Taking Advantage of Collective Operation Semantics for Loosely Coupled Simulations [*]

Joe Shang-Chieh Wu and Alan Sussman
UMIACS and Department of Computer Science
University of Maryland, College Park, MD 20742, USA
email: {meou,als}@cs.umd.edu

Abstract

Although a loosely coupled component-based framework offers flexibility and versatility for building and deploying large-scale multi-physics simulation systems, the performance of such a system can suffer from excessive buffering of data objects which may or may not be transferred between components. By taking advantages of the collective properties of parallel simulation components, which is common for data-parallel scientific applications, an optimization method, which we call **buddy-help***, can greatly enhance overall performance. Buddy-Help can reduce the time taken for buffering operations in an exporting component, when there are timing differences across processes in the exporting component. The optimization enables skipping unnecessary buffering operations, once another process, which has already performed the collective export operation, has determined that the buffered data will never be needed. Because an analytical study would be very difficult due to the complexity of the overall coupled simulation system, the performance improvement enabled by buddy-help is investigated via a micro-benchmark specifically designed to illustrate the behavior of coupled simulation scenarios under which buddy-help can provide performance gains.*

1 Introduction

Integrating well-tested modules, each of which models part of the target system, and deploying them as a loosely coupled framework, rather than developing a single tightly coupled monolithic code, is a more efficient and flexible way to develop complex or large-scaled systems and has been used in many scientific simulations. Applications include large-scale space science simulations solving the magnetohydrodynamics (MHD) equations [7] and multi-scale multiresolution petroleum reservoir simulation [9].

The various coupled application programs (*components*) may use different simulation time and space scales, either because of the scale of the phenomena being modeled or because of the numerical techniques being employed. Some components may themselves be parallel programs, perhaps implemented using message passing [16] or threads. The interfaces between components, which are the shared boundaries or the overlapped regions between physical models, must be made consistent in both time and space to obtain correct results. How to exchange data between coupled programs is a major concern in these scenarios. The spatial resolution problem involves proper interpolation between the grids used in the two components, and several projects, including the MxN working group in the Common Component Architecture (CCA) Forum [2], are currently addressing some of these issues. Our earlier work [18] describes a method for solving the temporal consistency issue. That work describes a temporal consistency model in which each exported data object must be buffered by the runtime system implementing the model, until there is no possibility that an object will be requested by an importing component. That can be determined by determining that either there is no importing component for objects of that type or because importer requests that have already been processed can be shown to ensure that the object in question cannot be requested. Although this approach ensures the correctness of the data exchange mechanism, overall system performance may suffer from unnecessary buffering, when one process in a data exporting (parallel) component performs the collective export operation early relative to the other processes (i.e. it is the first process to execute the export runtime library call). In that case, other processes can use the information produced in resolving the call in one process for later calls in other processes in the exporting parallel component.

[*]This research was supported by the National Science Foundation under Grant #CNS-0615072 and NASA under Grants #NAG5-12652 and #NNG06GE75G.

This paper focuses on the the temporal consistency issue by taking advantage of the semantics of collective operations, which ensure that *all* processes in a parallel component must make the same sequence of export (or import) operations (similar to the required behavior of parallel programs that use MPI collective operations [16]). Collective operations are commonly used in many single program multiple data (SPMD) parallel program implementations, and require that (1) the same code is running on all processes, (2) the dataset is partitioned across the multiple processes, and (3) each process performs computation on the part of the data object it owns. Moreover, collective operations, such as broadcast, barrier, reduce, etc., require *all* processes in the *same* program to execute the same function with appropriately matching parameters. These operations are well supported in popular parallel libraries such as PVM [17] and MPI [16], and play important roles in SPMD programs. Performance studies have shown that some parallel programs spend more than 80% of their interprocessor communication time in collective operations [15].

