all possible compositions of smaller protocols for each of the modes.

9.1.5 Applications Study

Two real world simulation codes have been applied to our framework. In both cases, we show it is not hard to port original source codes into our framework, and once the deployment finishes, it is easy to change the participants — only the configuration file need to be modified and the source codes of related components can be kept untouched.

Besides, in the study for the MAS and the ENLIL, we shows that, for the computation-based program such as the MAS, the overhead introduced by our framework is very small and the overall performance of the original direct coupling and that of the stamped-based coupling is very similar. In the study of the LFM-Coupler and the TING, we shows that, for the data transfer-based program such as the LFM-Coupler, the introduced overhead is hard to ignore, besides, not only the background data transfer needs more time, but the performance of the foreground computation is also effected.

9.1.6 Enhanced Architecture

Our framework is implemented using C++/STL, TCP sockets, POSIX thread library, as well as InterComm, and it contains the following important elements.

For each data export call, if related data importer can not be found, the function will return to the user application immediately. If related data importers exist, the framework will perform one of the following: If no pending requests are in the framework, the pattern for eager request will be checked. If the timestamp of this export data object fits the pattern, a control message for eager requests will be issued, and if the eager request is granted later, the predicted data will be sent to the request process.

The approximate match control thread handles data request events. After receiving data requests, the approximate match will be performed and return possible answers: *Never*, *Yes*,and *Pending*. If the answer is Never, no data transfer will happen. If the answer is Pending, the requested timestamp will be saved in the framework and no data transfer will be triggered. If the answer is Yes, the matched timestamp and data will be sent to the request process.

To support this collective property at runtime, we employ an extra thread in each program, called the representative (or *rep* for short), to act as a low-overhead control gateway. Not only does the rep forwards, reduces,and caches control messages between the processes in the same program and the reps in other programs, it also participates the collective optimization.

9.1.7 Porting to Other Architectures

Our framework, which is implemented for Linux clusters now, needs support for POSIX thread and user-level server sockets from operation systems, and the porting strategy for other architectures, including multicore x86, Cray XT4, Cray XMT, and IBM BlueGene, are considered here.

The multi-core processor (MCP) is the design choice by Intel, AMD, and many other companies. In this architecture, one machine can have multiple processor chips, and one chip can have more than one computation cores. Our multi-threaded framework is a nature fit for those multi-core processors. For x86 MCP architecture, control messages can be executed on one or two cores and the user application computation can run on remaining cores.

The Cray XT4 [28] is a distributed memory massively parallel supercomputer, and in compute processing elements (PEs), it runs a light-weighted Catamount microkernel which has no supports for POSIX threads and TCP/IP sockets. To port our framework to Cray XT4, we can split our framework into two parts: the part for domain computations and the part for runtime message exchanges. All domain computations are performed by allocated compute PEs, all control message exchanges are performed by allocated network service PEs, and data exchange to/from compute PEs to other programs are through network service PEs. The Cray XMT supercomputing system is a scalable massively multithreaded platform with a shared memory architecture for large-scale data analysis. Support for multi-threading makes the porting our framework to Cray XMT easier – each process can perform both application computation and control message exchanges.

The Blue Gene/L (BG/L) supercomputer is a massively parallel system developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). As Cray XT4, is also run a light-weighted operating system (without the support for POSIX threads) in compute nodes. Therefore the similar approach for porting our framework to XT4 can also be applied to Blue Gene/L. Blue Gene/P is the second generation of the Blue Gene supercomputer and has limited supports for POSIX threads. Each Blue Gene/P chip can have up to four Pthreads, and it makes the porting of our framework easier. To use our framework in Blue Gene/P, we need to split our framework into three parts. (1) the rep thread runs on one thread, (2) integrate all other background services into another threads, (3) leave the reaming two threads for domain data computations, and (4) data exchange to/from compute nodes to other programs are through I/O nodes.

9.2 Future Work

Our framework has some nice features and properties, such as loosely coupled approach, low overhead approximate match, runtime-based collective buffering, eager transfer and distributed approximate match; however there are still some extensions can be made.

9.2.1 Runtime Connections Management

Our framework uses configuration files to identify the connections between exporters and importers, and as shown in Chapter 3 this method effectively separates coupling information from application source codes.

