Performance Analysis of a Multi-Physics Simulation System Based on Web Services*

Paul Stodghill, Rob Cronin, Keshav Pingali
Dept. of Computer Science
Cornell University

Gerd Heber
Cornell Theory Center
Cornell University

Abstract

The ongoing convergence of Grid computing and Web Services has motivated a number of research groups to study the use of SOAP-based Web Services for scientific computing. These studies have exposed various performance problems in using SOAP-based communication, and a number of suggested extensions to the SOAP standard and/or sophisticated implementation strategies have been proposed to eliminate these bottlenecks.

In this paper, we describe our experience in building a system based on Web Services for simulating a multi-physics, coupled fluid/thermal/mechanical fracture problem. The system is organized as a collection of geographically-distributed software components in which each component provides a Web Service, and uses standard SOAP-based Web Service protocols to interact with other components. There are a number of advantages to organizing a system in this way, which we discuss.

We have analyzed the performance of our system and found that the overhead for using SOAP-based Web Services is small. Our results suggest that the previously identified potential bottlenecks may not be major issues in practice, and that even a standards-compliant implementation like ours can delivery good performance provided Web Services are used judiciously.

1 Introduction

The Adaptive Software Project (ASP)[1] is a multi-institutional, multi-disciplinary computational science project that is studying adaptivity in computational science applications. Our project includes members from a number of colleges and universities and from a number of disciplines, ranging from civil engineering and physics, to numerical analysis and computational geometry, to restructuring compilers and distributed systems. One of our goals is to develop state of the art simulation systems for coupled multi-physics problems with complicated and evolving geometries.

We feel that building adaptive systems within a project like ours requires a design that is based upon distributed software components.

Why components? Componentization is a necessary (but not sufficient) condition for interchanging modules within an adaptive system. Consider, for example, an application that switches from one algorithmic technique to another: if the two techniques do not have clearly defined interfaces or use similar parameter (or data) types, then dynamically switching between them would be impossible.

Why distributed? Our project members use many different architectures and operating systems, and it would be a tremendous burden if every developer had to port their code to every other platform. Ideally, a component could be deployed on just one platform and invoked remotely by project partners. In other words, our components should be "write once, run *from* anywhere". In addition, some adaptive systems (e.g, DDDAS ([11])) include data collection devices (e.g., VLA radio telescopes, sensor arrays) that are necessarily physically separated.

To summarize, our applications require that we build a distributed component-based system whose configuration evolves dynamically and whose implementation and execution spans institutions. In the parlance of Grid computing, we need to build a *virtual organization* ([14]). To build such a virtual organization, we need a system infrastructure with certain functionality, much of which is specified in the Open Grid Services Architecture ([13]). In this paper, we will focus on one aspect of this infrastructure, namely the programming model and its implementation.

Our current programming model is based upon the concept of the *remote procedure call (RPC)*. In our current sys-

*This research is partially supported by NSF grants EIA-9726388, EIA-9972853, and ACIR-0085969.

[1] Additional information about the ASP project can be found at `http://www.asp.cornell.edu/`.

tem, a client implements the simulation workflow by using the RPC mechanism to execute components on remote servers. This is a baseline; in a more advanced system, parts of the workflow might be executed by agents or other specialized servers.

There are at least two systems available that implement RPC for virtual organizations, Ninf ([22]) and NetSolve ([1]). The Global Grid Forum (GGF) GridRPC Working Group is attempting to define standard API's for these systems ([25]), but the protocols that these systems use for communicating over the Internet are not standardized.

The lack of standard protocols presents us with two difficulties. The first is philosophical: we have argued that an adaptive system like ours should be built from distributed components with standardized interfaces. If we then use a proprietary mechanism for invoking these standardized components, we will lose many of the benefits of standardization. The second difficulty is practical: in order not to force our project members and users to migrate to new client and programming tools, we need to use standard protocols that are supported by a wide array of programming languages and frameworks.

By our evaluation, one set of Internet protocols meets our needs, namely Web Services ([30]). Many businesses need to form collaborations that span company boundaries, and they need a software infrastructure that will support their efforts and not force them to use a small set of proprietary client and programming tools. As a result of this, the W3C, Oasis and other organizations have started to define the necessary Web Services protocols.

Of course, these protocols have been designed primarily with business applications in mind, so one question is whether they are suitable for scientific applications like ours. Previous papers ([9],[26]) have looked at the use of XML and SOAP, two of the fundamental Web Services protocols, for this class of application. Their conclusions are that their use in scientific applications can impose a large performance penalty and recommend several sophisticated implementation strategies and changes to these protocols to account for this.

These two papers measured the overhead of using Web Services for benchmarks and small parts of applications. In this paper, we offer an end-to-end performance evaluation of our Web Services-based system. To the best of our knowledge, this is the first performance evaluation of an entire state-of-the-art scientific application built upon Web Services.

