Enabling Workflows in GridSolve: Request Sequencing and Service Trading

Yinan Li[1], Asim YarKhan[1], Jack Dongarra[1,2,3], Keith Seymour[1], and Aurèlie Hurault[4]

[1] University of Tennessee, Knoxville, TN USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN USA
[3] University of Manchester, Manchester, UK
[4] University of Toulouse, France

Abstract. GridSolve employs a RPC-based client-agent-server model for solving computational problems. There are two deficiencies associated with GridSolve when a computational problem essentially forms a workflow consisting of a sequence of tasks with data dependencies between them. First, intermediate results are always passed through the client, resulting in unnecessary data transport. Second, since the execution of each individual task is a separate RPC session, it is difficult to enable any potential parallelism among tasks. This paper presents a *request sequencing* technique that addresses these deficiencies and enables workflow executions. Building on the request sequencing work, one way to generate workflows is by taking higher level service requests and decomposing them into a sequence of simpler service requests using a technique called *service trading.* A service trading component is added to GridSolve to take advantage of the new dynamic request sequencing. The features described here include automatic DAG construction and data dependency analysis, direct inter-server data transfer, parallel task execution capabilities, and a service trading component.

1 Introduction

GridSolve [5] employs a brokered-RPC client-agent-server model for handling computational problems in a distributed computing or grid environment (Figure 1). A complete RPC session in GridSolve consists of two stages. In the first stage, the client sends a request for a remote service to the agent, which returns a list of capable servers ordered by some measure of their capability. The actual remote service call takes place in the second stage. The client sends input data to a selected server; the server handles the request and returns the result back to the client.

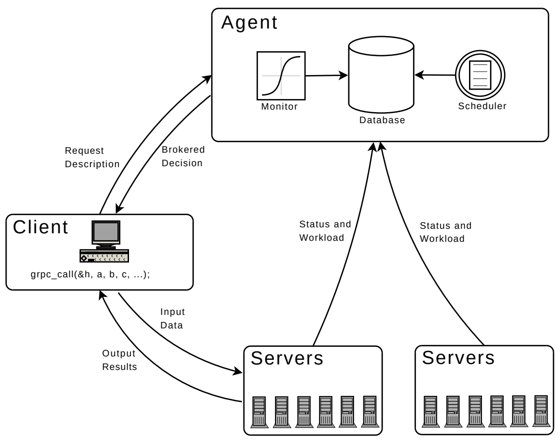


Fig. 1. The standard RPC-based computation model of GridSolve.

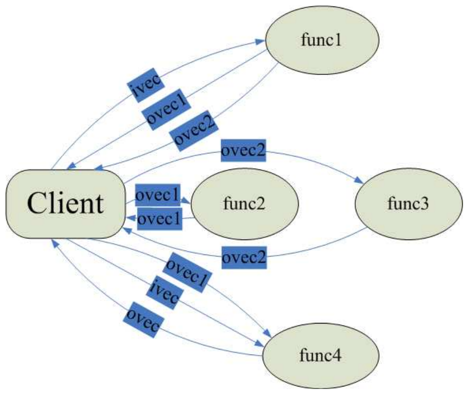


Fig. 2. An example of the standard data flow in GridSolve.

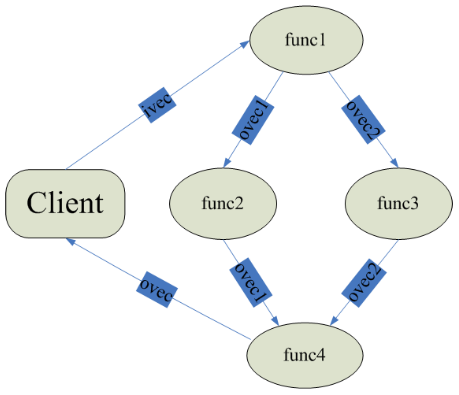


Fig. 3. The data flow in Figure 2 with direct inter-server data transfer.

This request-reply RPC model forms a star topology of requests and replies with the client at the center, since all data traffic must involve the client. The is efficient for solving computational problems consisting of a single task. However, when a computational problem forms a workflow consisting of a set of service requests with data dependencies, this model is highly inefficient due to two deficiencies. First, intermediate results are passed among tasks via the client, resulting in additional, unnecessary data traffic between the client and the servers. Second, since the execution of each individual task is a separate RPC session, it is difficult to exploit the potential parallelism among tasks.

For example, consider the following sequence of GridRPC calls:

```
grpc_call("func1", ivec, ovec1, ovec2, n);
grpc_call("func2", ovec1,n);
grpc_call("func3", ovec2, n);
grpc_call("func4", ovec1, ovec2, ovec, n);
```

In this example, the outputs of `func1`, namely `ovec1` and `ovec2`, are returned back to the client and immediately sent from the client to the servers running `func2` and `func3`, resulting in two unnecessary data movements. Figure 2 illustrates the data flow for the above calls. This example demonstrates that when data dependencies exists among tasks, it may be unnecessary to transfer intermediate results back to the client, since such results will be needed immediately by the subsequent tasks.