Currently configuration files are used to establish connections at runtime, and those connections will be kept open until the framework finishes. However if the runtime connection management can be provided (by the representative), the applications can establish/disconnect their connections based on their own requirements, and the runtime resources can be utilized more efficiently.

Besides this approach really benefits those codes which involves more than one connections but only some of them are used to exchange data at anytime. For example, some scientific programs save the simulation data in files and use them as the input for the visualization programs. If the runtime connection management is supported, those components can establish connections with other simulation components in the first phase and then switch the connections to the visualization components later.

9.2.2 Predictions for Eager Transfer

When the importer runs more slowly than the exporter and the request pattern has been identified, the eager transfer will send predicted data and meta-data, which is the output from the inverse approximate match, to the exporter before data import function is called. As shown in Chapter 5, when the prediction is correct, the overall performance can be greatly improved.

Currently we only use a simple pattern predictor: if the interval p between successive request timestamps are kept the same for certain times, which is 3 in default and can be changed by application programs, we assume future request timestamps are periodic with the period of p. Clearly this predictor is not good enough in general, an effective and low overhead prediction algorithm would be a great extension for our current framework.

9.2.3 Scalability

The representative (or *rep* for short) thread plays an important role in our framework. For each program, the rep act as a gateway for control messages: it forwards, reduces,and caches control messages between the processes in the same program and the reps in other programs. Currently a flat structure is used for exchange messages inside each program: For example, when an on-demand data request message is received by an importer rep, the rep forwards the request to *all* of the processes in the same program, and receives the response from *all* of them later.

When the number of processes is hundreds, thousands or more, clearly the flat structure message passing between the rep and all the processes in the same program is not a good choice. In this case, a tree-based approach should be considered, and it would be helpful in performance if some of the processes are also involved in rep's work.

BIBLIOGRAPHY

[1] Narasimha R. Adiga, Matthias A. Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D. Steinmacher-Burow, Todd Takken, Mickey Tsao, and Pavlos Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2-3):265–276, 2005.

[2] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, 2007.

[3] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 183–193, 2007.

[4] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[5] Francis J. Alexander, Daniel M. Tartakovsky, and Alejandro L. Garcia. Noise in algorithm refinement methods. *IEEE Comput. Sci. Eng.*, 7(3):32–38, 2005.

[6] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a Common Component Architecture for high-performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*. IEEE Computer Society Press, 1999.

[7] Ghassem R. Asrar. A pathway to decisions on Earth's environment and natural resources. *IEEE Comput. Sci. Eng.*, 6(2):13–16, January/Feburary 2004.

[8] Jim Basney. A distributed implementation of the C-Linda programming language, May 1995. http://www.cs.oberlin.edu/ jbasney/honors/thesis.html.

[9] Federico Bassetti, David Brown, Kei Davis, William Henshaw, and Dan Quinlan. Overture: an object-oriented framework for high performance scientific computing. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–9, 1998.

[10] Peter Beckman, Pat Fasel, William Humphrey, and Sue Mniszewski. Efficient coupling of parallel applications using PAWS. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 215–222. IEEE Computer Society Press, July 1998.

[11] Christian Bell, Wei-Yu Chen, Dan Bonachea, and Katherine Yelick. Evaluating support for global address space languages on the cray x1. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 184–195, 2004.

[12] Felipe Bertrand, Randall Bramley, Alan Sussman, David E. Bernholdt, James A. Kohl, Jay W. Larson, and Kostadin B. Damevski. Data redistribution and remote

method invocation in parallel component architectures. In *Proceedings of the 2005 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*. IEEE Computer Society Press, 2005.

[13] Blue Gene. http://en.wikipedia.org/wiki/BlueGene.

[14] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[15] David R. Butenhof. *Programming With POSIX Threads*. Addison-Wesley Professional, USA, 1997.

[16] Francois Cantonnet, Tarek A. El-Ghazawi, Pascal Lorenz, and Jaafer Gaber. Fast address translation techniques for distributed shared memory compilers. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 52.2, 2005.

