Runtime Coupling of Data-parallel Programs *

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 mappings between data structures in different data-parallel programs and implements a user-specified consistency model. Mappings are established at runtime and can be added and deleted while the programs being coupled 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 consistency requirements allows buffering of data as well as concurrent execution of the coupled applications. Efficient data movement is achieved by pre-computing an optimized schedule. We describe our prototype implementation and evaluate its performance using a set of synthetic benchmarks. We examine the variation of performance with variation in the consistency requirement. We demonstrate that the cost of the flexibility provided by our coupling scheme is not prohibitive when compared with a monolithic program that performs 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 consistency requirements at run-time allows concurrent execution of interacting programs by buffering the data being communicated. 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 [3] and is able to couple data-parallel programs written in different languages (including High Performance Fortran (HPF) [2], C and pC++ [1]) and using different communication libraries (including Multiblock PARTI [12] and CHAOS [6]).

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 [4, 11], our approach can work with off-the-shelf language implementations as long as the implementations provide a small number of query functions about the distributions of data structures [3].

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.

*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.

To appear in the 1996 ACM International Conference on Supercomputing, Philadelphia, May 25-28, 1996

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 grids corresponding to neighboring regions in space and which periodically exchange data at the boundary. In this case, the array sections in the two 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 of the simulation 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.

Mappings are established at runtime. New mappings may be added between programs while they are in execution and existing mappings may be deleted. For example, dynamic mapping addition could be used to attach a visualization program to a long running simulation long after it has commenced execution.

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 practical 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 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 such that each of the processes in a data-parallel program sees the same number of `acquires` and `releases` at a given logical point in the program execution. This implies 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:

`with` *consistency_specification* $\{asect_1 = asect_2\}$

where $asect_1$ and $asect_2$ are array sections and *consistency_specification* is a consistency specification.

Arrays are referred to by their external names and can be multi-dimensional. 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 being coupled 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.[1] 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 of the assignment appearing in the mapping is called the *consumer* and the owner of the right hand side of the assignment is called the *producer* for that mapping. The array (or array section) that appears on the right hand side of a mapping is called the source array for the mapping and the one that appears on the left hand side 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 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. Because of space limitations, we defer the description of dynamic mapping to a technical report [8].

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

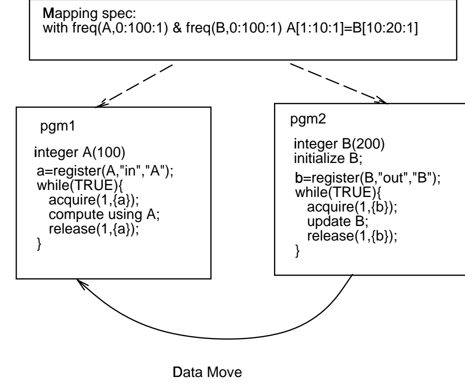

Figure 1: Program text and mappings for example.

3.3 A simple example

In this section, we illustrate the use of the proposed programming model with a simple example. Consider the two programs in Figure 1. Data parallel program pgm_1 `registers` its distributed array A with the global name **A** and the `in` mode. Data parallel program pgm_2 `registers` its distributed array B with the global name **B** and the `out` mode.

This mapping specified in the figure couples the section $[1 : 10 : 1]$ of the one-dimensional array **A** to the section $[10 : 20 : 1]$ of the one-dimensional array **B** with a consistency that is explained as follows.

Consider the variable **B** in pgm_2 :

- Before the first `acquire` $(1, \{b\})$ completes, the zeroth version of B is transmitted to the consumer pgm_1.
- 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` $(1, \{b\})$ returns.
- In between `acquire` $(1, \{b\})$ and the subsequent `release` $(1, \{b\})$, no transfers from B can occur.
- When `release` $(1, \{b\})$ 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` $(1, b)$ completes.

Similarly, consider the variable **A** in pgm_1 :

- Before the first `acquire` $(1, \{a\})$ completes, there is no defined relationship between A and B.
- After the first `acquire` $(1, \{a\})$ 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` $(1, \{a\})$ 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 $(1,\{a\})$ and the subsequent release $(1,\{a\})$ no transfers into array A may occur.
- When release $(1,\{a\})$ is executed, the version counter for A is incremented and pending updates for A may begin for the new version counter.