To eliminate unnecessary data traffic involving the client, GridSolve has incorporated and enhanced a technique called *request sequencing*. An earlier version of GridSolve implemented a much simpler version of request sequencing [10] [2] which avoided unnecessary data transfers back to the client, but forced all the requests involved in the sequence to run on a single server and ignored any potential parallelism in the tasks. The current work improves on the older request sequencing techniques by exposing and exploiting parallelism in the tasks.

The objective of this work is to provide a technique for users to efficiently solve computational problems in GridSolve by constructing workflow applications that consist of a set of tasks, among which there exist data dependencies.

In GridSolve request sequencing, a request is defined as a single GridRPC call to an available GridSolve service. GridRPC [11] is a standardized API that describes a Remote Procedure Call (RPC) interface in a Grid computing environment. The terms request, service request, and task are used interchangeably in this paper. For each workflow application, the set of requests is scanned, and the data dependency between each pair of requests is analyzed. The output of this analysis is a DAG representing the workflow. The workflow scheduler then schedules the DAG to run on the available servers. A set of tasks can potentially be executed concurrently if the DAG permits it. In section 2 the details of request sequencing are presented.

In order to eliminate unnecessary data transport when tasks are run on multiple servers, the standard RPC-based computational model of GridSolve must be extended to support direct data transfer among servers. Figure 3 illustrates the alternative data flow of Figure 2, with direct data transfer among servers. Supporting direct inter-server data transfer requires server-side data storage. A server may have already received some input arguments and stored them to the local storage, while waiting for the additional arguments. In addition, a server may store its outputs to the local storage in order to later transfer them to the servers that need them. In section 3, we present a method of using special data files as data storage. Section 4 discusses approach to scheduling the workflow on the available resources, and section 5 presents the request sequencing API in GridSolve.

Building on the request sequencing work, one possible way to generate workflows is by taking higher level service requests and decomposing them into a sequence of simpler service requests using a technique called *service trading.* Section 6 presents some of the ideas involved in service trading, though a detailed presentation of service trading in GridSolve is available separately [7].

2 Workflow Modeling and Automatic Dependency Analysis

2.1 Data Dependencies and Directed Acyclic Graphs

In GridSolve request sequencing, the workflow is represented by a DAG. Each node in the DAG represents a request (task), and each edge represents the data dependency between requests. Consider our usual example of GridRPC requests, which have various input (*ivec*) and output (*ovec*) data items.

```
grpc_submit("return_int_vector",ivec,n,ovec);
grpc_submit("vpass_int", ovec, n);
grpc_submit("iqsort", ovec, n);
grpc_submit("int_vector_add5",n,ovec,ovec2);
```

The dependencies between the requests are implicit in the definition of the data items (input, output, inout) and in the sequential order that the requests are submitted. Given a set of GridRPC calls, we identify four types of data dependencies:

Input-After-Output (RAW) Dependency A service reads a data argument after a previous service writes that data argument. The actual data involved in the dependency will be transferred directly between servers, without the client being involved.

Output-After-Input (WAR) Dependency One service writes an data argument after a previous service reads that data argument. The write cannot proceed until the read completes. This is sometimes also called a *false* dependency, since it can be eliminated by creating copies of the data. However, we do not attempt to eliminate false dependencies at this time.

Output-After-Output (WAW) Dependency This represents the case where two requests in the sequence write the same output argument. The output of the request that is depended on will not be transferred back to the client since the shared data will be overwritten shortly by the depending request.

Conservative-Scalar Dependency This type of scalar data dependency occurs in the conservative sequencing mode that will be introduced shortly.

If all these dependencies are maintained then the sequential program order will be preserved. The first three types of dependencies apply to non-scalar arguments such as vectors and matrices. Figure 4 gives an example DAG with all types of non-scalar data dependencies (RAW, WAR, and WAW). For scalar arguments, it is much more difficult, and even impossible to determine if two scalar arguments are actually referencing the same data, since in GridSolve scalar data can be passed by value as well as by reference. Our solution is to provide users with several modes that use different approaches for analyzing data dependencies among scalar arguments. The supported modes are as follows:

Optimistic Mode In this mode, scalar arguments are ignored when analyzing data dependencies.

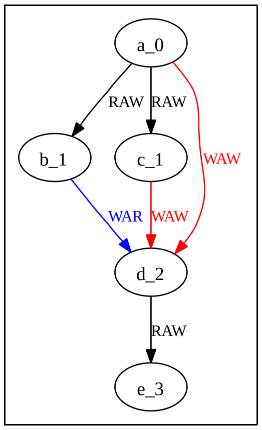


Fig. 4. An example DAG with all three kinds of non-scalar data dependencies.

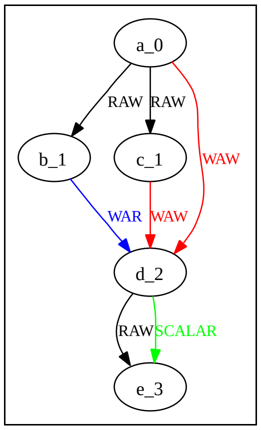


Fig. 5. The example DAG in Figure 4 with an additional scalar data dependency.

Conservative Mode In this mode, two successive requests with one having an input scalar argument and the other having an output scalar argument, are viewed as having a conservative-scalar dependency, if these two scalar arguments have the same data type.

