Runtime Coupling of Data-parallel Programs [1]

M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman and J. Saltz
Dept. of Computer Science and UMIACS
University of Maryland, College Park MD 20742
{ranga,acha,edjlali,als,saltz}@cs.umd.edu

Abstract

We consider the problem of efficiently coupling multiple data-parallel programs at runtime. We propose an approach that establishes a mapping between data structures in different data-parallel programs and implements a user specified consistency model. Mappings are established at runtime and new mappings between programs can be added and deleted while the programs are in execution. Mappings, or the identity of the processors involved, do not have to be known at compile-time or even link-time. Programs can be made to interact with different granularities of interaction without requiring any re-coding. A priori knowledge of data movement requirements allows for buffering of data and overlap of computations between coupled applications. Efficient data movement is achieved by pre-computing an optimized schedule. We describe our prototype implementation and evaluate its performance for a set of synthetic benchmarks that examine the variation of performance with coupling parameters. We demonstrate that the cost of the added flexibility gained by our coupling method is not prohibitive when compared with a monolithic code that does the same computation.

1 Introduction

In the sequential programming world, inter-application data transfer facilities abound. Applications can use simple abstractions such as sockets, pipes or shared memory segments to move data between address spaces. There are no restrictions on the programming language used to develop the communicating applications. This provides flexibility and reconfigurability for sequential applications. Similar facilities are not available for data parallel programs. The obvious technique of using a shared file system is not efficient.

In this paper, we propose an approach that achieves direct application to application data transfer. Our approach is library-based and is independent of the programming language used to develop the communicating applications. Programs written to use this approach are required to adhere to a certain discipline with respect to the data structures involved in the interaction, but they do not need to know either the identity or the number of programs they interact with.

Our approach is built around the notion of *mappings* between data structures in different data-parallel programs. Mappings are established at runtime. Every mapping has a consistency specification which mandates the logical frequency with which the mapped structures are to be made mutually consistent. Mappings, or the identity of the processors involved, do not have to be known at compile-time or even link-time. A priori knowledge of the mapping specification at run-time allows for overlapping execution of the interacting programs by buffering the data. Efficient data movement is achieved by pre-computing an optimized plan (schedule) for data movement. Our prototype implementation uses a generalized data movement library called Meta-Chaos [5] and is able to couple data-parallel programs written in different languages (including High Performance Fortran (HPF) [4], C and pC++ [1]) and using different communication libraries (including Multiblock PARTI [19] and CHAOS [12]).

By coupling multiple concurrently executing data parallel applications, we gain the added benefit of combining task and data parallelism. In contrast to other approaches that require language extensions to achieve this [7, 17, 18], our approach can work with off the shelf compiler implementations as long as the implementations provide certain query functions about the distributions of data structures [5].

[1] This research was supported by NASA under grant NASA #NAG-1-1485 (ARPA Project Number 8874), by ARPA under grant #F19628-94-C-0057 and by NSF under grant #ASC9318183. The Digital AlphaServer used for experiments was provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and a grant from Digital Equipment Corporation.

We have developed a prototype implementation based on this approach. Our implementation currently runs on a cluster of four-processor Digital Alpha Server 4/2100 symmetric multiprocessors. Our results indicate that data-parallel programs can be coupled together in a flexible fashion with acceptable overhead.

2 Basic Concepts

Central to our approach is the notion of *mappings* between individual data structures belonging to the programs being coupled. A mapping binds a pair of data structures of equal size and identical shape, and has an associated consistency specification that specifies the *frequency* with which the mapped data structures are to be made mutually consistent. Consider the example of a pair of simulations which work on neighboring grids and periodically exchange data at the boundary. In this case, the array sections in both programs that correspond to the shared boundary would be mapped to each other. The consistency specification would depend on the requirements of the physical process being simulated and the accuracy desired; the strongest consistency requirement would be *exchange data every time-step* and the weakest *never exchange data*. For a different kind of interaction, consider the coupling of a program that simulates a physical process and a visualization program that displays its state. In this case, the mapping would be between the array containing the state and the array used to hold the data points for visualization. The consistency would depend on the closeness of monitoring desired - for instance, *display every time-step*, or *display as many frames as possible without slowing down the simulation*.

The *frequency* referred to above is logical. It refers to the number of times execution in either program crosses specific user (or compiler) identified synchronization points. In the example of interacting simulations, the synchronization points could be the bottom of the respective time-step loops; in the coupling between a simulation and a visualization program, the synchronization points could be the bottom of the time-step loop in the simulation program and the end of the frame buffer update in the visualization program.

In the implementation we have developed, mappings are established at runtime and are maintained by a runtime library. New mappings may be added between programs in execution and existing mappings may be deleted. For example, such a dynamic mapping addition feature may be useful for coupling a visualization program to a long running simulation as it progresses.

Our approach derives its efficiency from buffering and asynchronous transfer of data, as well as precomputation of optimized schedules. A schedule consists of a plan for moving the data from the sending processors to the receiving processors. Schedules are optimized to minimize the number of messages transmitted.

While our approach is general enough for a variety of data structures, in this paper we restrict ourselves to arrays and array sections. We do this for two reasons. First, at this stage in our research, we would like to focus on maximizing flexibility and reconfigurability rather than on specification of complex data structures. Restricting our focus to arrays allows us to use simple existing techniques to describe the data structures of interest. Second, the primary data structures in most data-parallel programs in use today are arrays. Therefore the restriction does not significantly limit the applicability of our approach.

3 The Programming Model

The programming model provides two primary operations: *exporting* individual arrays and establishing a *mapping* between a pair of exported arrays. Arrays are exported by application writers, who use a set of primitives to identify exported arrays and to specify the points in the application program at which consistency operations can be safely applied. Mappings between exported arrays are established by users who wish to couple the corresponding applications.