Data exchange between shared or overlapping regions in different coupled simulation components can be viewed as a collective operation, where the data to be transferred spans both multiple processes in a single component and the processes in two separate components. That is because the exchange is not complete until *all* involved processes transfer their share of the data (however it does not require that all processes transfer data at the same time, meaning no barrier synchronization is required). In addition, the collective operation semantics guarantee that all processes in the same exporting component must make the *same* decision about which copies of the generated data should (and should not) be transferred to the corresponding importing program(s). When some of the processes in the data exporting component run more slowly than other others, perhaps because of imperfect load balancing within the component or for other application-specific reasons, those slower processes can be sped up if the decision about which transfers to make are performed by the fastest process in the component (the one that executes the export call first). The rest of the paper describes this optimization and its implications in more detail, and is organized as follows. Section 2 describes related work, Section 3 outlines the overall coupled simulation architecture we have designed in prior work, Section 4 explains the optimization strategy in more detail and expands on the use of collective semantics in the framework, Section 5 provides some micro-benchmark experimental results, and we conclude in Section 6.

2 Related Work

Both Interoperable MPI(IMPI) [6], which is a set of protocols across different MPI implementations, and MPICH-G2 [14], which is a grid-enabled implementation of the MPI v1.1 standard, allow one MPI program to run in heterogeneous environments that are composed of different architectures and operating systems. These systems mainly focus on heterogeneous integration, and higher level coupling issues between application components must be handled by the participating components.

Data exchanges between distributed data structures are provided in other software packages, including Meta-Chaos [3], InterComm [11], Parallel Application Work Space (PAWS) [4], the Model Coupling Toolkit (MCT) [10], Roccom [8], the Collaborative User Migration User Library for Visualization and Steering (CUMULVS) [5], and the MxN working group in the Common Component Architecture (CCA) Forum [1, 12, 2]. All of that work targets mapping of the elements of distributed data structures to the processes in an application component, as well as distributed data exchange between participating (parallel) programs, with the higher level coupling and integration left to the participating application components.

Collective operations, which must be invoked by a set of processes in a parallel program to perform an operation, are widely used in parallel libraries, such as PVM [17] and MPI [16], either for collective data movement (broadcast, scatter, gather, etc.) or for collective computation (maximum, summation, etc.).

3 System Architecture

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

Although each program must define its contributions (called *regions* in our framework) to a data transfer, the related counterparts on the other side of the data transfer do not need to be defined. From the point of view of a data exporting program, the program defines its regions once, and exports the desired data as often as it desires, when a new, consistent version of the data across the parallel program is produced (note that the data for a region can span multiple processes in the program, so the parallel program must ensure that a consistent version is exported.) The program does not have to concern itself about which and how many programs will receive the data, or even whether a data transfer will actually occur. Data importing programs also only define their regions once, and import data as needed, without knowing anything about the corresponding exporters. Examples of user programs are shown in Figure 1.

```
define region r1     | define region r1
define region r2     | ...
define region r3     | for(...) {
...                  |   import r1
for(...) {           |   computation
  export r1          | }
  export r2          |
  export r3          |
  computation        |
}                    |
Exporter P0 code     | Importer P1 code
```

Figure 1. Example exporter and importer programs

3.1 Runtime Coupling and Match

The connection between importer and exporter programs is provided by a framework-level configuration file that (1) is separate from all user programs, (2) is read in the initialization stage of each participating program, and (3) contains runtime environment information for executing participating user programs as well as information about how to connect the imported and exported regions in each program. An example is shown in Figure 2.

```
P0 cluster0 /home/meou/bin/P0 16 ...
P1 cluster1 /home/meou/bin/P1 8  ...
P2 cluster1 /home/meou/bin/P2 32  ..
P4 cluster1 /home/meou/bin/P4 4  ...
#
P0.r1 P1.r1 REGL 0.2
P0.r1 P2.r3 REG  0.1
P0.r2 P4.r2 REGU 0.3
```

Figure 2. An example configuration file

Separating the export/import connections from user programs makes coupling flexible – even recompiling a program is not necessary if another program that it will communicate with is replaced. In addition, in its initialization stage the framework can determine whether any exported or imported regions are not involved in the connection specification, enabling both early detection of an incorrect coupling specification (e.g., and imported region that has no corresponding exported region) and a low overhead implementation for an exported region that is not imported by any other program (i.e., the connection specification has no entries for that exported region).