Restrictive Mode In this mode, scalar arguments are restricted to be passed by reference, and data dependencies among scalar arguments are analyzed as usual.

Figure 5 depicts what it looks like in Figure 4 with one conservative scalar dependency.

2.2 Automatic Dependency Analysis and DAG Construction

In GridSolve, non-scalar arguments are always passed by reference. In addition, each argument has some attributes associated with it specified in the service description. These attributes describe the data type of the argument (integer, float, double, etc.), the object type of the argument (scalar, vector, or matrix), and the input/output specification of the argument (`IN`, `OUT`, or `INOUT`). These attributes, along with the data reference, can be used to determine if two arguments refer to the same data item. The pseudo-code of the algorithm for automatic DAG construction and dependency analysis is presented in Algorithm 1. Notice that in the algorithm, each node is assigned a rank, which is an integer representing the scheduling priority of this node. The algorithm for workflow scheduling and execution uses this rank information to schedule nodes to run. The algorithm for workflow scheduling and execution is presented in Section 4.

As an example, considering the following workflow (this workflow is programmed using the API functions that will be introduced in Section 5):

```
grpc_sequence_begin(OPTIMISTIC_MODE);
grpc_submit("return_int_vector",ivec,n,ovec);
grpc_submit("vpass_int", ovec, n);
grpc_submit("iqsort", ovec, n);
grpc_submit("int_vector_add5",n,ovec,ovec2);
grpc_sequence_end(0);
```

The DAG produced by the above algorithm (Figure 6) may contain redundant edges from the perspective of both execution and data traffic. For example, in Figure 6, the RAW dependency between `return_int_vector` and `int_vector_add5` is redundant, since the input argument `ovec` of `int_vector_add5` will come from `iqsort` instead of `return_int_vector`. Removing this redundant edge will affect neither the execution order nor the effective data flow of the DAG. The final step in building and analyzing the DAG is to remove all such redundant dependencies. Figure 7 shows the DAG in Figure 6 after all redundant edges are removed.

Algorithm 1 The algorithm for automatic DAG construction and dependency analysis.

```
1: Scan the set of tasks, and create a DAG node for each task;
2: Let $NodeList$ denote the list of nodes in the DAG;
3: Let $N$ denote the number of nodes in the DAG;
4: for $i = 1$ to $N - 1$ do
5:   Let $P$ denote node $NodeList[i]$, and $PArgList$ denote the argument list of node P;
6:   for $j = i + 1$ to $N$ do
7:     Let $C$ denote node $NodeList[j]$, and $CArgList$ denote the argument list of node C;
8:     for each argument $PArg$ in $PArgList$ do
9:       for each argument $CArg$ in $CArgList$ do
10:        if $Parg$ and $CArg$ have identical references then
11:          if $PArg.inout$ = (INOUT OR OUT) AND $CArg.inout$ = (IN OR INOUT) then
12:            Insert a RAW dependency RAW($P$, $C$);
13:          else if $PArg.inout$ = IN AND $CArg.inout$ = (INOUT OR OUT) then
14:            Insert a WAR dependency WAR($P$, $C$);
15:          else if $PArg.inout$ = (INOUT OR OUT) AND $CArg.inout$ = OUT then
16:            Insert a WAW dependency WAW($P$, $C$);
17:          end if
18:        end if
19:      end for
20:    end for
21:    Assign the appropriate rank to node $C$;
22:  end for
23:  Assign the appropriate rank to node $P$;
24: end for
```

3 Inter-Server Direct Data Transfer

One approach to inter-server data transfer via a Grid file system called Gfarm was introduced in [13]. This is similar to the Distributed Storage Infrastructure (DSI) implementation [3] in GridSolve. In GridSolve, DSI is mainly used for building external data repositories to provide large chunks of input data and output storage to tasks running on servers. Both approaches use external libraries that must be installed and configured prior to use.

In this paper, we describe our approach to direct inter-server data transfer via file staging. File staging is a service in GridSolve that moves files between two servers. Our approach uses file staging as a medium of transferring intermediate data between two servers. Specifically, intermediate results are first saved as data files, and are then staged to the target servers, on which they are restored by the tasks depending on them. This approach not only eliminates unnecessary data transport, it also protects the system from losing data, since data can be easily retrieved from locally saved files. In our approach, servers that need an intermediate result "pull" the that data from the servers that produce it. It is therefore necessary for the server that needs an intermediate result to know which server produces it. Our solution is to have the server that produces the intermediate result send a *data handle* via the client to the servers that need the result. A data handle is a small data structure that describes various aspects of an argument in GridSolve, including object type, data type, storage pattern, task name, server name, data size, file name, file path. The "pull" approach to data-movement was used because it allows a more dynamic approach to scheduling.

In GridSolve request sequencing, data handles are used as virtual pointers to intermediate results stored in special data files. Data handles are passed between two servers via the client. The recipient of a data handle, the server that needs the data referenced in the data handle, asks for the intermediate data by sending a request to the server that stores the data. Upon receiving the request, the server that stores the intermediate data sends it directly to the requester via file staging, without the client being involved.