3.1 Exporting arrays

Four primitives are provided for exporting arrays : `register()` and `unregister()` to control the visibility of the array outside the application and `acquire()` and `release()` to specify the points in the application code at which consistency operations can be safely applied.

The following are the primitives in our model :

- `register(`$array, mode, name$`)`: binds *array* to the system-wide unique identifier *name* and makes it "visible" to other applications. *array* is a "distributed array descriptor". It describes the distribution of the distributed array among the processors of the calling program. There are two possible values for *mode*, `in` and `out`. Data can only be transferred *into* arrays that have been marked `in`. Similarly, data can only be transferred *out* of arrays which have been marked `out`. `register` returns a *handle* that can be used to refer to the exported array in subsequent code.

- `unregister(`$handle$`)`: permanently hides the (previously exported) array associated with *handle*.

- `acquire(`$num\_handles, set\_of\_handles$`)`: All consistency operations involving an array for which the `acquire` call has been issued must be completed before the `acquire` call returns. For an array exported in the *in* mode, all transfers into the array that are required for maintaining the desired consistency must complete before the `acquire` returns. For an array exported in the *out* mode, all transfers out of the array that are required for maintaining the desired consistency must be complete before `acquire` returns.

- `release(`$num\_handles, set\_of\_handles$`)`: For an array that has been exported in the *out* mode, `release` indicates that a new version of the array is now in place and will remain in place until the next `acquire` on it. For an array that has been exported in the *in* mode, `release` indicates that it is now safe to change the data in the array.

The `acquire()` and `release()` calls must be placed in the data parallel program such that each of the processes in the data-parallel program sees the same number of `acquire`s and `release`s at a given logical point in the program execution (implying a loosely synchronous SPMD execution model).

3.2 Establishing mappings between data structures

A mapping consists of two parts - the names of the arrays (or array sections) being mapped and a specification of the desired consistency. The general form of a mapping is:

$$\texttt{with}\ consistency\_specification\ \{arraysection_1 = arraysection_2\}$$

Arrays are referred to by their external names and can be multi-dimensional. As was mentioned in the previous section, external names are bound to arrays using the `register` primitive. Array sections are specified using an HPF-like syntax (i.e., $array[init : final : stride]$). For instance, `x[1:100:2]` specifies a section of the one-dimensional array `x` consisting of every second point in the range 1 to 100. There can be many active mappings connecting different arrays in different programs.

A consistency specification mandates the frequency with which the array sections are to be made mutually consistent. The frequency is specified logically, in terms of a version counter. Operationally, a zero-initialized counter is associated with every exported array and is incremented on every `release` of the array. For an array exported in the *out* mode, the counter contains the number of versions of that array that have been made available to other applications. For an array exported in the *in* mode, the counter contains the current number of safe opportunities for data to be placed into the array.

A consistency specification consists of a pair of conditions, one for each array in the mapping. The mapped data structures must be consistent whenever (and as long as) both conditions hold. The general form of a consistency condition is

$$\texttt{freq}(array, init : final : stride)$$

.

The value of *init* can be a non-negative integer or the special symbol `current`, with or without a positive integral offset. The symbol `current` stands for the value of the version counter for the given array at the time the mapping is established. The domain of $final$ is the set of natural numbers and a special symbol `forever` (which is denoted in this paper by ∞). If *init* is specified as `current` , $final$ may be specified as `current` plus an integer value. The value of *stride* can be a natural number or the wild-card symbol `*`. The wild-card symbol stands for *any* natural number. The expression *init:final:stride* defines a (possibly infinite) sequence of non-negative integers[2]. A consistency condition holds whenever (and as long as) the value of the counter associated with *array* belongs to the sequence defined by *init:final:stride*.

We use the following terminology for the rest of the paper : The data parallel program where a given exported array is defined is called the *owner* of the array. The owners of the exported arrays that appear in a mapping are the *participants* in the mapping. The owner of the Left Hand Side (LHS) of the assignment appearing in the mapping is called the *consumer* and the owner of the Right Hand Side (RHS) of the assignment is called the *producer* for that mapping. The array (or array section) that appears on the RHS of a mapping is called the source array for the mapping and the one that appears on the LHS of a mapping is called the sink array for the mapping. We refer to the processes that constitute a data-parallel program in execution as data-parallel peer processes.

Mappings can be specified in two ways. For static couplings, in which all participants start executing at the same time and the interactions between the applications do not change throughout the execution, the mappings can be specified in a configuration database that can be read by all applications as a part of their initialization. For dynamic couplings, in which some participants may start executing after others or the interactions between the participants change during execution, the mappings can be created or deleted as the participants are in execution. The following primitives are provided for this :

- `is_exported(`*name*, *mode*`)`: returns `true` if the external name *name* has been bound to an array exported in mode *mode*; returns `false` otherwise.
- `create_mapping(`*mapping*`)`: parses the mapping, which is specified as a string, and sets it up, returning a handle to the caller. Neither of the arrays being mapped need be exported by the application that calls `create_mapping`. That is, a mapping can be established by a program that does not participate in the mapping. This allows couplings to be established and/or controlled by other programs. A mapping can be successfully established if and only if all the following conditions hold:

  1. Both arrays are currently exported and the two arrays (or array sections) being mapped are of the same size and shape with identical base data types,
  2. The source array is exported in *out* mode; the sink array is exported in the *in* mode.
  3. The consistency specification associated with the mapping is *satisfiable*. A consistency specification is not satisfiable if the value of the *init* parameter for either array is a non-negative integer k and version k of that array is not available currently and will never become available in the future.

  If the mapping is successfully established, a handle is returned to the caller ; else an error is returned.
- `delete_mapping(`*handle*`)`: dismantles the mapping corresponding to *handle*.