3.4 Informal semantics

There are two major classes of consistency conditions - *strided* and *wild-card*. Strided conditions can take one of two forms: freq $(array, const_1 : const_2 : const_3)$ or 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 : freq $(array, const_1 : const_2 : *)$ or 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, freq$(array, const_1 : \infty : const_3)$ for strided requests and 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 denote integer constants ≥ 0; and all mappings are of the form $A = B$ where program P_1 exports A and program P_2 exports B.

Fully Constrained Coupling : the form of the consistency specification is:

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

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 $(const_3 + k \times const_4)^{th}$ 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 release(B) and the completion of the following call to acquire(B) [2]. This data must be transferred into A after the $(const_1 + (k \times const_2) - 1)^{th}$ release(A) call has completed and before the following call to acquire(A) completes [3]. The *fully-constrained* model is able to capture a wide range of consistency requirements for relatively closely coupled programs.

Producer-constrained coupling : the form of the consistency specification is:

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

In this model, every $const_4^{th}$ version of B is copied over to A. No data is transferred to A for the first $const_1$ calls to acquire(A). The data must be transferred out of B between the start of the $(const_3 + k \times const_4)^{th}$ call to release(B) and the completion of the following call to acquire(B). This data must be transferred into A at a subsequent call to acquire(A) after the first $const_1$ calls to 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 is not needed.

Consumer-constrained coupling : the form of the consistency specification is:

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

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

Free-running coupling : the form of the consistency specification is:

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

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

[2] If $const_3 = 0$, the first data transfer out of B must happen between the register(B) call and the first call to acquire(B).

[3] If $const_1 = 0$, the first data transfer into A must happen between the register(A) call and the first call to acquire(A).

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.

Since we allow a program to have multiple sources and multiple sinks we can form 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 problems when building an interconnection of applications. We address these issues in greater detail in a technical report [8].

4 Implementation

We have implemented our system on a network of four-processor SMP Digital Alpha Server 4/2100 workstations running Digital Unix 3.2. The nodes are connected by an FDDI network.

The primary goals of our implementation were language independence and efficiency. The concern for language independence prompted the use of the Meta-Chaos library [3], which we describe below. For efficiency, we used three techniques. First, we used asynchronous, one-sided message-passing for inter-application data transfer; 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 being 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 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 for these models, 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 structure) 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. 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 [8].

4.2 Data Transfer

For inter-application data transfer, our library is built on a more basic data movement library called *Meta-Chaos* [3]. *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 computing a canonical representation for the different distributed arrays 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), or it could be as large as the array(eg. irregular distributions). 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.

For portability, *Meta-Chaos* relies on only a small number of query and mapping functions that must be made available by the runtime libraries of the languages in which the applications are written. These functions include queries on index ownership, location and mapping between global and local indices.

For the underlying messaging layer between applications, we used PVM [5]. 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 examined the performance of our system using mini-applications. These mini-applications were designed to eval-

uate our system in four ways: (1) comparison of mapping-based coupling with hand-coded message-passing; (2) comparison of mapping-based coupling with a single monolithic data-parallel program; (3) variation of producer and consumer performance with variation in the consistency requirements; and (4) cost of additional synchronization caused by the coupling system, in particular by the *single-version* guarantee.

To provide a context for our results, we also measured the communication performance on our experimental platform. The application-to-application data transfer rate between two C programs on the network using connection-oriented sockets and transferring 40 KBytes of data per send averaged 24.4 MBits/sec. Inter application data transfer between nodes using PVM and transferring 40 KBytes per send, was measured at 23.5 MBits/sec on average. The rated maximum transfer rate of the network is 100 MBits/sec.