We require that each data object in each export/import region be associated with an increasing simulation timestamp, so data exchange between two regions can not start until a matched export timestamp is identified for each import request. However, in a set of loosely coupled programs programs that have been developed independently, in general it may happen that an exported region with the exact timestamp requested by an import in another program might never become unavailable. Our framework therefore uses approximate matching [18], which defines a per connection *match policy* and a related *tolerance*, to handle this issue. Roughly speaking, given a requested timestamp and the user-defined tolerance, an *acceptable region* can be identified, and then based on the match policy, one of the candidate timestamps in the acceptable region will be chosen as the match for a given import operation. For example, in Figure 2 the connection `P0.r1 P1.r1` has a match policy `REGL` and tolerance `0.2` which means if the requested timestamp is x, the acceptable region would be $[x-0.2, x]$, and the match is the one closest to x if more than one export timestamp is located in the interval $[x-0.2, x]$.

In an implementation of approximate matching, compared to the standard exact matching (where for any given import/export pair it is possible to determine whether the result is `MATCH`/`NO MATCH`), `PENDING` is a possible result for an approximate match. That means that, based on data exported up to the point in time that the match is performed (presumably because of an import request), the *best* match cannot be decided, either because exported data in the acceptable region has not yet been generated, or because an export that has not yet occurred *might* be a better match (closer to the timestamp requested by the import) than the current candidate.

4 Collective Semantics

Compared to traditional collective operations, such as broadcast (copying data from one to a group of processes) and reduce (aggregating with some binary operation data supplied by a group of processes) in PVM [17] and MPI [16], data transfers in our framework also exhibit collective properties. This means that *all* processes in the same program must execute the same export (or import) operations in the same order (but not necessarily at the same time), with appropriately matching parameters. Formally the following property must always hold in our framework:

> **Property 1** If one process transfers (exports or imports) data with timestamps $t_1, \ldots, t_n$ during execution, all other processes in the same program must also transfer data with those timestamps, in the same order.

To support and monitor collective behavior at runtime, our framework implementation employs an extra process in each program, called the representative (or *rep* for short), to act as a low-overhead control gateway [18]. For example when the *rep* in an exporting program receives a request from an importing program, it (1) forwards the request to all processes in the exporting program, (2) collects the responses from all processes, (3) combines all responses to produce the final answer to the request, and (4) sends back the final answer to the requester (to the *rep* of the importing program).

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

It is incorrect for some of the responses to be `MATCH` and some to be `NO MATCH` for the same request, because only those processes whose responses are `MATCH` try to transfer data, which is a clear violation of Property 1. It is also incorrect if the matched timestamps from those `MATCH` responses are not the same, because those processes will try to transfer data with different timestamps and Property 1 would not hold again.

Property 1 is maintained if all responses are the same. Interestingly, it is still legal if the collective responses are a mixture of `PENDING` and `MATCH` or a mixture of `PENDING` and `NO MATCH`. This situation means that some processes are running more slowly than others (e.g., either because of load imbalance or because of other application-specific properties), such that when receiving forwarded requests the *best* match cannot yet be decided (so their responses are `PENDING`s.) Based on the guarantee (because of the collective nature of export and import operations) that those slower processes will make the same decisions as their faster peers, the answer sent by the *rep* is `MATCH` if the collective responses are a mixture of `PENDING` and `MATCH` and is `NO MATCH` if the collective responses are a mixture of `PENDING` and `NO MATCH`.

4.1 Buddy-Help

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

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

When the data importing program runs more slowly than the related exporting counterpart, as shown in Figure 3(a), the timestamp of a newly generated data object will *pass* the latest acceptable region which is identified by the last request timestamp and a user-defined tolerance. Buffering of this generated data object is necessary because it might be a match for future requests. Although the buffering operation may be time-consuming when the data size is large, the overall application performance will not be affected much because the data exporting program is not the slowest component in the whole system.