3.3 A simple example

In this section, we illustrate the use of our system with a simple example. Consider the two programs in Figure 1. Data parallel program pgm_1 `registers` its distributed array A giving it the global name **A** and giving it "in" permission. Data parallel program pgm_2 `registers` its distributed array B giving it the global name **B** and giving it "out" permission. Independently of pgm_1 and pgm_2, indices 1 to 10 of **A** may be associated with indices 10 to 20 of **B** with a mapping as shown in Figure 1.

This mapping couples one section of the one-dimensional array **A** to another section of the one-dimensional array **B** and specifies that for every time-step in the range 0 to 100, elements 1 to 10 of **A** must contain the same values as elements 10 to 20 of **B** after the `acquire` completes in pgm_1.

In this example, the meanings of the `acquire` and `release` calls in pgm_2 can be explained as follows:

- Before the first $acquire(b)$ - the mapping says that the producer will transmit values of B for every version of B starting at version 0. Thus, before the first $acquire(b)$ completes, the zeroth version of B is transmitted to the consumer pgm_1.

[2] We will discuss the different kinds of sequences possible and their associated semantics in Section 3.4.

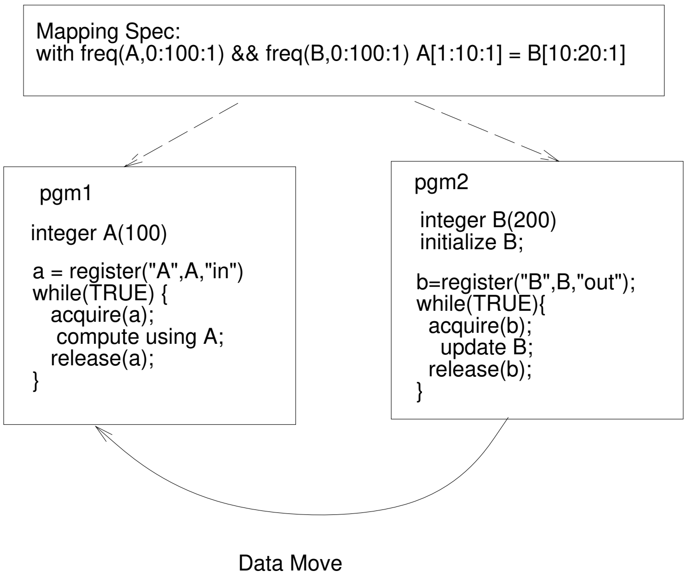

Figure 1: Pprogram text and mappings for example.

- For any version t of array B, such that $0 < t \leq 100$, the transfer from B corresponding to version $(t-1)$ must complete before $acquire(b)$ returns.
- In between $acquire(b)$ and $release(b)$, no transfers from B can occur.
- When $release(b)$ is executed, the version counter corresponding to B is incremented at the producer. Data transfer to consumers may commence at this point. The transfer of data out of B needs to complete before the next $acquire(b)$ completes.

Similarly, consider the state of variable A in pgm_1 for the example above :

- Before the first $acquire(a)$ completes, there is no defined relationship between A and B.
- After the first $acquire(a)$ completes, values at locations `[10:20]` of version 0 of B have been copied into locations `[1:10]` of version 0 of A.
- For any version t of Array A (such that $0 < t \leq 100$), after $acquire(a)$ completes, values at locations `[10:20]` of version t of B have been copied into locations `[1:10]` of version t of A.
- In between $acquire(a)$ and $release(a)$ no transfers into array A may occur.
- When $release(a)$ is executed, the version counter for A is incremented and pending updates for A may begin for the new version counter.

As the programs are in execution, a new mapping may be added to the program with the following invocation :

```
create_mapping( "with freq(A,current:current+100:1) && freq(B,current:current+100,1)
                A[20:30:1] = B[40:50:1]");
```

This adds a new dynamic mapping between A and B which will remain active for 100 increments of the version counters for A and B after the time the mapping is added. After the dynamic mapping is added, sections `[20:30]` of A are kept consistent with `[40:50]` of B for 100 iterations after the mapping goes into effect. Existing mappings are not changed by the addition of the dynamic mapping.

3.4 Informal semantics

There are two major classes of consistency conditions - *strided* and *wild-card*. Strided conditions can take one of two forms: $\texttt{freq}(array, const_1 : const_2 : const_3)$ or $\texttt{freq}(array, const_1 : \infty : const_3)$. Strided conditions are useful for specifying periodic interactions between coupled programs, e.g. a pair of interacting simulations that communicate after a fixed number of time-steps. Wild-card conditions can also take one of two forms : $\texttt{freq}(array, const_1 : const_2 : *)$ or $\texttt{freq}(array, const_1 : \infty : *)$. Such conditions capture the consistency requirement for loosely coupled programs - for example a coupling between a simulation and a visualization program that displays as many frames as possible without slowing down the simulation and forcing it to run at the same speed as the visualizer. In the rest of the paper, we shall use the general forms of both these classes, that is, $\texttt{freq}(array, const_1 : \infty : const_3)$ for strided requests and $\texttt{freq}(array, const_1 : \infty : *)$ for wild-card requests.

The primary synchronization primitives in the model are `acquire` and `release`. They are used as synchronization points for the user specified consistency operations. The following consistency guarantees are provided :

1. **Safe transfer guarantee:** No data is transferred from or to an array between a matching `acquire/release` pair involving that array. Data can be transferred from or to an array any time between a `register` call and the first `acquire` or between a `release` and the next `acquire` (or `unregister`).

2. **Single version guarantee:** all data transferred to or from a single array in a single consistency action belongs to the same version. Note that this requirement does not necessarily imply explicit barrier synchronizations at every `acquire` and `release`.