5.1 Comparison with message-passing

We compared the performance of mapping-based coupling and hand-coded message-passing by measuring the time required to transfer a 100x100 integer array between two data-parallel programs. In both cases, Meta-Chaos was used for the actual data movement. 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. The mapping-based coupling scheme incurs additional overhead due to scheduling delays for the threads used for asynchronous communication and due to lock contention between the communication threads and the computation thread.

In this experiment, the producer and the consumer are run on disjoint sets of 4 nodes each. The set of processes for both applications were distributed round-robin over all the processors on these nodes. For the larger configurations, multiple processes were assigned to individual nodes but all processes assigned to the same node belonged to the same application. This ensured that all communication was performed over the network and not via shared memory. Identical process distributions were chosen for both the hand coded send/receive and mapping-based coupling. The data was uniformly partitioned, in a blocked fashion, between all the peer processes in each application.

Table 1 shows the performance of both versions; the timings are averaged over 1000 iterations of both programs. The measure used is the time for a single data transfer; time for generating the schedule is not included. The results show that the overhead of mapping-based coupling with respect to message-passing is very low and is consistently within 0.1 ms and 0.4 ms. For both versions, the time to each transfer decreases from one processor to four. This is due to an increase in the number of communicating nodes and thereby the aggregate communication bandwidth. For larger configurations (the 8- and 16-process configurations), multiple processes are placed on individual nodes and the throughput drops due to contention for the network adaptor.

We also measured the overhead of `acquire` and `release` calls when no mappings were in place. We found that the overhead was negligible.

5.2 Comparison with monolithic programs

We compared the performance of mapping-based coupling and monolithic data-parallel programs by measuring the performance of a simulation over two neighboring grids. The simulation sweeps 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 and structural mechanics. Each of the grids contains 100x100x100 cells (one integer per cell); the two grids are connected at their shared boundary.

Processors (Send/Recv)	mess-passing	mapping-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 1: Comparison of transfer time for mapping-based coupling and direct message passing (ms per send averaged over 1000 iterations).

We compared the performance of three versions:

- A monolithic HPF program that sweeps over a 200x100x100 grid using single forall loop. The graph for this version in Figure 2 is labelled "Monolithic HPF simulation".
- Two HPF programs, one simulating each grid. They perform the same computation as the monolith program; mapping-based coupling is used to exchange data (fill ghost cells) at the boundaries where the grids meet. Each HPF program runs on a different set of nodes. The graph for this version in Figure 2 is labelled "Coupled HPF simulations 1".
- Same as the second version except that an interpolation program is added between the two HPF simulation programs. This represents situations where the grids are not exactly aligned (due to resolution differences or otherwise). In this version, the interpolation program processes data going in both directions. The interpolation program is co-located with one of the simulations. The graph for this version in Figure 2 is labelled "Coupled HPF simulations 2".

The graphs in Figure 2 compare the performance of the three versions. The monolithic version and the coupled version without interpolation perform the same computation and communication. The primary difference is the underlying messaging layer – the monolithic version uses a proprietary version of UDP (a part of the Digital HPF implementation) whereas the coupled version uses PVM.

The coupled version with interpolation performs extra computation and and communication. The computation itself is very simple - just averaging the data values on $grid_1$ and $grid_2$ and writing the result out to $grid_3$ which is then read back by pgm_1 and pgm_2 on each iteration. The additional communication, too, is not expensive as the added communication is local to the node. The interpolation does introduce additional multi-tasking, the effect of which is difficult to quantify. But as shown by the graphs in Figure 2, it is not large.

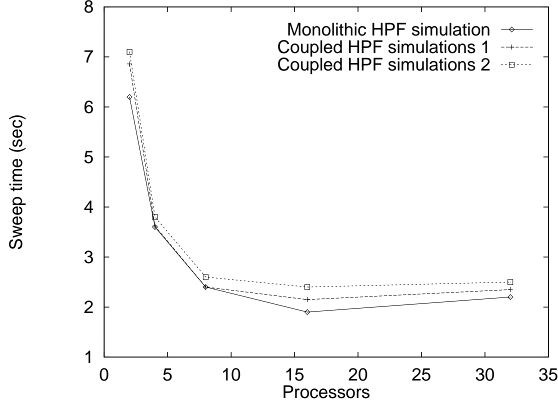

Figure 2: Performance of coupled simulation vs. monolithic coupling

Coupling Type	Producer Loop Time	Consumer wait time
fully-constrained	14.7	8.7
consumer-constrained	1.5	190
producer-constrained	14.0	.41
free-running	1.5	.11

Table 2: Producer loop time and consumer wait time for different consistency specifications (ms).