[17] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[18] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[19] K. Mani Chandy, Ian Foster, Ken Kennedy, Charles Koelbel, and Chau-Wen Tseng. Integrated support for task and data parallelism. *Journal of Supercomputing Applications*, 8(2), 1994. Also available as CRPC Technical Report CRPC-TR93430.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10:

an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[22] Siddhartha Chatterjee, Leonardo R. Bachega, Peter Bergner, Kenneth A. Dockser, John A. Gunnels, Manish Gupta, Fred G. Gustavson, Christopher A. Lapkowski, Gary K. Liu, Mark P. Mendell, Rohini D. Nair, Charles D. Wait, T. J. Christopher Ward, and Peng Wu. Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L. *IBM Journal of Research and Development*, 49(2-3):377–392, 2005.

[23] Wei-Yu Chen, Arvind Krishnamurthy, and Katherine A. Yelick. Polynomial-time algorithms for enforcing sequential consistency in spmd programs with arrays. In *LCPC*, pages 340–356, 2003.

[24] Nancy Collins, Gerhard Theurich, Cecelia Deluca, Max Suarez, Atanas Trayanov, V. Balaji, Peggy Li, Weiyu Yang, Chris Hill, and Arlindo Da Silva. Design and implementation of components in the Earth system modeling framework. *Int. J. High Perform. Comput. Appl.*, 19(3):341–350, 2005.

[25] Cray C/C++ reference manual. http://www.cray.com/craydoc/manuals/004-2179-003/html-004-2179-003/.

[26] Cray XMT platform. http://www.cray.com/products/xmt/index.html.

[27] Pacific northwest national laboratory acquires cray XMT supercomputer. http://www.pnl.gov/topstory.asp?id=271.

[28] Cray XT4 and XT3 supercomputers. http://www.cray.com/products/xt4/index.html.

[29] Cray XT series programming environment user's guide. http://docs.cray.com/books/S-2396-15/S-2396-15.pdf.

[30] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol Composition Logic (PCL). *Electron. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[31] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 40–47, New York, NY, USA, 2006. ACM.

[32] Roxana E. Diaconescu and Hans P. Zima. An approach to data distributions in Chapel. *Int. J. High Perform. Comput. Appl.*, 21(3):313–335, 2007.

[33] Ciprian Docan, Manish Parashar, J. Lofstead, K. Schwan, and S. Klasky. Dart: An infrastructure for high speed asynchronous data transfers, Oct 2007. submitted for publication.

[34] Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness, 2001.

[35] Guy Edjlali, Alan Sussman, and Joel Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the Eleventh International Parallel Processing Symposium*. IEEE Computer Society Press, April 1997.

[36] Heiko Eifeldt. POSIX: a developer's view of standards. In *ATEC'97: Proceedings of the Annual Technical Conference on Proceedings of the USENIX 1997 Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 1997. USENIX Association.

[37] Patricia Fasel and Susan Mniszewski. PAWS: Collective interactions and data transfers. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, Washington, DC, USA, 2001. IEEE Computer Society.

[38] J. A. Fedder, S.P. Slinker, J.G. Lyon, and R.D. Elphinstone. Global numerical simulation of the growth phase and the expansion onset for substorm observed by viking. *J. Geophys. Res.*, 100:19083–19094, 1995.

[39] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1, June 1999. RFC 2616.

[40] Ian Foster. Compositional parallel programming languages. *ACM Trans. Program. Lang. Syst.*, 18(4):454–476, 1996.

[41] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall, Englewood Cliffs, NJ, USA, 1990.

[42] Alan Gara. Blue Gene: A next generation supercomputer (BlueGene/P). In *Computing in Atmospheric Sciences Workshop 2007 (CAS2K7)*, 2007.

[43] Alan Gara, Matthias A. Blumrich, Dong Chen, George L.-T. Chiu, Paul Coteus, Mark Giampapa, Ruud A. Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V. Kopcsay, Thomas A. Liebsch, Martin Ohmacht, Burkhard D. Steinmacher-

Burow, Todd Takken, and Pavlos Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2-3):195–212, 2005.

[44] Intrepid technology, inc. GCC/UPC compiler. http://www.intrepid.com/upc/.

[45] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. CUMULVS: Providing fault tolerance, visualization and steering of parallel applications. *Int. J. High-Perform. Comput. Appl.*, 11(3):224–235, August 1997.