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

However if the new generated data object, call it $\mathbf{A}@t$, (a distributed array $\mathbf{A}$ with simulation timestamp t), is in one of acceptable regions R, as shown in Figure 3(b), in general buffering is necessary because $\mathbf{A}@t$ *might* be the match. If the *next* generated data object $\mathbf{A}@t'$ is outside region R, then $\mathbf{A}@t$ is confirmed as the match for this region R, and the buffering step was indeed required. However, if $\mathbf{A}@t'$ is also located in region R, $\mathbf{A}@t'$ would be a better match, and the system could free the buffer for object $\mathbf{A}@t$.

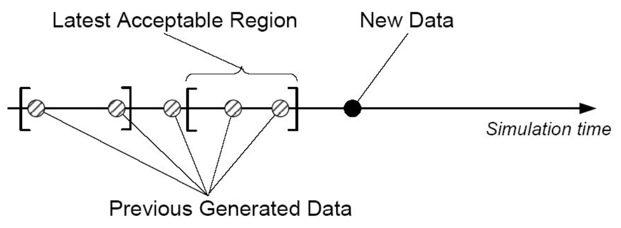
It is no surprise that much unnecessary buffering can occur in the framework if multiple objects are exported that fall in one acceptable region – which can easily occur in coupling physical simulation components that act on different time scales. Formally, if data objects $O_1, \ldots, O_{n(i)}$ are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

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

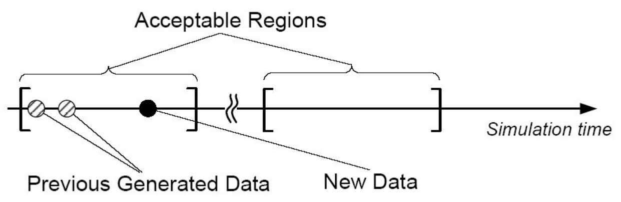
If a total of N requests are received (so that the acceptable regions are $R_1, \ldots, R_N$) during the program execution, and all acceptable regions are mutually disjoint ($R_i \cap R_j = \emptyset$, $i \neq j$), the total time T_{ub} spent on unnecessary buffering is:

$$T_{ub} = \sum_{i=1}^{N} T_i = \sum_{i=1}^{N} \sum_{k=1}^{n(i)-1} t_k \tag{2}$$

Compared to currently used ad-hoc tightly coupled approaches, approximate matching and buffering of generated data are two extra tasks that our framework must perform,



(a) Slower Importer



(b) Slower Exporter

Figure 3. Buffering Exported Data

and it is clear that T_{ub} plays an important role in overall performance when the data exporting program runs more slowly than the related importing counterpart.

One way to decrease T_{ub} is taking advantage of Property 1 described previously. More precisely, if for a given request the collective responses in the *rep* are a mixture of `MATCH` (or `NO MATCH`) and `PENDING`, the rep not only sends the final answer (which is `MATCH` or `NO MATCH`) to the requester, but also sends it to those processes whose responses are `PENDING` (we call this *buddy-help*.) In this way those slower processes can know the *right* match for this request, and avoid unnecessary buffering of data objects that cannot possibly be a match, even *before* the data objects are generated by export operations in those slower processes.. Decreasing T_i (and therefore T_{ub}) in those slower processes matters for overall performance, because the processes that benefit from buddy-help are the slowest processes in the slower program – and therefore are the performance limiting factor for that pair of coupled programs.