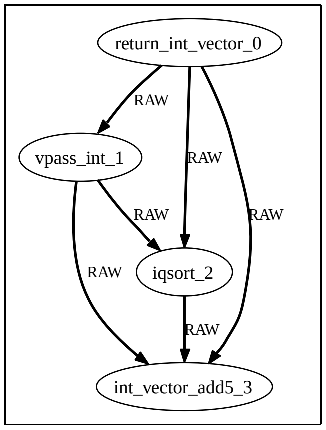


Fig. 6. An example DAG before redundant edges are removed.

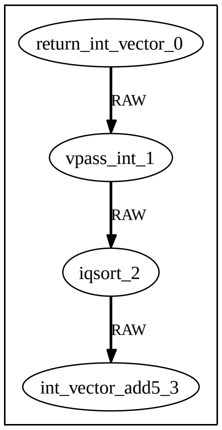


Fig. 7. The example DAG in Figure 6 after all the redundant edges are removed.

4 Workflow Scheduling and Execution

As mentioned in Section 2, after the dependency analysis, a DAG is built for a workflow and each node in the DAG is assigned an integer representing the rank of that node. The rank of a node indicates the scheduling priority of the node. The client will schedule and execute a workflow based on the rank of each node. Specifically, nodes with the same rank are independent of each other and can be scheduled to run simultaneously. Initially, the client will schedule the nodes with the highest rank (rank 0) to start executing. Notice that all the input arguments for such nodes should be available at the time of execution. The client will schedule the remaining nodes to run if and only if both the following two conditions are satisfied: 1) all the input arguments are available, and 2) all the dependencies involving the node are resolved.

A dependency is resolved if the parent node in the DAG has finished its execution. A resolved dependency is removed from the DAG. The algorithm for workflow scheduling and execution is shown in Algorithm 2. The client executes the algorithm and acts as the manager of DAG execution. The algorithm uses level-based

Algorithm 2 The algorithm for workflow scheduling and execution.

```
1: Let $N$ denote the total number of nodes in the workflow to be scheduled;
2: Let $M$ denote the number of requests that have been scheduled;
3: $M = N = 0$; CurrentSchedRank = 0;
4: repeat
5:     NodeList = NodeWithRank(CurrentSchedRank);
6:     $K$ = NumNodes(NodeList);
7:     AssignRequestsToServers(NodeList);
8:     ExecuteNodes(NodeList);
9:     WaitUntilFinished(NodeList);
10:    CurrentSchedRank = CurrentSchedRank + 1;
11:    $M = M + K$;
12: until $M = N$
```

clustering to group nodes that can be scheduled to run simultaneously. Nodes with the same rank are viewed as on the same scheduling level, and are clustered and scheduled to run simultaneously. Notice that the routine `AssignRequestToServers` assigns the nodes on the current scheduling level to the available servers. The assignment of requests to the available servers is critical to the overall performance of the execution of a DAG. In our current implementation, we use a simple round-robin strategy to assign tasks on a specific level onto the available servers.

The above round-robin scheduling algorithm is primitive and probably will be highly inefficient when the workflow is complex. A major deficiency of the algorithm is that it does not take into consideration the differences among tasks and does not really consider the mutual impact between task clustering and network communication. In addition, sometimes it will be helpful for reducing the total execution time if some tasks on a specific scheduling level are scheduled to run before other tasks on the same level. The algorithm, however, does not support this kind of out-of-order execution of tasks on the same scheduling level. This primitive algorithm will be replaced by a more advanced one in our future work.

5 GridSolve Request Sequencing API

One important design goal of GridSolve request sequencing is to ease the programming of workflow applications by providing users with a small set of API functions, presented in Table 1. The current API does

Table 1. GridSolve Request Sequencing API.

Function prototypes
`gs_sequence_begin(int)`
`gs_submit(char*, ...)`
`gs_submit_arg_stack(char*, arg_stack*)`
`gs_sequence_end(int)`

not support advanced workflow patterns such as conditional branches and loops. We are planning to investigate such advanced workflow patterns in the future to make GridSolve request sequencing a more powerful technique for workflow programming.

6 Generating Workflows via Intelligent Service Trading

We have defined an API for submitting a sequence of service requests to GridSolve so that a workflow DAG can be automatically inferred from the sequence of GridSolve requests. The DAG can be executed using GridSolve so that it preserves the sequential semantics of the original sequence of service requests, reduces data movement requirements, and allows potential parallelism to be exploited.

However, the users still needs to know about the services that are currently available, and to choose the appropriate sequence of services that will satisfy the users overall goal. In recent work, we have turned our attention to considering techniques for the generation of a workflow from a higher level description of a problem. Many complex problems can be handled by the combination of different sets of available services. The process of *service trading* [6] searches for the appropriate sequence of services to satisfy the complex problem at hand.

A service trading framework has been incorporated into the GridSolve middleware system on an experimental basis [7]. A complex problem request is described within the system using a higher-level mathematical description. The services provided by GridSolve are described using similar mathematical descriptions. For example, matrix addition (*saxpy*) and multiplication (*sgemm*) are can be view this algebraic specification as:
$Matrix\ x,\ y : daxy(x, y) = x + y$
$Scalar\ \alpha, \beta, Matrix\ x,\ y,\ z : dgemm(\alpha, x, y, \beta, z) = \alpha * x * y + \beta * z$
When a request for the higher level problem is received, the service trader component examines the available services and uses equational unification to search for and compose a sequence of available services that satisfy

the complex higher-level request. This sequence of related service requests is then executed as a distributed workflow using the request sequencing API.