There are four classes of consistency specifications, each corresponding to a different consistency model. They are *fully-constrained*, *producer-constrained*, *consumer-constrained* and *free-running*. In the following discussion, k is a non-negative integer. $const_t$ symbols are used to denote integer constants ≥ 0. For simplicity, we assume the upper limit of the consistency condition specification is unbounded (∞). For the rest of this section, we assume a mapping of the form $A = B$ where program P_1 exports A and program P_2 exports B.

A consistency specification for a *fully-constrained* coupling is of the form

$$\texttt{with freq}(A, const_1 : \infty : const_2) \ \&\& \ \texttt{freq}(B, const_3 : \infty : const_4) \ A \ = \ B$$

In this model, every $const_4^{th}$ version of B is copied into A on every $const_2^{th}$ `acquire` call involving A. More precisely the data contained in B at the $(const_3 + k \times const_4)^{th}$ $\texttt{release}(B)$ call must be transferred to A. The data must be transferred out of B between the start of the $(const_3 + k \times const_4)^{th}$ call to $\texttt{release}(B)$ and the completion of the following call to $\texttt{acquire}(B)$ [3]. This data must be transferred into A after the $(const_1 + (k \times const_2) - 1)^{th}$ $\texttt{release}(A)$ call has completed and before the following call to $\texttt{acquire}(A)$ completes [4]. The *fully-constrained* model is able to capture a wide range of consistency requirements for relatively closely coupled programs.

A consistency specification for a *producer-constrained* coupling is of the form

$$\texttt{with freq}(A, const_1 : \infty : *) \ \&\& \ \texttt{freq}(B, const_3 : \infty : const_4) \ A \ = \ B.$$

In this model, every $const_3^{th}$ version of B is copied over to A. No data is transferred to A for the first $const_1$ calls to $\texttt{acquire}(A)$. The data must be transferred out of B between the start of the $(const_3 + k \times const_4)^{th}$ call to $\texttt{release}(B)$ and the completion of the following call to $\texttt{acquire}(B)$. This data must be transferred into A at a subsequent call to $\texttt{acquire}(A)$ after the first $const_1$ calls to $\texttt{release}(A)$. The *producer-constrained* model constrains only the producer and allows the consumer to run freely. It can be used to couple programs in which the producer runs much faster than the consumer and periodic consistency with a known period is not needed.

A consistency specification for a *consumer-constrained* coupling is of the form

$$\texttt{with freq}(A, const_1 : \infty : const_2) \ \&\& \ \texttt{freq}(B, const_3 : \infty : *) \ A \ = \ B$$

In this model, no data is transferred into A for the first $const_1$ calls to $\texttt{acquire}(A)$. Subsequently, data

[3] If $const_3 = 0$, the first data transfer out of B must happen between the $\texttt{register}(B)$ call and the first call to $\texttt{acquire}(B)$.
[4] If $const_1 = 0$, the first data transfer into A must happen between the $\texttt{register}(A)$ call and the first call to $\texttt{acquire}(A)$.

Coupling Type	Coupling Specification	Versions seen by consumer
fully-constrained	freq(A,0,∞,1) && freq(B,0,∞,1)	P_1 sees each version of B in increasing order
consumer-constrained	freq(A,0,∞,1) && freq(B,0,∞,*)	P_1 sees increasing versions of B
producer-constrained	freq(A,0,∞,*) && freq(B,0,∞,1)	P_1 sees non-decreasing versions of B. P_2 sends every version of B
free-running	freq(A,0∞,*) && freq(B,0,∞,*)	P_1 sees non-decreasing versions of B. If B is updated P_1, is guaranteed to see a new version of B.

Table 1: Versions seen by consumer (P_1) for different consistency specifications

must be transferred into A once every $const_2$ calls to `acquire`(A). There is no restriction on the version of B that can be copied over at each such transfer point, as long as the sequence of versions is monotonically increasing starting at the $const_3^{th}$ version. That is, every transfer gets a new version of B. With this proviso, data can be transferred out of B between any call to `release`(B) after the $const_3^{th}$ call and the the following `acquire`(A). The *consumer-constrained* model constrains only the consumer and allows the producer to run freely. It can be used to couple programs in which the consumer runs much faster than the producer and periodic consistency is not needed.

A consistency specification for a *free-running* coupling is of the form

$$\texttt{with freq}(A, const_1 : \infty : *) \ \&\& \ \texttt{freq}(B, const_3 : \infty : *) \ A \ = \ B$$

This model provides the loosest coupling. In this model, there are four restrictions on data transfer. First, no data transfer takes place for the first $const_1$ calls to `acquire`(A) and the first $const_3$ calls to `release`(B). Second, at least one data transfer takes place. Third, monotonically increasing versions of B are transferred. That is, every transfer gets a new version of B. However, there may an arbitrary number of `acquires` of A and `releases` of B between any data transfers. Finally, if B's version number has been changed since the last `acquire` of A, a consistent new version of B will be propagated to A. The *free-running* model constrains neither the producer nor the consumer. The consumer observes a trend of the producer's values as the producer progresses.

The consistency specifications and version numbers that the consumer sees for different consistency specifications are summarized in Table 1. For simplicity, we have chosen a unit stride in the consistency specification.

Since we allow a program to have multiple sources and multiple sinks we can form rather general interconnections between programs. However, this flexibility can also lead to deadlocks and infinite buffering requirements for certain configurations. These are not always detectable by looking at the mappings alone. In our implementation, we expect the programmer to be aware of these situations when building an interconnection of applications. We address these issues in greater detail in a technical report [14].

4 Implementation

We have implemented our system on a network of four-processor SMP Digital Alpha Server 4/2100 workstations. The nodes are connected by an FDDI network [5].

The primary goals of our implementation were language independence, flexibility and efficiency. The concern for language independence prompted the use of the Meta-Chaos library [5], which we introduce below. There are several points to be made about our implementation. First, we used asynchronous, one

[5] We are in the process of upgrading to an ATM switch providing 155 Megabits/sec links between processors on which we intend to re-evaluate our implementation.

