Intelligent Service Trading and Brokering for Distributed Network Services in GridSolve

Aurlie Hurault[1] and Asim YarKhan[2]

[1] Institut de Recherche en Informatique de Toulouse - UMR 5505
University of Toulouse, France
hurault@enseeiht.fr
[2] Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN 37996
yarkhan@eecs.utk.edu

Abstract. One of the great benefits of computational grids is to provide access to a wide range of scientific software and a variety of different computational resources. It is then possible to choose from this large variety of available resources the one that solves a given problem, and even to combine these resources in order to obtain the best solution.
Grid service trading (searching for the best combination of software and execution platform according to the user requirements) is thus a crucial issue. Trading relies on the description of available services and computers, on the current state of the grid, and on the user requirements. Given the large amount of services that may be deployed over a Grid, this description cannot be reduced to a simple service name.
In this paper, a sophisticated service specification approach similar to algebraic data types is combined with a grid middleware. This leads to a transparent solution for users: they give a mathematical expression to be solved, and the appropriate grid services will be transparently located, composed and executed on their behalf.

1 Introduction

Grid computing and distributed computing projects have been very effective in exposing large collections of services and computational resources to users. But users still need to handle all the difficulties involved in finding and composing the appropriate resources to solve their problem. In Figure 1 we express the general problem of matching user service requests to the appropriate resources. On the right side of the figure, there is a set of services running on some computational resources, and on the left side there are end users that want to use these services to solve their problems. The goal is to find the services or combination of services that can solve the users problem accurately and efficiently, to execute these services and return the result to the user.

A corollary part of this problem is providing the user with the appropriate syntax to express their needs. This syntax has to be sufficiently precise to find relevant solutions, and it should also be easy for the user to express complex problems.

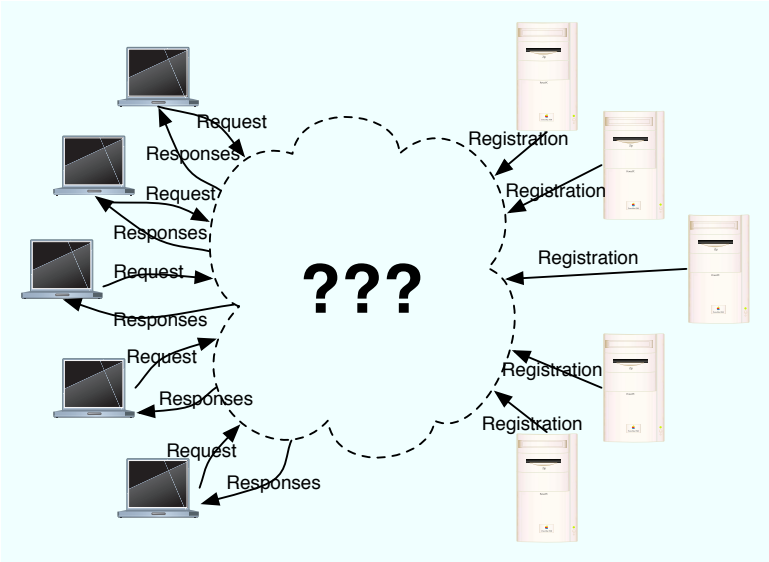


Fig. 1. A simplified overview of the general problem of matching service requests to available resources

Our solution (see Figure 2) combines a service trader and a GridRPC middleware. In this scenario, the end user asks the GridSolve [DLSY08,YSS+06] GridRPC middleware to solve some (possibly complex) problem request. The service trader uses its knowledge of the available services and matches the problem request with a service or a combination of services. GridSolve is then used to execute the services and the final result is returned to the user.

2 Service Trading

A service trader acts on the behalf of a user to find a combination of the available computational services that will handle the users request. The details of the service trader used in this work are described in [HDP09]. For this research, improvements have been made in the way it searches for solutions to the users request. The computational complexity of the available services is used to select less computationally expensive solutions from the available services. This leads to an improvement in the execution time for the users request, since, in practical situations, we can prune the more computationally complex solutions.