As a small demonstration of the capabilities of the combination of GridSolve with service trading, some linear algebra services (e.g. level 3 BLAS and LAPACK) have been enhanced with mathematical descriptions in order to support service trading. However, the ideas behind service trading are generalizable, so any complex process that can be described with the appropriate algebraic description can be incorporated in this framework.

The ease-of-use of this approach is demonstrated from the Matlab client interface supported by GridSolve:

```
a=[1,2,3;4,5,6;7,8,9]
b=[10,20,30;40,50,60;70,80,90]
[output]=gs_call_service_trader("(a+(b*a))"),
output =
         301          362          423
         664          815          966
        1027         1268         1509
```

The service trader analyzes the input data items and their operations, discovers available services that could be composed to satisfy these operations (e.g. *daxpy*, *dgemm*), tries multiple methods of composing these services, selects an appropriate sequence of services, and submits the services requests via the GridSolve request sequencing API.

Using the service trader with GridSolve makes the users service requests more efficient by the following two factors. First, the service trader evaluates the computational cost of the various service choices for the users specific data in order to choose more efficient services. Second, the request sequencing API enables the parallelism in the sequence of requests. There is a cost to the analysis performed by the service trader, but we expect to reduce this cost in future work.

The major advantages provided by the combined framework are ease-of-use and transparency. The end user does not have to be knowledgeable in grid computing, mathematical libraries, algorithmic complexity, data dependency analysis, parallel asynchronous exaltation, fault-tolerance, or any of the details that are transparently handled by this system. The framework provides fault tolerance at various levels of failure. If a single service fails, it is transparently retried; if all copies of a service have failed, the trader can formulate the complex request as different combination of services. The user need not be aware of the exact composition of services that satisfies the request. A detailed description of the service trading implementation in GridSolve is available elsewhere [7].

7 Applications and Experiments

This section presents experiments using GridSolve request sequencing to build practical workflow applications. The first application is to implement Strassen's algorithm for matrix multiplication. As shown below, Strassen's algorithm works in a layered fashion, and there are data dependencies between adjacent layers. Thus it is natural to represent Strassen's algorithm as a workflow using GridSolve request sequencing.

The second application is to build a Montage [8] [4] workflow for creating science-grade mosaics of astronomical images. Montage is a portable toolkit for constructing custom science-grade mosaics by composing multiple astronomical images.

7.1 Experiments with Strassen's Algorithm

This subsection discusses a series of experiments with the implementation of Strassen's algorithm using GridSolve request sequencing. The servers used in the experiments are Linux boxes with Dual Intel Pentium 4 EM64T 3.4GHz processors and 2.0 GB memory. The client, the agent, and the servers are all connected via 100 Mb/s Ethernet.

Implementation Strassen's algorithm is a fast divide-and-conquer algorithm for matrix multiplication. The computational complexity of this algorithm is $O(n^{2.81})$, which is better than the $O(n^3)$ complexity of the classic implementation. Strassen's algorithm is recursive and works in a block, layered fashion, as shown by Table 2. Table 2 illustrates the outline of a single level recursion of the algorithm [12].

Table 2. The outline of a single level recursion of Strassen's algorithm.

$T_1 = A_{11} + A_{22}$	$T_2 = A_{21} + A_{22}$
$T_3 = A_{11} + A_{12}$	$T_4 = A_{21} - A_{11}$
$T_5 = A_{12} - A_{22}$	$T_6 = B_{11} + B_{22}$
$T_7 = B_{12} - B_{22}$	$T_8 = B_{21} - B_{11}$
$T_9 = B_{11} + B_{12}$	$T_{10} = B_{21} + B_{22}$
$Q_1 = T_1 \times T_6$	$Q_2 = T_2 \times B_{11}$
$Q_3 = A_{11} \times T_7$	$Q_4 = A_{22} \times T_8$
$Q_5 = T_3 \times B_{22}$	$Q_6 = T_4 \times T_9$
$Q_7 = T_5 \times T_{10}$	
$C_{11} = Q_1 + Q_4 - Q_5 + Q_7$	$C_{12} = Q_3 + Q_5$
$C_{21} = Q_2 + Q_4$	$C_{22} = Q_1 - Q_2 + Q_3 + Q_6$

As shown in Table 2, Strassen's algorithm is organized in a layered fashion, and there are data dependencies between adjacent layers. Thus it is natural to represent a single level recursion of the algorithm as a workflow and construct the algorithm using GridSolve request sequencing. It can be seen that on each layer, tasks can be performed fully in parallel, since there is no data dependency among tasks on the same layer. For instance, the seven submatrix multiplications (Q1 to Q7) can each be executed by a separate process running on a separate server.