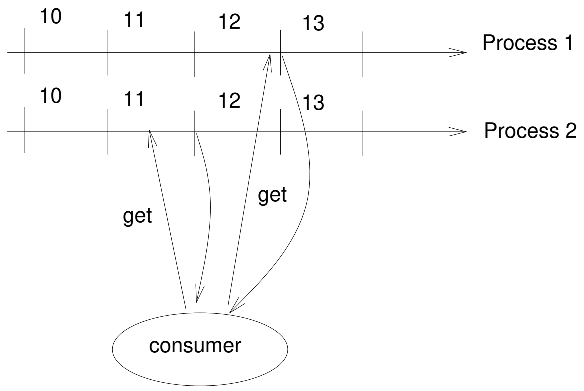

Figure 2: The problem with consumer initiated transfers for consumer constrained consistency.

sided message-passing for inter-application data transfer with the goal being to overlap data transfer with computation. Second, we computed optimized messaging schedules for data transfer for each mapping and reused these schedules for all transfers for the given mapping. The goal here is to minimize the number of messages transmitted thereby reducing the amount of time spent in communication. Third, we used buffering to reduce idle time spent waiting for data. We now present further details as well as some problems we encountered.

4.1 Implementation of Mappings

Data transfer can be initiated by either the producer or the consumer data-parallel program in a mapping. A consumer initiated transfer is implemented by a "get" request to the producer, which is processed at an appropriate time in the producer's execution. A producer initiated transfer is implemented by the producer dispatching the necessary data in a "put" request. The data may be received asynchronously at the consumer and buffered for later consumption.

The initiation scheme is specific to each mapping and depends only on the consistency model it implements. For mappings implementing the *fully-constrained* model or the *producer-constrained* model, data transfer is initiated by the producer. This eliminates the need for a consumer initiated request message. Since the relative time when the data is to be supplied is known a-priori in these cases, a consumer initiated request is unnecessary. For mappings implementing the *consumer-constrained* model or the *free-running* model, the data transfer is initiated by the consumer. In the first two cases, the producer initiates the data transfer at the end of the `release` call that generates the version to be transferred. In the last two cases, the consumer initiates the data transfer at the beginning of the appropriate `acquire` call.

If the transfer is producer initiated, ensuring the *Single Version guarantee* (i.e., the guarantee that the consumer sees a single consistent version of the distributed data) is simple. The peer processes of the data-parallel producer application may send their sections of the distributed array to the consumer on a `release`. Since the data is buffered and consumed in FIFO order at the consumer, and the loosely synchronous SPMD assumption holds for the producer, the *Single Version Guarantee* is ensured.

If the transfer is consumer initiated, the problem is more complicated. This complication is caused by the fact that different peers of the data-parallel program can see the same request at different logical points in their computation, as is illustrated in Figure 2. If the peers respond as soon as they see the "get" request, the consumer may see different portions of the distributed array with different version numbers thereby violating the *Single Version Guarantee*. Some coordination between the producer peers is required to ensure that this situation does not happen. A simple distributed protocol that guarantees that the consumer sees a consistent version of the source array has been implemented and is described in greater detail in [14].

Next we consider dynamic addition of mappings. A mapping is safe for the producer iff all the versions of the array it would transfer are available (either buffered or yet to be generated). Similarly, a mapping is safe for the consumer iff all the consistency operations specified by the mapping can be performed, meaning that execution has not passed the point at which the mapping would require a data transfer into an array.

A mapping may be added if and only if both participants in the mapping agree that the mapping is safe. This is determined by shipping the mapping to both participants and getting their agreement. A distributed agreement algorithm is implemented to add a mapping dynamically. The details are omitted here for brevity.

4.2 Data Transfer

For inter-application data transfer, we use a library called Meta-Chaos. Meta-Chaos is able to manage data movement between data-parallel programs written in different languages (including HPF, C and pC++) and using different communication libraries (including Multiblock PARTI and CHAOS). Meta-Chaos operates by transforming the different distributed arrays into the same canonical representation and building a schedule for data movement between the two arrays. Depending on the structure of the distributed data the canonical representation can be compact (e.g. block distributed arrays), but in the case of irregular distributions could be as large as the array. These canonical representations are mapped to each other, and a plan for data movement between processors is computed based on this mapping. This plan is optimized to minimize the number of messages between processors. Once the plan is computed, it is cached and re-used for later inter-application data movements.

To port our mappings library to work with different data parallel languages and runtime libraries, the Meta-Chaos library must be portable. For this to be possible, the Meta-Chaos library relies on several inquiry functions that must be made available by the runtime libraries or languages that are being coupled. These inquiry functions include queries on index ownership and location. In addition, functions need to be provided to convert between global and local indices.

The Meta-Chaos library, including descriptions of the necessary support functions that it requires from the data-parallel run-time library being linked, is described in greater detail in [5]. We have successfully implemented Meta-Chaos to test inter-operability between data-parallel programs written using HPF, CHAOS, Multiblock Parti and pC++.

For the underlying messaging layer between applications, we used PVM [8]. Each data parallel program is assigned a distinct PVM group. Asynchronous data transfer is achieved by using a dedicated thread for receiving messages. Since, PVM currently does not handle multiple threads concurrently performing `pvm_receive` operations in the same process correctly, we assume that intra-program communication between the peers of the data-parallel program will be done through some other means. This has not been an operational problem for our experiments, since the Digital HPF compiler uses a proprietary version of the UDP protocol for communication between the peers of an HPF program.