2.1 Inputs for the service trader

The trader needs a description of the **application domain** using an algebraic specification. To this goal we use an order-sorted signature $(S,\leq,\Sigma)$ where:

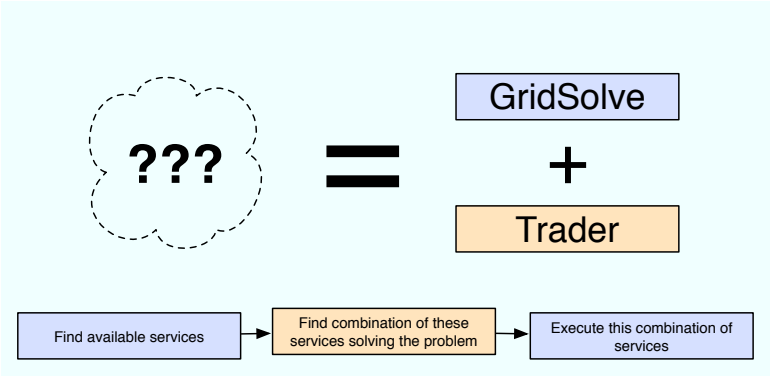


Fig. 2. Our solution takes complex user problems and using a service trader matches them to grid services exposed via GridSolve

S is a set of *sorts* (usually called *types* in programming languages, both terms will be used in this discussion);
$\leq$ is an order to express the subsorting (subtyping) link between sorts;
Σ is a set of symbols standing for constants and sorted (typed) functions.

For a complete definition see [GM92]. The operators used may be overloaded and some extra equations $\mathcal{E}$ are added to describe the properties of the operators (e.g., commutativity, associativity, neutral element). For example, a partial representation for linear algebra over scalars and matrices is: $S = \{Scalar, Matrix\}$
$\Sigma = \{$
$0, I : Matrix$
$0, 1 : Scalar$
$+ : Matrix \times Matrix \rightarrow Matrix$
$+ : Scalar \times Scalar \rightarrow Scalar$
$* : Matrix \times Matrix \rightarrow Matrix$
$* : Scalar \times Matrix \rightarrow Matrix$
$* : Scalar \times Scalar \rightarrow Scalar$
$\}$
$\mathcal{E} = \{$
$Matrix\ x, Matrix\ y : x + y = y + x$
$Matrix\ x : 1 * x = x$
$Matrix\ x : 0 * x = 0$
$Matrix\ x : I * x = x$
$\}$
The $Matrix$ types can have lots of subtypes to describe and handle different matrix properties, for example, symmetric, dense, space, triangular and band.

The **computational services** are described as terms in an order-sorted signature. This leads to a really natural description, in particular in mathematical domains, since the notations are very similar. For example, the BLAS (Basic

Linear Algebra Subroutines) [BDD+02] *saxpy* (addition) and *sgemm* (multiplication) functions can be described (in a simplified way) in the algebraic notation as:

$$Scalar\ \alpha, Matrix\ x,\ y : saxpy(\alpha, x, y) = \alpha * x + y$$
$$Scalar\ \alpha, \beta, Matrix\ x,\ y,\ z : sgemm(\alpha, x, y, \beta, z) = \alpha * x * y + \beta * z$$

A user **request** is also specified in this algebraic notation. For example a request to add three matrices would simply be expressed as:

$$Matrix\ a,\ b,\ c : a + b + a$$

The service trader is generic and the application domain can include anything that can be described using an algebraic specification. We have performed additional work with some optimization libraries [Hur06]. The main types in the optimization domain are functions and constraints, and the elements manipulated by these functions and constraints (e.g., `Real`, `Matrix`). The operators are minimization and maximization operators, the function description ($\rightarrow$), and the operators for constraints ($\leq$, &, ...) and the operators for the manipulated elements (*, +, ...). Equations are used to express the constraints on the optimization.

The optimization libraries we have considered provided the following functionality: $\min\limits_{x} f(x)$, $\min\limits_{x} c^T x$, $\min\limits_{x} \frac{1}{2}\|Cx - d\|_2^2$ and $\min\limits_{x} \frac{1}{2}x^T Hx + f^T x$. With or without the constraints: $Ax \leq b$, $Ax = b$ and $l \leq x \leq u$. We are particularly interested in the Matlab optimization toolbox[3] and in the E04 package of the NAG[4] library.