[46] William L. George, John G. Hagedorn, and Judith E. Devaney. IMPI: Making MPI interoperable. *Journal of Research of the National Institute of Standards and Technology*, 105(3):343–428, 2000.

[47] William L. George, John G. Hagedorn, and Judith E. Devaney. Parallel programming with interoperable MPI. *Dr. Dobb's Journal*, (357):49–53, February 2004.

[48] Mark Giampapa, Ralph Bellofatto, Matthias A. Blumrich, Dong Chen, Marc Boris Dombrowa, Alan Gara, Ruud A. Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V. Kopcsay, Ben J. Nathanson, Burkhard D. Steinmacher-Burow, Martin Ohmacht, Valentina Salapura, and Pavlos Vranas. Blue Gene/L advanced diagnostics environment. *IBM Journal of Research and Development*, 49(2-3):319–332, 2005.

[49] Tamas I. Gombosi, Kenneth G. Powell, Darren L. De Zeeuw, C. Robert Clauer, Kenneth C. Hansen, Ward B. Manchester, Aaron J. Ridley, Ilia I. Roussev, Igor V. Sokolov, Quentin F. Stout, and Gábor Tóth. Solution-adaptive magnetohydrodynamics for space plasmas: Sun-to-Earth simulations. *IEEE Comput. Sci. Eng.*, 6(2):14–35, 2004.

[50] C.C. Goodrich, A.L. Sussman, J.G. Lyon, M.A. Shay, and P.A. Cassak. The CISM code coupling strategy. *J. Atm. Terr. Phys.*, 66:1469–1479, 2004.

[51] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, USA, 2006. 4th edition.

[52] Paul N. Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Amir Kamil, Ben Liblit, Geoff Pike, Jimmy Su, and Katherine Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15, U.C. Berkeley Tech Report, 2005.

[53] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[54] HP UPC version 2.0 for Tru64 UNIX. http://h30097.www3.hp.com/upc/.

[55] James M. Hyman. Patch dynamics for multiscale problems. *IEEE Comput. Sci. Eng.*, 7(3):47–53, 2005.

[56] ITU-T. Specification and Description Language (SDL), 1993. Recommendation Z.100.

[57] Xiangmin Jiao, Michael T. Campbell, and Michael T. Heath. ROCCOM: an object-oriented, data-centric software integration framework for multiphysics simulations. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 358–368. ACM Press, 2003.

[58] Amir Kamil, Jimmy Su, and Katherine Yelick. Making sequential consistency practical in titanium. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 15, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Amir Kamil and Katherine A. Yelick. Hierarchical pointer analysis for distributed programs. In *SAS*, pages 281–297, 2007.

[60] Ohannes Karakashian and Charalambos 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.

[61] T.L. Killeen. Energetics and dynamics of the Earth's thermosphere. *Rev. Geophys*, 25:433–454, 1987.

[62] Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Visualization of large datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4):24–33, July/August 2001.

[63] Tahsin Kurc, Umit Catalyurek, Xi Zhang, Joel Saltz, Malgorzata Peszynska, Ryan Martino, Mary Wheeler, Alan Sussman, Christian Hansen, Mrinal Sen, Roustam Seifoullaev, Paul Stoffa, Carlos Torres-Verdin, and Manish Parashar. A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience*, 17(11):1441–1467, 2005.

[64] Jay Larson, Robert Jacob, and Everest Ong. The Model Coupling Toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models. *Int. J. High-Perform. Comput. Appl.*, 19(3):277–292, 2005.

[65] Jay Walter Larson, Robert Jacob, Ian Foster, and Jing Guo. The Model Coupling Toolkit. In *Proceedings of International Conference on Computational Science*. Springer-Verlag, April 2001.

[66] Jae-Yong Lee and Alan Sussman. High performance communication between parallel programs. In *Proceedings of 2005 Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models (HIPS-HPGC 2005)*. IEEE Computer Society Press, April 2005.

[67] Sophia Lefantzi, Jaideep Ray, and Habib N. Najm. Using the Common Component Architecture to design high performance scientific simulation codes. In *Proceedings of the 2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*. IEEE Computer Society Press, 2003.