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

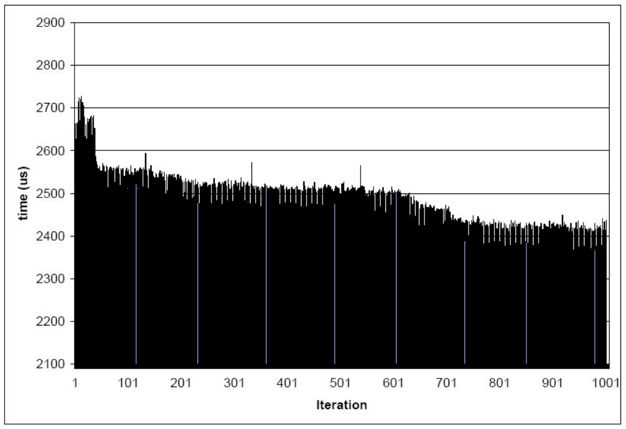
5 Experiment

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

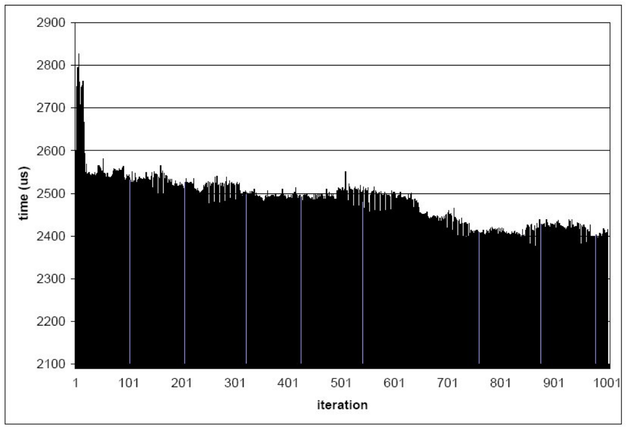
- Solve $u_{tt} = u_{xx} + u_{yy} + f(t,x,y)$, a two dimensional diffusion equation with a forcing function $f(t,x,y)$ which can be viewed as the external input for $u(t,x,y)$.
- Program U, which computes $u(t,x,y)$, owns a 1024 *x* 1024 array which is evenly distributed among the participating processes.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

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

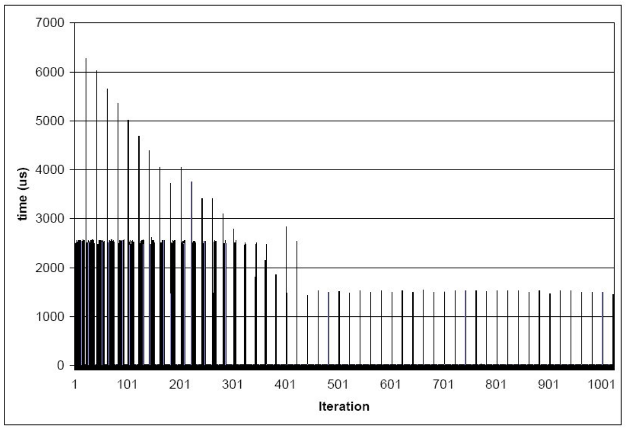
Figure 4(a) shows the case when the importer program U has only 4 processes and is running more slowly than the exporter program F. In this case every exported data object will be saved in the framework buffer because there is no way to know which exported data objects might be needed for a match – therefore the execution time for all 1001 data



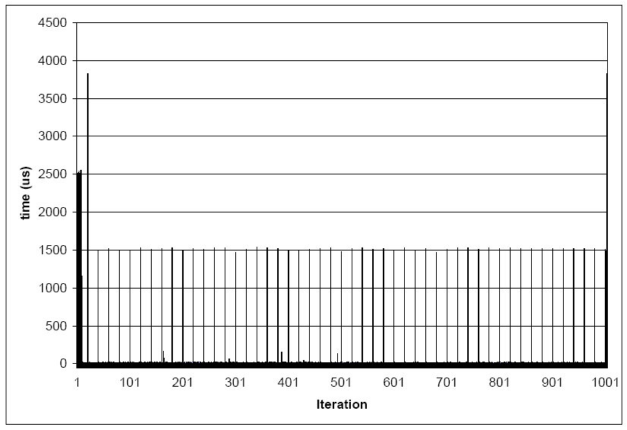
(a) Coupled with 4 Importer Processes



(b) Coupled with 8 Importer Processes



(c) Coupled with 16 Importer Processes



(d) Coupled with 32 Importer Processes

Figure 4. Data Exporting Time for the Slowest Export Process

exports should be similar and Figure 4(a) confirms that, except for early iterations (where the time is 8% greater) and after 600 iterations (where the time is 4% less). The extra 8% is a result of initialization of the framework and its underlying data structures. The 4% decrease in later iterations is likely the result of less congestion in the network and a lighter workload in the framework, because the times shown in Figure 4(a) are from the slowest process p_s in program F, and after 600 iterations all other processes p_i, i = 1, 2, and 3, in program F have already completed.