2.2 The trader output

The trader generates a list of services and combination of services that satisfy the request. For example, given the linear algebra domain and *saxpy* and *sgemm* services described earlier, for the user request of $a + b + c$, the possible solutions satisfy the request include:

$saxpy(1, a, saxpy(1, b, c))$
$saxpy(1, a, sgemm(1, I, b, 1, c))$
$sgemm(1, I, a, 1, saxpy(1, b, c))$,

If the *saxpy* function is less computationally expensive than the *sgemm* function (which is true if for the BLAS functions), then the solution

$saxpy(1, a, saxpy(1, b, c))$

[3] `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/optim.shtml`
[4] `http://www.csc.fi/cschelp/sovellukset/math/nag/NAGdoc/fl/html/E04_fl19.html`

will be only solution returned by the trader, since the other possible solutions ($saxpy(1, a, sgemm(1, I, b, 1, c))$, $sgemm(1, I, a, 1, saxpy(1, b, c))$, ...) are more computationally expensive. To this end, the complexity of the services are input to the trader as well as the size of the matrices. Since before finishing the comparison we do not know what the parameters will be, we can't compute the exact cost. We use a medium size of the data to have an approximate cost and decide if it is interesting to do the comparison, or if we currently have better answers. We might lose some interesting responses, but by not doing some comparisons, we will win in the search time taken the trader.

For example, we are trying to find solutions to the request "$A * B$", where A and B are matrices. To evaluate if it is interesting to compare another solution with the *sgemm* service, we will approximate the cost of the *sgemm* service. To do that we use an estimated medium size for the matrices, since the trader does not know the sizes we will give when calling the *sgemm* service. This cost can be compared with the cost other known solutions.

The BLAS *saxpy* function has additional parameters not related to function signature (for example, the sizes of the matrices). These need to be available to do data transfer and execute the services. These parameters are considered after the analysis of the functional signature . The trader focuses on the functional aspect and considers the other parameters later. For example, considering the BLAS *strsm* (triangular solver) routine:

> `STRSM (SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)`
> `M` specifies the number of rows of the parameter `B` (matrix);
> `UPLO` specifies if the matrix `A` is an upper or a lower triangular matrix;
> `SIDE` specifies if the problem solved is $op(A) * X = \alpha * B$ or $X * op(A) = \alpha * B$.

We add different mechanisms in the algorithm that allow us to find the value for those parameters:

- When a procedure implements several functionalities, we introduce the "switch / case" functionality. The procedure will generate different services, each with one parameter set for a given value. In our example, one service with $SIDE$ set to 'left' and one service with $SIDE$ set to 'right' are generated.
- When some parameters depend on other, we describe this dependence, and when the value of the first parameter is known, the value of the second one is computed. In our example, when the exact type of A is known, $UPLO$ will be set to 'u' or 'l'.
- When some parameters specify the properties of other parameters or for default value, a term is assigned to a parameter, using only the constant of the domain and of the request. In our example, M will be assigned to $m(B)$ (number of rows of B). When B will be known, $m(B)$ will also be known and the value of M will be fixed.

The full service description of *saxpy* is:

```
SAXPY( n,alpha,x,incx,y,incy ):
```

```
y <- alpha * x + y
n = m(x) * n(x) || m(y) * n(y);
incx = default 1;
incy = default 1;
```

Where $m(x)$ is the number of rows of the x matrices and $n(x)$ the number of columns. Some default values are given for *incx* and *incy*.

2.3 Inside the trader

As explained in [HDP09], the trader is based on equational unification and more particularly on the work of Gallier and Snyder [GS89]. It has a type system adapted to overloaded functions with subtyping, based on the $\lambda\&-$calculus defined by Castagna [CGL92]. The algorithm of Gallier and Snyder has been modified by adding an amount of energy to ensure that the computation will end. This amount is composed by the depth of combination and the number of equations that can be applied.

Since we are trying to compute an efficient combination of services, we first check the services that are less complex. To this end, we use the mathematical expression of the computational complexity of the services (as provided by GridSolve). The service complexity (e.g., $O(2n^2)$) uses the sizes of its parameters (e.g.,n) to estimate the execution time. We can thus compare the execution cost of the various service choices taking into account the size of the parameters. This has two major benefits:

- We find some efficient solutions;
- We prune branches on search tree, since we do not look at more expensive solutions.