[68] Jon A. Linker, Roberto Lionello, Zoran Mikić, and T. Amari. Magnetohydrodynamic modeling of prominence formation within a helmet streamer. *J. Geophys Res.*, 106:25165–25176, 2001.

[69] Jon A. Linker and Zoran Mikić. Disruption of a helmet streamer by photospheric shear. *J. Atm. Terr. Phys.*, 438:L45–L48, 1995.

[70] Jon A. Linker, Zoran Mikić, Roberto Lionelloand, Pete Riley, T. Amari, and Dušan Odstrčill. Magnetohydrodynamic modeling of prominence formation within a helmet streamer. *J. Geophys Res.*, 106:25165–25176, 2003.

[71] Roberto Lionello, Zoran Mikić, and Jon A. Linker. Stability of algorithms for waves with large flows. *J. Comput. Phys.*, 152(1):346–358, 1999.

[72] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int'l Journal of Parallel Programming*, 32(3), 2004.

[73] Steven Lucco and Oliver Sharp. Delirium: an embedding coordination language. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Su-*

percomputing, pages 515–524, Washington, DC, USA, 1990. IEEE Computer Society Press.

[74] Janet G. Luhmann, Stanley C. Solomon, Jon A. Linker, John G. Lyon, Zoran Mikić, Dušan Odstrčil, Wenbin Wang, and Michael Wiltberger. Coupled model simulation of a Sun-to-Earth space weather event. *J. Atm. Terr. Phys.*, 66:1243–1256, 2004.

[75] J.G Lyon, J.A. Fedder, and C.M. Mobarry. The Lyon-Fedde-Mobarry (LFM) global MHD magnetospheric simulation code. *J. Atm. Terr. Phys.*, 66:1333–1350, 2004.

[76] Zoran Mikić and Jon A. Linker. Disruption of coronal magnetic field arcades. *Astrophysical Journal*, 430:898–912, 1994.

[77] Zoran Mikić, Jon A. Linker, Dalton D. Schnack, Roberto Lionello, and Alfonso Tarditi. Magnetohydrodynamic modeling of the global solar corona. *Physics of Plasmas*, 6:2217–2224, 1999.

[78] José E. Moreira, George Almási, Charles Archer, Ralph Bellofatto, Peter Bergner, José R. Brunheroto, Michael Brutman, José G. Castaños, Paul Crumley, Manish Gupta, Todd Inglett, Derek Lieber, David Limpert, Patrick McCarthy, Mark Megerian, Mark P. Mendell, Michael Mundy, Don Reed, Ramendra K. Sahoo, Alda Sanomiya, Richard Shok, Brian Smith, and Greg G. Stewart. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development*, 49(2-3):367–376, 2005.

[79] MPIG2. http://www3.niu.edu/mpi.

[80] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[81] Jarek Nieplocha, Andrés Márquez, John Feo, Daniel Chavarría-Miranda, George Chin, Chad Scherrer, and Nathaniel Beagley. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 47–58, 2007.

[82] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[83] Robert W. Numrich and John Reid. Co-Arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[84] Dušan Odstrčil, Murray Dryer, and Zdenka Smith. Propagation of an interplanetary shock along the heliospheric plasma sheet. *J. Geophys. Res.*, 101:19973–19984, 1996.

[85] Dušan Odstrčil and Victor J. Pizzo. Distortion of interplanetary magnetic field by three-dimensional propagation of cmes in a structured solar wind. *J. Geophys Res.*, 104:28225–28239, 1999.

[86] Dušan Odstrčil, Victor J. Pizzo, Jon A. Linker, Pete Riley, Roberto Lionello, and Zoran Mikić. Initial coupling of coronal and heliospheric numerical magnetohydrodynamic codes. *J. Atm. Terr. Phys.*, 66:1311–1320, 2004.

[87] Martin Ohmacht, Reinaldo A. Bergamaschi, Subhrajit Bhattacharya, Alan Gara, Mark Giampapa, Balaji Gopalsamy, Ruud A. Haring, Dirk Hoenicke, David J. Krolak, James A. Marcella, Ben J. Nathanson, Valentina Salapura, and Michael E.