Results and Analysis Figure 8 plots the execution time as a function of N (matrix size) of Strassen's algorithm on a single server, both with and without inter-server data transfer. This figure demonstrates the advantage of eliminating unnecessary data traffic when a single server is used. It can be seen in the figure that the computational performance with direct inter-server data transfer is consistently better than that without the feature. This figure shows the case that only one server is used. In this case, intermediate results are passed between tasks locally within the single server when direct inter-server data transfer is enabled. When multiple servers are used, intermediate results are transferred directly among servers. Considering that servers are typically connected using high-speed interconnections, the elimination of unnecessary data traffic will still be helpful in boosting the performance in the case that multiple servers are used. Figure 9 plots the execution time as a function of N of Strassen's algorithm on 4 servers, both with and without inter-server data transfer. The same conclusion that eliminating unnecessary data transport is beneficial can be obtained as in Figure 8.

It can be seen in Figure 8 and 9 that parallel execution in this case is disappointingly ineffective in improving the computational performance. This is attributed to several important reasons. As discussed above, in the case that a single server is used, intermediate results are passed between tasks locally within the single server, resulting in no real network communication. In contrast, when multiple servers are used, some intermediate results have to be passed among tasks running on different servers, resulting in real network transfer of large chunks of data. Considering that the client, the agent, and the servers are all connected via 100 Mb/s Ethernet, the overhead of network traffic can be relatively large. Therefore, the effect of parallel execution in improving the overall performance is largely offset by the overhead of additional network traffic. In addition, the overhead within the GridSolve system further reduces the weight of the time purely spent on computation in the total execution time, making it even less effective to try to reduce the computation time by parallel execution. Another important reason is that the primitive algorithm for DAG scheduling and execution is highly inefficient for complex workflows, as discussed in Section 4. The above result indicates

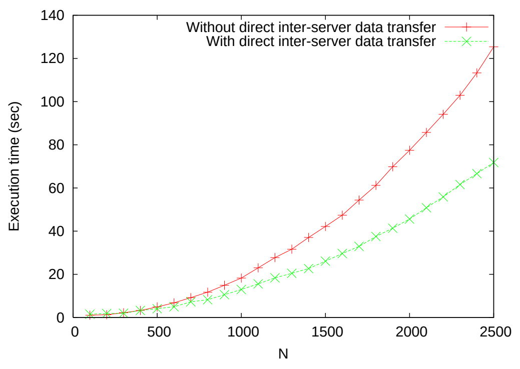


Fig. 8. The execution time of Strassen's algorithm as a function of N on a single server, both with and without inter-server data transfer.

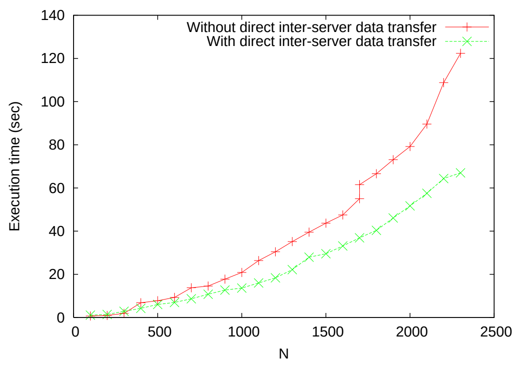


Fig. 9. The execution time of Strassen's algorithm as a function of N on four servers, both with and without inter-server data transfer.

that GridSolve request sequencing is not an appropriate technique for implementing fine-grained parallel applications, since the overhead of network communication and remote service invocation in GridSolve can easily offset the performance gain of parallel execution.

7.2 Experiments with Montage

Montage is program for building highly detailed mosaics of astronomical images. It uses three steps to build a mosaic:

Re-projection of input images this step re-projects input images to a common spatial scale and coordinate system.

Modeling of background radiation in images this step rectifies the re-projected images to a common flux scale and background level.

Co-addition : this step co-adds re-projected, background-rectified images into a final mosaic.

Each step consists of a number of tasks that are performed by the corresponding Montage modules. There are dependencies both between adjacent steps and among tasks in each step.

We present a series of experiments with a simple application of the Montage toolkit, introduced in the following subsection. Unlike Strassen's algorithm for matrix multiplication, this is essentially an image processing application and is more coarse-grained. The servers used in the experiments are Linux boxes with Dual Intel Pentium 4 EM64T 3.4GHz processors and 2.0 GB memory. The client, the agent, and the servers are all connected via 100 Mb/s Ethernet.

A Simple Montage Application We use the simple Montage application introduced in "Montage Tutorial: m101 Mosaic" [9], which is a step-by-step tutorial on how to use the Montage toolkit to create a mosaic of 10 2MASS [1] Atlas images. This simple application generates both background-matched and uncorrected versions of the mosaic [9]. The step-by-step instruction in the tutorial can be easily converted to a simple workflow, which is illustrated by the left graph in Figure 12. The rest of this section refers to this workflow as the naive workflow. The detailed description of each Montage module used in the application and the workflow can be found in the documentation section of [8].

The output of the naive workflow, both the uncorrected and background-matched versions of the mosaic, are given in Figure 10 and 11, respectively. It can be seen that the background-matched version of the mosaic has a much better quality than the uncorrected version.