By keeping the size of the distributed data array fixed (1024 *x* 1024), the program U runs faster as the number of importer processes increases – because less computation is performed for each importing process. Figure 4(b) shows the case when the importer program U has 8 processes, but it is still running slower than the exporter program F. The result is very similar to the case of 4 processes.

The results start to become more interesting when program U has 16 processes, as shown in Figure 4(c). Here U catches up to process p_s in program F, and a typical scenario is shown in Figure 5, where **D**@t denotes the distributed data **D** at the simulation time t. Process p_s receives the first data request **D**@20 after exporting 14 copies of **D** in line 5. Because the matching policy is REGL (described in Section 3.1) and the tolerance is 2.5, the acceptable region is [17.5, 20] and the reply from process p_s is {**D**@20, `PENDING`, **D**@14.6}, which means that for the request **D**@20, the answer is `PENDING` and the current latest exported data is **D**@14.6. Once the answer is generated, process p_s knows immediately that any version of **D** exported with a timestamp less than 17.5, as in lines 10-11, can be discarded it is not in the acceptable region.

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

This pattern occurs again after the match **D**@19.6 is produced by process p_s. Because extra memory allocations and deallocations `memcpy`s are performed by process p_s, as shown in the beginning part of Figure 4(c), the processes in program U have a chance to catch up so that between successive data transfers new data requests will show up earlier, and the number of skipped data copies increases (so T_i starts to decrease.) For example 4 `memcpy`s are skipped in lines 10-13 and then 7 `memcpy`s are skipped in lines 26-29 of Figure 5. Eventually the optimal state, as shown in Figure 6, is reached and maintained, where only the matched data are buffered in the framework. The optimal state has the following characteristics:

- For each matched and then transferred data object, a corresponding buddy-help message will be received early enough by a slow exporter process. (In the example, that is process p_s.)
- For slow exporter processes, the framework can determine which versions (timestamps) of exported data objects will be requested by the corresponding importer program even before those data are exported, and only the matched data objects are buffered in the framework.
- The remaining exported data that is not a match will not be saved by the framework. Namely T_i is equal to 0 once the optimal state is entered.

By keeping the exporter program and its participating processes fixed, the exporter program can reach the optimal state earlier if the importer program is running faster. The reason is that when the importer program is running faster, the related exporter processes will receive the data requests earlier, and based on the information provided by buddy-help, unmatchable data can be identified earlier and more `memcpy`s can be skipped (so T_i starts to decrease.) This claim can be validated from the data in Figures 4(c) and 4(d). Both configurations have the exact same exporter program and around 400 iterations are needed to reach the optimal state when the importer program U has 16 processes, but only around 25 iterations are needed to reach the optimal state when the importer program U has 32 processes.

The performance benefits of avoiding unnecessary buffering from the buddy-help optimization depend on the ratio of the size of the acceptable region to the inter-arrival time between successive importer match requests. Consider the following example. If the matching policy is REGL and the precision is 5.0, the result *with* buddy-help is shown in Figure 7. After receiving the request for **D**@10.0, the acceptable region would be identified as [5.0, 10.0], and the

1	export **D**@1.6, call memcpy.
2	export **D**@2.6, call memcpy.
3	⋮
4	export **D**@14.6, call memcpy.
5	receive request for **D**@20,
6	reply {**D**@20, `PENDING`, **D**@14.6}.
7	remove **D**@1.6, ⋯, **D**@14.6.
8	receive buddy-help {**D**@20, `YES`, **D**@19.6}.
9	remove **D**@16.6.
10	export **D**@15.6, *skip* memcpy.
11	export **D**@16.6, *skip* memcpy.
12	export **D**@17.6, *skip* memcpy.
13	export **D**@18.6, *skip* memcpy.
14	export **D**@19.6,
15	call memcpy,
16	send **D**@19.6 out.
17	export **D**@20.6, call memcpy.
18	export **D**@21.6, call memcpy.
19	⋮
20	export **D**@31.6, call memcpy.
21	receive request for **D**@40,
22	reply {**D**@40, `PENDING`, **D**@31.6}.
23	remove **D**@19.6, ⋯, **D**@30.6.
24	receive buddy-help {**D**@40, `YES`, **D**@39.6}.
25	remove **D**@31.6.
26	export **D**@32.6, *skip* memcpy.
27	export **D**@33.6, *skip* memcpy.
28	⋮
29	export **D**@38.6, *skip* memcpy.
30	export **D**@39.6,
31	call memcpy,
32	send **D**@39.6 out.
33	export **D**@40.6, call memcpy.
34	⋮