In all cases, 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. This can be attributed to the relatively high network latency and the relatively small grain of the computation.

These results show that using mapping-based coupling to compose a pair of frequently communicating programs instead of rewriting them into a single monolithic program does not degrade performance unacceptably. Even with an additional interpolation program, the performance loss is not prohibitive.

5.3 Performance impact of consistency requirements

For this experiment, the producer runs in an infinite loop incrementing each of the elements in a 100x100 integer array A on each iteration. The consumer adds all the elements of integer array B on each iteration. The mapping is of the form $A = B$. Both producer and consumer are sequential applications. Each runs on a dedicated node. Table 2 shows the variation of the average loop time for the producer and the average wait time for the consumer for different consistency requirements.

Table 2 shows the impact of changing consistency requirements on 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 which version it 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 consistency requirement is the weakest and the performance is the best. The only guarantee here is that the consumer will observe a *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.

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 3: Worst-case cost of additional synchronization (ms)

5.4 Worst-case cost of additional synchronization

The transfers of data required to implement the consistency requirements can require additional synchronization. This has the greatest impact on performance if (1) the peer processes in the producer application do not already synchronize for computational purposes and (2) the data movement is consumer-initiated which requires consumer processes to wait till all producer processes synchronize and generate a consistent version. We evaluated the worst-case cost of additional synchronization using a mini-application where the producer processes are independent and the consistency model was *consumer-constrained*.

The producer is a data-parallel simulation program and exports an array A which contains the state of the simulation; the consumer is a sequential visualization program and exports an array B which contains the data points for visualization. Each array is a 100 x 100 integer array and the mapping is *consumer-constrained*. We implemented skeleton HPF applications for both the producer and the consumer. We ran the "visualizer" in a tight loop doing only *acquire* and *release* and measured the average wait time for the `acquire` operation. 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. Table 3 shows the results. The synchronization cost is approximately the difference between the *avg consumer wait time* in column 2 and the *avg producer loop time* in column 3.

As shown in Table 3, the worst-case cost of synchronization can be substantial. But note that this is for the relatively rare case of data-parallel programs with independent processes which have been coupled in a *consumer-constrained* model. The cost increases with the number of processors for two reasons: (1) the consumer process has to communicate with an increasing number of producer processes and (2) the potential skew between the otherwise-independent producer processes increases as the number of processes increases.

The sharp increase in the cost from a 4-processor configuration to a 8-processor configuration is due network traffic required for the synchronization. For the four-processor configuration, all communication is local to a single node. 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 not coupled and compared it with the loop time when the producer was coupled; the difference was not significant. This shows that even in this case, only one of the programs, the consumer pays an significant cost.

6 Related Work

Our approach is similar in some respects to the software bus approach used in Polylith [10]. Our approach differs from Polylith in that it 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.

Linda [9] 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 code 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.

Communication libraries like PVM and MPI [7] 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 consistency requirements are 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 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.

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

[3] Guy Edjlali et. al. Meta-Chaos - an inter-operability layer for data-parallel programs. Technical Report In Preparation., Center For Research on Parallel Computation, 1996.

[4] I. Foster, M. Wu, B. Avalani, and A. Choudhari. 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.

[5] 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.

[6] 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 Practise and Experience*, 25(6):597–621, June 1995.

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

[8] M.Ranganathan, A.Acharya, G.Edjlali, A.Sussman, and J.Saltz. Run-time coupling of data-parallel programs. Technical Report CS-TR-3565, UMIACS TR-95-116, University of Maryland, 1995.

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

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

[11] J. Subhlok, D. O'Hallaron, and T. Gross. Task parallel programming in Fx. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburg, 1994.

[12] 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.