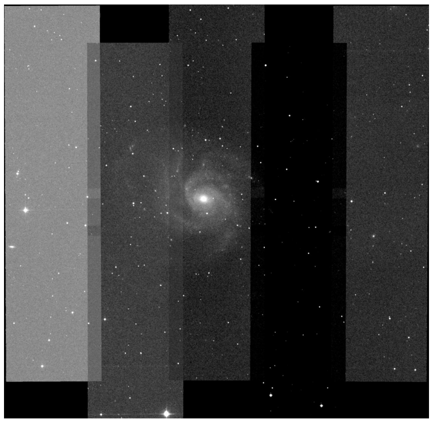


Fig. 10. The uncorrected version of the mosaic.

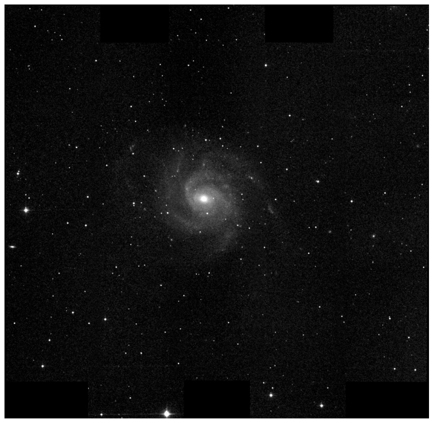
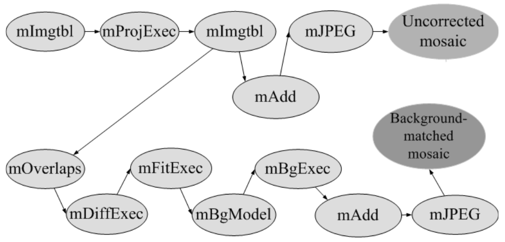


Fig. 11. The background-matched version of the mosaic.



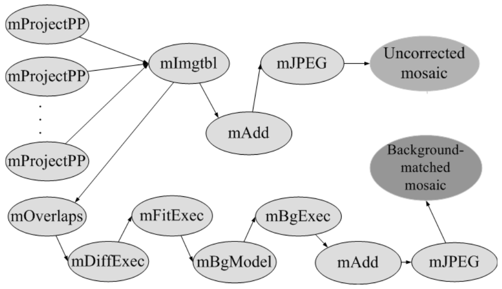


Fig. 12. The naive (left) and modified (right) workflows built for the simple Montage application. In each workflow, the left branch generates the uncorrected version of the mosaic, whereas the right branch generates the background-matched version of the mosaic. Both branches are highlighted by the wrapping boxes.

Parallelization of Image Re-projection The execution time of the naive workflow on a single server is approximately 90 to 95 seconds, as shown below. The most time-consuming operation in the naive workflow is `mProjExec`, which is a batch operation that re-projects a set of images to a common spatial scale and coordinate system, by calling `mProjectPP` for each image internally. `mProjectPP` performs a plane-to-plane projection on the single input image, and outputs the result as another image. It is obvious that the calls to `mProjectPP` are serialized in `mProjExec`. Thus, an obvious way to improve the performance of the naive workflow is replacing the single `mProjExec` operation with a set of independent `mProjectPP` operations and parallelize the execution of these independent image re-projection operations. The workflow with this modification is illustrated by the right graph in Figure 12. The rest of this section refers to this workflow as the modified workflow.

Results and Analysis Figure 13 shows the execution time (sorted) on a single server of the best 10 of 20 runs of both the naive and modified workflows. It can be seen that the performance of the modified workflow is significantly better than that of the naive workflow. The reason is that the single server has two processors as mentioned above, and therefore can execute two `mProjectPP` operations simultaneously. This result demonstrates the benefit of parallelizing the time-consuming image re-projection operation by replacing the single `mProjExec` with a set of independent `mProjectPP` operations. It is still interesting to see whether using more than one server can further speed up the execution. This is investigated by the following

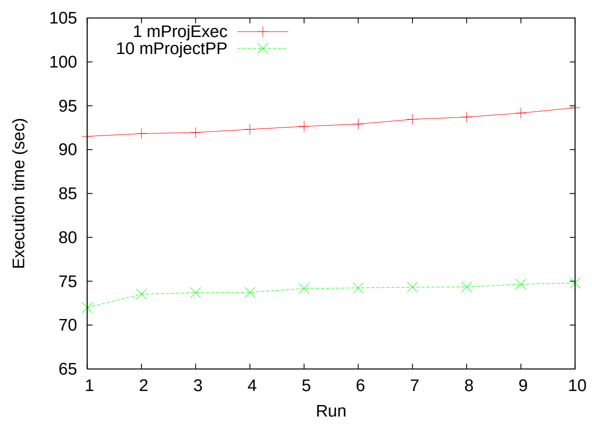


Fig. 13. The execution time (sorted) on a single server of the best 10 of 20 runs of both the naive and modified workflows.