3 Overview of GridSolve and GridRPC

The purpose of GridSolve is to create the middleware necessary to provide a seamless bridge between computational scientists using desktop systems and the rich supply of services supported by the emerging Grid architecture. The goal is that the users of desktop systems can easily access and reap the benefits (in terms of shared processing, storage, software, data resources, etc.) of using grids. GridSolve is designed to enable a broad community of scientists, engineers, research professionals and students to easily draw on the vast, shared resources of the Grid. Working with the powerful and flexible tool set provided by their familiar desktop computing environment, they can tap into the power of the Grid for unique or exceptional resource needs.

GridSolve is a client-agent-server (or *brokered RPC*) system which provides remote access to hardware and software resources through a variety of client interfaces.

The system consists of three entities, as illustrated in Figure 3.

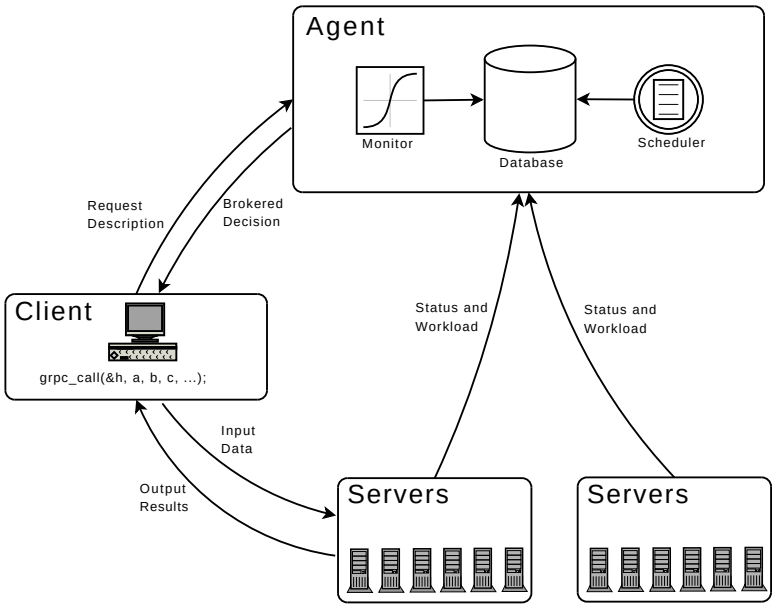


Fig. 3. Architectural overview of GridSolve

- The *Client*, which needs to execute some remote procedure call. In addition to C and Fortran programs, the GridSolve client may be an interactive problem solving environment such as Matlab, Octave, or IDL (Interactive Data Language).
- The *Server* executes functions on behalf of the clients. The server hardware can range in complexity from a uniprocessor to a MPP system and the functions executed by the server can be arbitrarily complex. Server administrators can add straightforwardly their own function services without affecting the rest of the GridSolve system.
- The *Agent* is the focal point of the GridSolve system. It maintains a list of all available servers and performs resource selection for client requests as well as ensuring load balancing of the servers.

In practice, from the user's perspective the mechanisms employed by GridSolve make the remote procedure call fairly transparent. However, behind the scenes, a typical call to GridSolve involves several steps as follows:

1. The client asks the agent for appropriate servers that can execute the desired function.
2. The agent returns a list of available servers, ranked in order of suitability.
3. The client attempts to contact a server from the list, starting with the first and moving down through the list. The client then sends the input data to the server.

4. Finally the server executes the function on behalf of the client and returns the results.

In addition to providing the middleware necessary to perform the brokered remote procedure call, GridSolve aims to provide mechanisms to interface with other existing Grid services. This can be done by having a client that knows how to communicate with various Grid services or by having servers that act as proxies to those Grid services. GridSolve provides some support for the proxy server approach, while the client-side approach would be supported by the emerging GridRPC standard API [SHM+02].

4 Integration of service trading into GridSolve

As explained in the introduction, the goal is to facilitate a user job by making transparent calls to the grid, so that the user does not have to be knowledgeable about the available Grid resources or services.