5 Evaluation

We first evaluated the overhead of our flexible coupling system when compared with hand-coded message passing. Next, we examined the performance of our system using mini-applications. Our experimental scenarios were set up to examine the following aspects of our system :

- The variation of producer and consumer performance with coupling granularity.
- The performance of consumer-constrained coupling for data-parallel applications.
- The performance of tightly constrained coupling applied to a class of computation that occurs frequently in scientific codes.

Our experimental platform has already been described in the previous section. The application to application data transfer rate between two C applications on the network using connection oriented sockets and transferring 40 KBytes of data per send averaged 24.4 Megabits/sec. Inter application data transfer between nodes using PVM and transferring 40 KBytes per send, was measured at 23.5 Megabits/sec on average. The rated maximum transfer rate of the network is 100 Megabits/sec. We expect to repeat the experiments in this section in the very near future, as soon as the ATM switch connecting the nodes is installed.

Processors(Sender/Receiver)	Direct Message Passing	Mappings based coupling
1	14.5	14.7
2	13.6	14.0
4	12.8	12.9
8	15.2	15.6
16	36.8	36.9

Table 2: Comparison of transfer time for Mappings based coupling of 100x100 integer array and direct message passing between applications (ms per send averaged over 1000 iterations).

5.1 Overhead

The raw performance of the system is evaluated by comparing a mappings based linkage to a direct coupling using `send` and `receive` calls with the schedules generated by Meta-Chaos. Once the schedules for data movement have been built, the performance of Meta-Chaos is identical to what can be achieved by direct message passing. Having a separate thread for handling data transfer requests adds very little overhead when compared with hand coded sends and receives for tight coupling. The overhead of asynchronous transfers is caused by thread switching and locking, but is offset to some extent by the fact that threads are kernel scheduled in the system and receives may be posted asynchronously by the dedicated thread that receives messages. The *acquire* and *release* calls add negligible overhead to a program when no mapping exists for the array being `acquire`d or `release`d.

Table 2 shows the overhead of using mappings to couple programs in comparison to hand coded `sends` and `receives` for a 100x100 integer array being transferred between two data parallel programs. The timings are averaged over 1000 iterations of the interacting programs.

In the experimental setup, the producer and consumer each run on disjoint sets of 4 workstations each. The distribution of the processes constituting the data parallel applications over nodes in this experiment was done in a round robin fashion over the nodes. Identical distributions were chosen for both the hand coded send/receive case and the mappings based coupling.

The time to transfer the array decreases because of improved aggregate bandwidth as the number of communicating nodes is increased. However, as we start to multi-task the nodes, the effect of contention for the network adapter becomes apparent and the throughput is seen to drop. The main point of this experiment is that coupling programs using our mappings approach does not add significant overhead to direct application to application message passing.

5.2 Performance impact of coupling granularity

The first of our mini-application experiments demonstrates the effect of changing the coupling strength between applications. For this experiment, the producer is a process that runs an infinite loop incrementing each of the elements in a 100x100 exported integer array on each iteration. The consumer is a summing process that sums all the elements of its array on each iteration. Both producer and consumer are sequential applications that run on a separate node. Table 3 shows the variation of the average loop time for the producer and the average wait time for the consumer as the strength of coupling is varied.

Table 3 shows how weakening the coupling affects the performance of the producer and the consumer. In the *fully-constrained* case, a difference between the wait time and the producer loop time is seen due to the buffering effect at the consumer. In the *consumer-constrained* case several producer loop iterations are allowed to run before a single consumer *acquire* is required to complete (the stipulation is that a new version should be supplied on each acquire but there is no stipulation on what the version is). In the *producer-constrained* case, consecutive acquires of A could get the same version - the stipulation here is that every version of B is seen by some loop iteration of Pgm_1. Thus in this case the producer runs approximately at the same rate as the fully constrained case. Finally, in the *free-running* case, the coupling strength is the weakest and the performance is the best. The only guarantee here is that the consumer will observe a

Coupling Type	Coupling Spec	Producer Loop Time	Consumer wait time
fully-constrained	freq(A,0:∞:1) && freq(B,0:∞:1)	14.7	8.7
consumer-constrained	freq(A,0:∞:1) && freq(B,0:∞:*)	1.5	190
producer-constrained	freq(A,0:∞:*) && freq(B,0:∞:1)	14.0	.41
free-running	freq(A,0:∞:*) && freq(B,0:∞:*)	1.5	.11

Table 3: Producer Loop time and consumer wait time for different consistency specifications (ms).

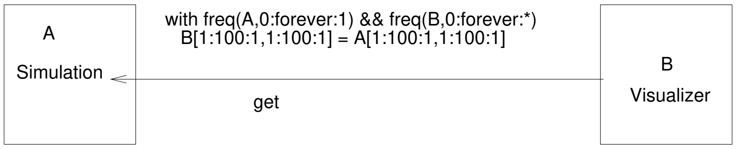

Figure 3: Coupling a visualization "frame grabber" to a simulation

"trend" of the producers values. For every `acquire` of A the consumer sees the same or a later version of B, as compared to the previous acquire.

5.3 Performance of consumer constrained coupling

This experiment is a demonstration of the utility of consumer-constrained coupling for a data-parallel program.

Consider a simulation producing output to be fed into a visualizer. Assume that A is the exported array owned by the simulation program and B is the exported array owned by the visualizer. The relative speed of the producer and the consumer is not necessarily known in this case. The user wants to "sample" the progress of the simulation interactively. Each array is is a 100 x 100 integer array. The scenario is shown in Figure 3.

We implemented skeleton HPF applications to test the performance of this coupling. To measure the performance of the system, we ran the visualizer in a tight loop doing only *acquire* and *release* and measured the average wait time for the `acquire` operation. Table 4 shows the result. The average wait time is an indication of the maximum rate at which the visualizer may grab frames from the simulation.

In this experiment, the processors for the simulation were allocated in a "greedy" fashion. All processors on a given node are assigned before another node is added. The visualizer runs on a separate node.