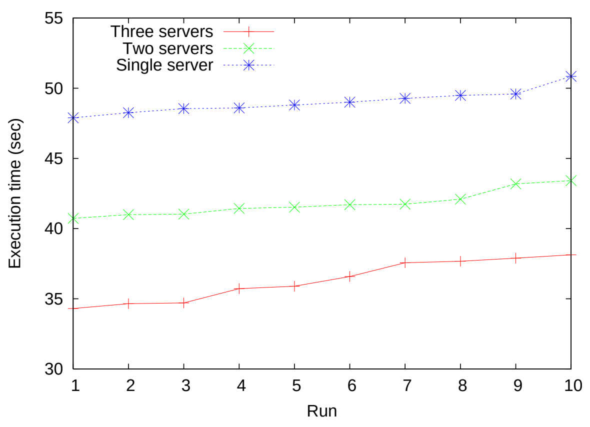


Fig. 14. The execution time (sorted) of the best 10 of 20 runs of the left branch of the modified workflow on different numbers of servers (1, 2, and 3).

experiment. The next experiment is based on a smaller workflow, which is the left branch of the modified workflow, i.e., the smaller branch that produces the uncorrected version of the mosaic. The reason for using a smaller workflow is that we want to minimize the influence of the fluctuating execution time of the right branch on the overall execution time. The expectation that using more than one server can further speed up the execution is demonstrated by Figure 14. The figure shows the execution time (sorted) of the best 10 out of 20 runs of the left branch of the modified workflow on different numbers of servers (1, 2, and 3). It is not surprising to see in the figure that the performance is better as more servers are used to increase the degree of parallelization.

8 Conclusions and Future Work

GridSolve request sequencing is a technique developed for users to build workflow applications for efficient problem solving in GridSolve. GridSolve request sequencing completely eliminates unnecessary data transfer during the execution of tasks. In addition, it is capable of exploiting the potential parallelism among tasks in a workflow. The experiments discussed in the paper promisingly demonstrate the benefit of eliminating unnecessary data transfer taking advantage of the potential parallelism. An important feature of GridSolve request sequencing is that the analysis of dependencies among tasks in a workflow is fully automated; users are not required to manually specify the dependencies among tasks in a workflow. These features plus the easy-to-use API make GridSolve request sequencing a powerful tool for building workflow applications for efficient parallel problem solving in GridSolve.

The service trading component integrated on top of the GridSolve request sequencing infrastructure provides additional advantages to a user in terms of ease-of-use and transparency. The combination of request sequencing and service trading provides fault tolerance at various levels of failure. If a single service fails, it is transparently retried by GridSolve; if all copies of a service have failed, the trader can formulate the complex request as different combination of services. The user need not be aware of the exact composition of services that satisfies the request.

In future work the request sequencing scheduler will be improved to make more intelligent resource management decisions rather than using the simple level based scheduler. The service trader will be optimized to reduce the associated with generating DAGs from complex service requests.

A integrating theme in GridSolve research is ease-of-use and our goal in this work is to make GridSolve an easy-to-use yet powerful tool to enable an end user to write workflow programs. The user should not have to be knowledgeable in grid computing, mathematical libraries, algorithmic complexity, data dependency analysis, parallel asynchronous exaltation, fault-tolerance, or any of the details that are transparently handled by this system.

9 Acknowledgments

This research made use of Montage, funded by the National Aeronautics and Space Administration's Earth Science Technology Office, Computation Technologies Project, under Cooperative Agreement Number NCC5-626 between NASA and the California Institute of Technology. Montage is maintained by the NASA/IPAC Infrared Science Archive.

References

1. The 2MASS Project. http://www.ipac.caltech.edu/2mass.
2. Dorian C. Arnold, Dieter Bachmann, and Jack Dongarra. Request Sequencing: Optimizing Communication for the Grid. *Lecture Notes in Computer Science*, 1900:1213–1222, 2001.
3. Micah Beck and Terry Moore. The Internet2 Distributed Storage Infrastructure project: an architecture for Internet content channels. *Computer Networks and ISDN Systems*, 30(22–23):2141–2148, 1998.
4. G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, and M.-H. Su. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In P. J. Quinn and A. Bridger, editors, *Optimizing Scientific Return for Astronomy through Information Technologies. Edited by Quinn, Peter J.; Bridger, Alan. Proceedings of the SPIE, Volume 5493, pp. 221-232 (2004).*, volume 5493 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 221–232, September 2004.
5. The GridSolve Project. http://icl.cs.utk.edu/gridsolve/.
6. Aurélie Hurault and Marc Pantel. Mathematical service trading based on equational matching. *Electron. Notes Theor. Comput. Sci.*, 151(1):161–177, 2006.
7. Aurélie Hurault and Asim YarKhan. Intelligent Service Trading for Distributed Network Services in GridSolve. *High Performance Computing for Computational Science - VECPAR 2010: 9th International Meeting*, 2010. Accepted.
8. The Montage Project. http://montage.ipac.caltech.edu/.
9. Montage Tutorial: m101 Mosaic. http://montage.ipac.caltech.edu/docs/m101tutorial.html.
10. The NetSolve Project. http://icl.cs.utk.edu/netsolve/.
11. Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 274–278, London, UK, 2002. Springer-Verlag.
12. Fengguang Song, Jack Dongarra, and Shirley Moore. Experiments with Strassen's Algorithm: from Sequential to Parallel. In *Parallel and Distributed Computing and Systems 2006 (PDCS06)*, Dallas, Texas, 2006.
13. Yusuke Tanimura, Hidemoto Nakada, Yoshio Tanaka, and Satoshi Sekiguchi. Design and Implementation of Distributed Task Sequencing on GridRPC. In *CIT '06: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, page 67, Washington, DC, USA, 2006. IEEE Computer Society.