To this aim, we proceed in four steps:

1. GridSolve provides information about available services.
2. The trader finds the combination of services that solves the user request.
3. The output of the trader is analyzed and the services are called.
4. The response is transferred back to the user.

4.1 Generating the inputs of the trader

GridSolve is responsible for providing information about the available services to the service trader. The service trader requires some additional semantic information that has to be added to the standard service descriptions provided by GridSolve.

This additional information is the mathematical expression of the functions computed by the service, and some information about the non-functional parameters. For example, for the BLAS *dgemm* and the *dsymm* functions, it will be:

```
SUBROUTINE dgemm
APPLICATION_DOMAIN="LinearAlgebra"
TRADER_DESCRIPTION="
c <-((alpha*((op transa a)*(op transb b)))+(beta*c)) ;
value m = (m c) || (m (op transa a)) ;
..."

SUBROUTINE dsymm
APPLICATION_DOMAIN="LinearAlgebra"
PARAMETERS_PROPERTIES = "a symmetric"
TRADER_DESCRIPTION="
c <- if (side='l') then ((alpha*(a*b))+(beta*c))
```

```
     if (side='r') then ((alpha*(b*a))+(beta*c)) ;
value m = (m c) ;
...
if ( a instanceof UpTriInvMatrix ) then ( uplo = 'u' );
..."
```

4.2 Discover the combination of services

Given a user request in the GridSolve client, GridSolve calls the service trader, which processes the request and returns a file containing a sequence of services calls that will satisfy the users request. For example, for the request $a + b + c$:

```
def, res2, copy c
def, res1, copy a
call, saxpy, m(b)*n(b), 1.0, copy b, 1, res2, 1
call, saxpy, m(res2)*n(res2), 1.0, res2, 1, res1, 1
```

The GridSolve runtime can then transform this sequence of requests into a workflow DAG [LDSY08] to improve the performance and be able to run different parts in parallel when possible.

4.3 Call the services

To do the calls, the GridSolve client parses the output file. When it finds a "def", a new local variable is created and set to the second pointer. When it finds a "copy", a copy of the user data is created. And when it finds a "call", the GridRPC call is done with the parameters that follow. To do the call, some additional information is needed: type of data, sizes of data, pointer to the data. In the C and Matlab interfaces, those parameters are discovered in different ways:

- Information provided by the user in the C interface.
- Information found via Matlab data querying mechanisms in the Matlab interface.

In fact, a first analysis will create the temporary variables needed for the computation. Then a second analysis is done in order to make the GridRPC call. The calls are executed using DAG interface of GridSolve in order to enable any the parallelism in the sequence of calls.

4.4 The Service Trader C API

We have added two functions to the GridSolve C API for the service trader.

```
int gs_call_service_trader(char *req,... );
int gs_call_service_trader_stack(char * req, grpc_arg_stack *argsStack);
```

The first parameter is a string that expresses the request in a simple analytical or mathematical expression . The other arguments are pointers to the actual data, and information about the variable name and size. An example call using the service trader interface:

```
float *a = malloc (sizeof(float)*ma*na);
float *b = malloc (sizeof(float)*mb*nb);
...
gs_call_service_trader("(a+(b+a))","a",a,ma,na,...);
```

4.5 The Matlab interface

The Matlab client interface is substantially simpler than the C interface since a variety of information about the variable names and sizes can be obtained by querying Matlab internal data representations. An example service trader call using the Matlab interface:

```
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 =
   12.  24.  36.
   48.  60.  72.
   84.  96.  108.
```

The output variable is integrated back into the Matlab workspace and can be used for later computation in Matlab.

5 Experiments

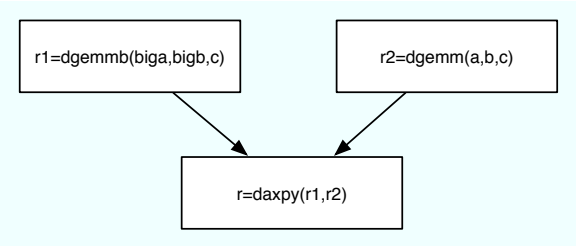
In these experiments we show the creation of a DAG from a users problem request, and the selection of the appropriate services using computational complexities to guide the selection.