Wazlowski. Blue Gene/L compute chip: Memory and Ethernet subsystem. *IBM Journal of Research and Development*, 49(2-3):255–264, 2005.

[88] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM Press.

[89] Rolf Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the Message Passing Interface Developer's and User's Conference*, pages 77–85, 1999.

[90] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[91] A. D. Richmond and Y. Kamide. Mapping electrodynamic features of the high-latitude ionosphere from localized observations: Technique. *J. Geophys Res.*, 93:5741–5759, 1988.

[92] R.M. Robinson, R.R. Vondrak, K. Miller, T. Dabbs, and D. Hardy. On calculating ionospheric conductances from the flux and energy of precipitating electrons. *J. Geophys Res.*, 92:2565–2569, 1987.

[93] R. G. Roble, E. C. Ridley, A. D. Richmond, and R. E. Dickinson. A coupled thermosphere/ionosphere general circulation model. *Geophys. Res. Lett.*, 15:1325–1328, 1988.

[94] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th*

ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 271–271, 2007.

[95] Vivek Sarkar. X10: An object oriented aproach to non-uniform cluster computing. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 393, 2005.

[96] Frank Schmuck and Roger Haskin. GPFS, a shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST 02)*. USENIX Press, January 2002.

[97] Spencer Shepler, Carl Beame, Brent Callaghan, Mike Eisler, David Noveck, David Robinson, and Robert Thurlow. Network File System (NFS) version 4 protocol, April 2003. RFC 3530.

[98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.

[99] Frank Suits, Michael Pitman, Jed Pitera, William C. Swope, and Robert S. Germain. Overview of molecular dynamics techniques and early scientific results from the Blue Gene project. *IBM Journal of Research and Development*, 49(2-3):475–488, 2005.

[100] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.

[101] Alan Sussman. Building complex coupled physical simulations on the grid with Intercomm. *Eng. with Comput.*, 22(3):311–323, 2006.

[102] Vasileios Symeonidis, George Em Karniadakis, and Bruce Caswell. A seamless approach to multiscale complex fluid simulation. *IEEE Comput. Sci. Eng.*, 7(3):39–46, 2005.

[103] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. The Panasas activescale storage cluster - delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004. IEEE Computer Society Press.

[104] TOP500 list - Nov 2007. http://top500.org/list/2007/11/100.

[105] Gábor Tóth and Dušan Odstrčil. Comparison of some flux corrected transport and total variation diminishing numerical schemes for hydrodynamic and magnetohydrodynamic problems. *J. Comput. Phys.*, 128(1):82–100, 1996.

[106] Charles D. Wait. IBM PowerPC 440 FPU with complex-arithmetic extensions. *IBM Journal of Research and Development*, 49(2-3):249–254, 2005.

[107] W. Wang, T.L. Killeen, A.G. Burns, and R.G. Roble. A high-resolution, three dimensional, time dependent, nested grid model of the coupled thermosphere-ionosphere. *J. Atm. Terr. Phys.*, 61:385–397, 1999.

[108] W. Wang, M. Wiltberger, A.G. Burns, S.C. Solomon, T.L. Killeen, N. Maruyama, and J.G. Lyon. Initial results from the coupled magnetosphere-ionosphere-thermosphere model: thermosphere-ionosphere responses. *J. Atm. Terr. Phys.*, 66:1425–1443, 2004.

[109] D. R. Weimer. Models of high-latitude electric potentials derived with a least error fit of spherical harmonic coefficients. *J. Geophys Res.*, 100:19595–19607, 1995.

[110] M. Wiltberger, W. Wang, A.G. Burns, S.C. Solomon, J.G. Lyon, and C.C. Goodrich. Initial results from the coupled magnetosphere.ionosphere.thermosphere model: magnetospheric and ionospheric responses. *J. Atm. Terr. Phys.*, 66:1411–1423, 2004.

[111] Joe Shang-Chieh Wu and Alan Sussman. Flexible control of data transfers between parallel programs. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 226–234. IEEE Computer Society Press, November 2004.

[112] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, 2007.

[113] Li Zhang and Manish Parashar. Enabling efficient and flexible coupling of parallel scientific applications. In *Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE Computer Society Press, 2006.