The relatively high average wait time for the consumer is a result of the fact that the rule is marked *consumer-initiated*. Before the `acquire` completes, a `get` request has to be dispatched to the producer and the producer nodes have to compute a version number corresponding to a consistent distributed snapshot of the source array before responding. The larger the number of peers in the consumer, the greater is the time spent in this operation. Also, with more peers, the possibility of skew between the version numbers is higher, thereby resulting in a higher average wait before the data can be dispatched to the consumer. The jump in

Processors	Avg Consumer wait time	Avg Producer Loop time
1	190	16.3
2	196	8.47
4	211	4.54
8	310	2.86
16	392	2.41

Table 4: Timings for visualization example (ms).

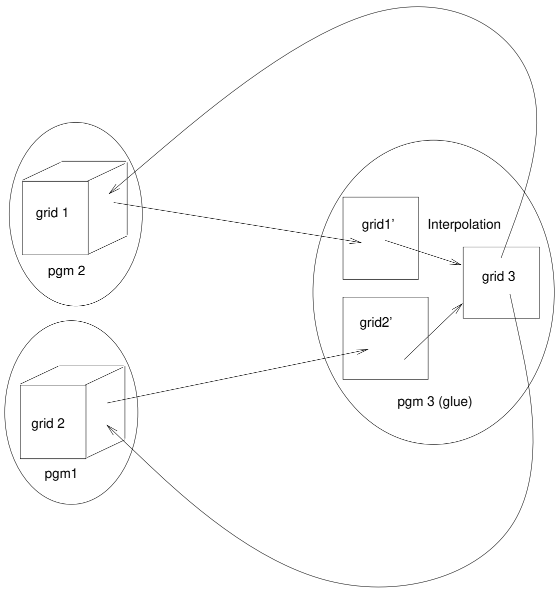

Figure 4: Coupling simulations.

wait time between 4 and 8 processors is due to network traffic as a result of inter-node communication for processing the *get* request. If all the peers are on the same node, communication is much cheaper than if the peers have to use the network. The table also shows the producer loop time for this experiment. We measured the loop time for the producer in the case when the producer was un-coupled and compared it with the loop time when the producer was coupled. The difference in performance between a coupled producer and an un-coupled producer was not significant.

5.4 Fully Constrained Coupling

In this experiment we consider a tight coupling between two simulations written in HPF. Consider linking two simulations that do computations by sweeping over a 3-dimensional grid doing local operations (nearest neighbor stencil computations) at each grid point. The loop doing the sweep is parallelized using an HPF *forall* statement. Such a sweep is representative of the computations in a large class of scientific applications, such as computational fluid dynamics codes and structural mechanics codes. The grids are connected at their boundaries.

Each of the coupled grids is 100x100x100 integers. We compared the performance of three cases :

- A monolithic HPF program that sweeps over a grid that is 200x100x100 integers doing a nearest neighbor summation at each point with a single forall loop.
- Coupling two grids with data exchange between the two grids at the boundary. This does the same computation as the monolithic grid but breaks the computation onto two grids. The computations exchange data (fill ghost cells) at the boundaries where the grids meet. The performance for this scenario is shown in Figure 5 labelled "Coupled HPF simulations 1".
- Coupling two grids by passing the data through a third "interpolation" program. The scenario is shown in Figure 4. In this figure pgm_1 and pgm_2 are computing simulations on their grids. pgm_3 is computing an interpolation on the connected faces of grid1 and grid2, which is then read back by pgm_1 and pgm_2 on each iteration. The performance for this scenario is shown in Figure 5 labelled "Coupled HPF simulations 2".

]

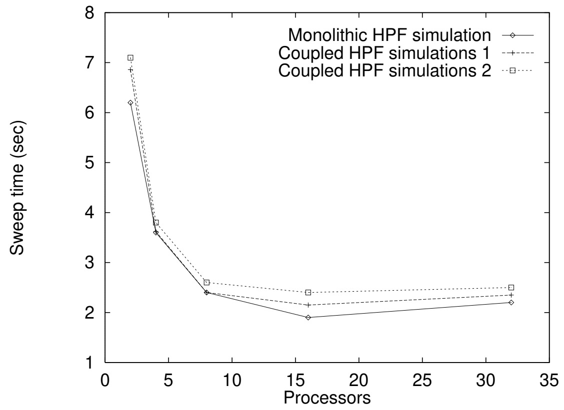

Figure 5: Performance of coupled simulation vs. monolithic coupling

Each simulation in our coupled simulation scenario runs on a different set of nodes. In the third case, where there is a separate interpolation program between the simulations, the interpolation program runs on the same set of nodes as one of the simulations in a multi-tasked fashion. In all cases, since our network has a relatively high latency and the computation for each grid point is relatively inexpensive, there is not much improvement in the results past 8 processors. In fact, there is a slight degradation for 32 processors when compared with 16 processors.

The communication and the computation for both the HPF monolithic simulation and the coupled simulation in the second case (without the interpolation program) is identical. The degradation in performance for the coupled simulations can be explained by the fact that we are using PVM to couple applications. In the third case (with the interpolation program) there is additional communication, but the additional communication is between processes that run on the same node. This is true because the interpolation program runs on the same nodes as one of the simulation programs in a multi-tasked fashion. The interpolation program itself is very simple - just averaging the data values on grid1' and grid2' and writing the result out to grid3. This generates no communication. Since intra-node communication is relatively cheap the interpolation program does not add appreciable overhead to the coupling.

The experiments indicate that performance is not degraded much by coupling data-parallel applications in a fully constrained manner.