For our small test experiment, the relevant Grid services are the previously described BLAS services (*saxpy*, *daxpy*, *sgemm* and *dgemm*) and a service implementing the Strassen-Winograd algorithm for matrix-multiplication (*sgemmb*). This variant has a lower computational complexity (approximately $O(n^{2.8})$) than standard matrix-multiplication algorithms ($O(n^3)$), but it only becomes efficient if the matrix size is sufficiently large. With the complexity information provided as an input to the trader, it will be able to choose the best algorithm as a function of the size of the matrices. The "exact" complexity information given to the trader is $7 * pow(n, log2(7)) + 3 * m * k - 6 * n * n$ for the Strassen-Winograd algorithm and $2.0 * m * n * k + 2.0 * m * k$ for the classic algorithm.

As an example, we want to compute $(((a * b) + c) + ((biga * bigb) + c))$, where a, b and c are $3x3$ matrices, *biga* is a 3x3000 matrix and *bigb* is a 3000x3 matrix. Using the Matlab interface:

```
gs_call_service_trader("(((a*b)+c)+((biga*bigb)+c)))")
```

The trader generates a sequence of 3 service calls that will solve this problem. GridSolve then creates a DAG based on the data dependencies between these calls and calls the services.



These events are totally transparent to the user, who simply needs to present the desired mathematical expression.

In this experiment, we observe that based on the size of the matrices, the service trader uses the complexity information to select the more efficient multiplication algorithm: the Strassen-Winograd matrix multiplication *dgemmb* for the bigger matrices and standard *dgemm* for the smallest ones.

6 Summary

In this paper we developed a combination of a service trader and a grid middleware system to enable a "novice" user to gain access a remote library, without knowing about grid computing or about the available library and services.

The key point for the end user is the transparency of all the details involved in this process. The fact that the services are evaluated at each call makes the solution more tolerant to a service crash. If a service disappear, the trader will find some other solution to the users problem with a different combination of services. For example, if the *dgemmb* service is not available, the service trader will be able to replace it by *dgemm*. If the *daxpy* is not available, the trader will be able to find a solution with *dgemm*. The user doesn't have to be aware of all the alternative services that may satisfy their request.

The execution of the users request is made more efficient by two factors; firstly, the service trader evaluates the computational complexity of the available services on the users specific data, and secondly, the GridSolve DAG execution system enables any parallelism in the execution of the services. There is a cost for the analysis done by the service trader in this current prototype, however we expect that this cost can be reduced in the future. Moreover, the major advantage provided by the work developed here is ease-of-use. The end user does not have to be knowledgeable in grid computing, mathematical libraries, algorithmic complexity, data dependency analysis, fault-tolerance, or any of the details that are transparently handled by this system.

References

[BDD+02] Susan Blackford, Jim Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Mike Heroux, Linda Kaufman, A. Lumsdaine, Andrew Petitet, Roldan Pozo, Karin Remington, and Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[CGL92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, volume 5, pages 182–192, 1992.

[DLSY08] J. Dongarra, Y. Li, K. Seymour, and A. YarKhan. Users' Guide to GridSolve V0.19. Innovative Computing Laboratory. Technical Report, University of Tennessee, Knoxville, TN, June 2008.

[GM92] Joseph Goguen and Jos Meseguer. Order-sorted algebra i: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[GS89] Jean Gallier and Wayne Snyder. Complete Sets of Transformations for General E-Unification. *Theor. Comput. Sci.*, 67(2-3):203–260, 1989.

[HDP09] Aurlie Hurault, Michel Dayd, and Marc Pantel. Advanced Service Trading for Scientific Computing over the Grid. *Journal of Supercomputing*, 49(1):64–83, juillet 2009.

[Hur06] A. Hurault. *Courtage smantique de services de calcul*. PhD thesis, INPT, Toulouse, Dcembre 2006.

[LDSY08] Yinan Li, Jack Dongarra, Keith Seymour, and Asim YarKhan. Request Sequencing: Enabling Workflow for Efficient Problem Solving in GridSolve. *International Conference on Grid and Cooperative Computing (GCC 2008)*, pages 449–458, Oct 2008.

[SHM+02] K. Seymour, N. Hakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *GRID 2002*, pages 274–278, 2002.

[YSS+06] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent Developments in GridSolve. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1):131–141, 2006.