Figure 5. A Typical Buddy-Help Scenario

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

Figure 6. Optimal State

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

Figure 7. With Buddy-Help

exported data object **D**@4.6 in line 8 will not saved because it is outside of the acceptable region. However all exported data in lines 9-11, which are within the acceptable region, are not saved either because they are not the match, **D**@9.6, which became known via the buddy-help mechanism. Figure 8 shows a different result *without* buddy-help for the same configuration. In that example, whenever *acceptable* data is exported, as shown in lines 9-18, the new exported data object *is* the best current candidate for a match, so must be saved, and the previous best candidate can be safely deleted. The final match will be identified only after a data object is exported *outside* the acceptable region, which is **D**@10.6 in lines 19-21.

The buddy-help message, which is the answer from the fastest process in an exporter program, plays an important part here – the farther the fastest process progresses, the more help the slowest process can get. However the processes in most scientific data-parallel programs will not usually get out of sync by too much, because data exchanges between the processes within the program occur relatively frequently, loosely synchronizing the processes. But if (1) at least one of the processes p_f acts as a data source, which receives external data and performs its computation without using data from its peer processes, and (2) non-blocking data transfers (such as MPI_Isend) or advanced facilities such as Remote Direct Memory Access (RDMA) over InfiniBand [13] are used for intra-program communication such that multiple copies of the computed data objects (with different timestamps) can be kept in the same program, then the fastest process has an opportunity to stay ahead and to help other, slower peer processes.

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

Figure 8. Without Buddy-Help

6 Conclusion

We have describe an optimization, called *buddy-help*, that improves the overall performance of a loosely-coupled framework for coupled simulations with parallel components. The method reduces the time to export data from the slowest component in a coupled set of components. By taking advantages of the collective properties of the data transfers operations in the framework, this performance enhancement can be achieved by eliminating the cost of buffering operations that are otherwise required. We have shown the effectiveness of the mechanism in a carefully designed micro-benchmark and shown other related scenarios.

This is a work in progress – we are currently investigating several issues related to our coupling framework, including integration with non-blocking data transfers or advanced data transfer facilities in modern high performance networks (e.g., RDMA over InfiniBand), the performance effects of finite buffer space in a coupled component, and the applicability of the framework to large-scale scientific simulation applications.

References

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Com-

mon Component Architecture for high-performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*. IEEE Computer Society Press, 1999.

[2] F. Bertrand, R. Bramley, A. Sussman, D. E. Bernholdt, J. A. Kohl, J. W. Larson, and K. B. Damevski. Data redistribution and remote method invocation in parallel component architectures. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*. IEEE Computer Society Press, 2005.

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

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

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

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

[7] T. I. Gombosi, K. G. Powell, D. L. D. Zeeuw, C. R. Clauer, K. C. Hansen, W. B. Manchester, A. J. Ridley, I. I. Roussev, I. V. Sokolov, Q. F. Stout, and G. Tóth. Solution-adaptive magnetohydrodynamics for space plasmas: Sun-to-Earth simulations. *IEEE Comput. Sci. Eng.*, 6(2):14–35, 2004.

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

[9] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, M. Peszynska, R. Martino, M. Wheeler, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar. A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience*, 17(11):1441–1467, 2005.

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

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

[12] S. Lefantzi, J. Ray, and H. N. Najm. Using the Common Component Architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS 2003)*. IEEE Computer Society Press, 2003.

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

[14] MPICH-G2. http://www3.niu.edu/mpi.

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

[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.

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

[18] J. S.-C. Wu and A. Sussman. Flexible control of data transfers between parallel programs. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 226–234. IEEE Computer Society Press, November 2004.