6 Related Work

Our approach is similar in some respects to the software bus approach used in Polylith [16]. A software bus isolates and encapsulates run time interfacing concerns for applications. The software bus allows modules to export interfaces and be invoked by remote processes. Our approach is data stream driven rather than remote procedure call driven. Data parallel components can interact not only at their entry and exit points but also concurrently when they are in execution. However, we do not provide a means for remotely invoking procedures. Indeed, a software bus approach could complement our work extending it to allow this facility. The other alternative would be to use PVM to start tasks remotely, and then use our approach to connect the data streams of the executing tasks. Polylith has also been extended to allow for dynamic re-configuration while the system is executing [11].

Single address space operating systems such as Opal [3] offer a uniform view of memory across all processes. All processes in Opal share the same address space and hence this facility could be used as a means of sharing objects between applications.

Linda [15] offers a tuple space oriented programming model which could be used to couple programs. A stream oriented model such as ours could be implemented on top of Linda. Given that our assumption is that the source codes for the individual applications is not available at the time the applications are to be composed, the performance would probably not be as good as our implementation. External control and the size of the data sets would probably present performance problems.

The general topic of integrating task and data parallelism has received a lot of attention. Some research efforts are working on enhancements of data parallel languages or task parallel languages in order to integrate data parallelism and task parallelism. Fx [18] uses additional HPF directives to specify task parallelism. Opus [10] is a set of HPF extensions that provides mechanism for communication and synchronization through a shared data abstraction (SDA). Fortran M [7, 2] makes extensions to Fortran77 for task parallel computations, and also allows some data distribution statements. Braid [6] introduces data parallel extensions to the Mentat Programming Language [9].

Communication libraries like PVM and MPI [13] may be used by the programmer to directly transfer messages from one data parallel task to another. However, such an approach burdens the programmer with having to understand low level details about data distributions and message passing. It is also "hard wired" in that support has to be developed for each instance of communicating data parallel programs. Once a program has been written in this fashion, it will have to be re-implemented if the components with which it interacts are altered or if the "coupling strength" is altered.

7 Conclusions

We have demonstrated that it is possible to link data parallel applications in a flexible and reconfigurable fashion such that re-compilation is avoided and data movement between applications does not have to be hand coded. The fact that large amounts of data are being produced and consumed and the fact that the data is distributed required us to invent a mapping specification that indicates relative consumption and production patterns and data structure linkages. Using this information, we constructed a communication priori schedule that optimized the flow of data between applications. We characterized the mapping specification into four classes and discussed how these classes might be useful for different application interactions.

We demonstrated the utility of our method by applying it to link HPF applications. Our method did not require any language extensions and we were able to implement our method using the Digital HPF compiler and intrinsics without any knowledge of compiler or runtime system internals. Our experiments indicate that coarse grained parallel tasks may be linked in this fashion without much loss in performance.

We are working on enhancing our system in various ways, including generalizing the mapping specification to allow arbitrary operations on the exported arrays, instead of just assignments. This would allow intermediate interpolation processes to be avoided. We are also working on improvements in the system implementation to get better performance.

Acknowledgements

We are grateful to Gagan Agarwal, Chialin Chang, and Shamik Sharma for several thought provoking discussions. Bill Pugh and Pete Keleher reviewed earlier drafts of this paper and pointed out inconsistencies and ill-specified semantics.

References

[1] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[2] K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated Support for Task and Data Parallelism. *Journal of Supercomputing Applications*, 8(2), 1994. Also available as CRPC Technical Report CRPC-TR93430.

[3] Jefferey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. Technical Report 93-04-2, University of Washington at Seattle, January 1994.

[4] C.Koebel, D.Loveman, R.Schreiber, G.Steele Jr., and M.Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[5] Guy Edjlali et. al. Meta-chaos - An Inter-operability layer for Data-Parallel programs. Technical Report In Preparation., Center For Research on Parallel Computation.

[6] E.West and A. Grishaw. Braid: Integrating Task and Data Parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 211–219. IEEE Computer Society Press, February 1995.

[7] I. Foster, M. Xu, B. Avalani, and A. Choudhary. A Compilation System that Integrates High Performance Fortran and Fortran M. In *Proceedings of the 1994 Scalable High Performance Computing Conference*. IEEE Computer Society Press, 1994.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[9] Andrew S. Grimshaw. The Mentat computation model data-driven support for object-oriented parallel processing. Technical Report CS-93-30, University of Virginia, May 93.

[10] M. Haines, B. Hess, P. Mehrotra, J. Van Rosendale, and H. Zima. Runtime Support for Data Parallel Tasks. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 432–439. IEEE Computer Society Press, February 1995.

[11] Christine R. Hofmiester and James M. Purtilo. A framework for dynamic re-configuration of distributed programs. Technical Report CS-TR 3119, University of Maryland, Department of Computer Science and UMIACS, 1993.

[12] Yuan-Shin Hwang, Bongki Moon, Shamik D. Sharma, Ravi Ponnusamy, Raja Das, and Joel H. Saltz. Runtime and language support for compiling adaptive irregular programs. *Software–Practice and Experience*, 25(6):597–621, June 1995.

[13] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[14] M.Ranganathan, A.Acharya, G.Edjlali, A.Sussman, and J.Saltz. Run-time coupling of Data-Parallel programs. Technical Report In Preparation., Center For Research on Parallel Computation.

[15] N.Carriero and D.Gelertner. Linda in context. *Communications of the ACM*, 32(4), 1989.

[16] James Purtillo. The Polylith software toolbus. Technical Report CS-TR-2469, University of Maryland, Department of Computer Science and UMIACS, March 1990.

[17] J. Subhlok, D. O'Hallaron, and T. Gross. Task Parallel Programming in Fx. Technical Report CMU-CS-94-112, School of Computer Science, Carnegie Mellon University, 1994.

[18] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 13–22, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.

[19] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1 and UMIACS-TR-93-36.1, University of Maryland, Department of Computer Science and UMIACS, December 1993.