Section 2 gives a high-level description of our system and the problem that it simulates. Section 3 describes the Web Services infrastructure that supports our system. In Section 4, we present preliminary performance measurements and in Section 5, we compare our performance results with the previous results. In Section 6 we discuss other related work. In Section 7, we draw some conclusions about the success of our efforts and about how to organize distributed simulation systems and discuss our future work.

2 Application Description

In this section, we briefly describe the physical problem that we are simulating and give a high-level description of our system. [7] contains a more detailed description of the physical problem and the components of our system.

The applications engineers on our research team work in computational fluid mechanics and solid mechanics, so we decided to tackle a problem involving high Reynolds number, chemically reacting gas flow coupled with linear elastic fracture mechanics.

The geometry of our problem is an idealized segment of a rocket engine modeled after actual NASA experimental spacecraft hardware. The object is a curved, cooled pipe segment that transmits a chemically reacting, high-pressure, high-velocity gas through the inner, large diameter passage, and cooling fluid through the outer array of smaller diameter passages. The curve in the pipe segment causes a non-uniform flow field that creates steady-state but non-uniform temperature and pressure distributions on the inner passage surface. These temperature and pressure distributions couple with non-uniform thermomechanical stress and deformation fields within the pipe segment. In turn, the thermomechanical fields act on an initial crack-like defect in the pipe wall, causing this defect to propagate.

The workflow of a single time step of the simulation is shown in Figure 1. In this figure, the components of our system appear like this, the intermediate data sets appear like *this*, and the "human in the loop" appears like this. In our current workflow, the only data that is passed from one timestep to the next is the geometric model of the pipe, which is updated in each time step as the defect is inserted and grown.

Crack initiation is an active area of research in Fracture Mechanics, and, at present, is not understood well enough to do automatically. Hence, in our present system, we require a knowledgeable user to manually determine the Initial Flaw Parameters by studying the displacement field at the end of time step $t = 0$. This is shown as a component labeled "Client: Crack Initiation" in Figure 1. In subsequent timesteps, $t = 1, 2, \ldots$, state of the art Fracture Mechanics techniques are used to predict the trajectory along which the crack defect will grow.

Here is a brief description of the some of the components of our system that are especially relevant to this paper:

- The *Surface Mesher* produces triangular meshes for each of a model's geometric surfaces. This component

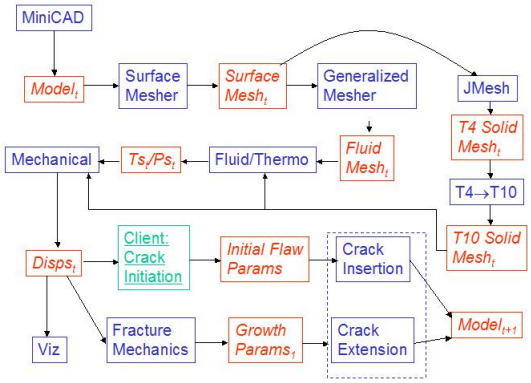

Figure 1. Workflow for the Pipe problem

produces surface meshes with certain quality guarantees [5].

- The *Generalized Mesher* ([4, 3]) generates high quality meshes consisting of extruded triangular prisms, tetrahedral elements, and generalized prisms. These highly anisotropic elements are required for simulating viscous fluid flows required in regions near no-slip boundaries, i.e., boundary layers.
- *Jmesh* ([2]) generates unstructured tetrahedral meshes for arbitrarily shaped three-dimensional regions, and was designed to handle the unique geometric problems that occur in Fracture Mechanics.
- The *Fluid/Thermal Solver* is based upon the CHEM code [18, 19], which is a library of Loci ([18, 17]) rules that simulate 3-D chemically reacting flows of thermally perfect, calorically imperfect gases.
- The *Mechanical Solver* solves the equations of linear elasticity to determine the deformation of the pipe due to different loading conditions (e.g. pressure on the inner pipe) and thermal expansion.

In a project such as this, common data formats are necessary for conveying information between the components. In past projects, we have used file formats encoded with XDR ([27]) and HDF5 ([15]). However, we found that, while binary encodings are ideal for productions systems, they make research and development more difficult.

One of the earliest decisions that we made in our project was to establish a set of common file formats that used XML [31] for encoding. We decided to use XML because it is widely supported and because XML parsers and validation tools are available for all of the languages and platforms that we use. We have also found that having a human readable format has made debugging much easier. Some of these formats are described elsewhere ([6],[8]).

3 Web Services Infrastructure

We have implemented the components described in the previous section as a set of Web Services. In particular, each of the components in our current system has been developed and deployed using the following Web Services frameworks:

Microsoft .NET On our Windows platforms, we use Microsoft .NET ([10]), which provides a "holistic" approach to distributed applications.

SOAP::Clean On our UNIX and Linux platforms, we use SOAP::Clean ([28]), a Perl module for exposing legacy applications as Web Services. Compared with .NET, SOAP::Clean provides a "minimalistic" approach to distributed applications.

Since .NET is already extensively documented and because SOAP::Clean is used for most of the components in our experiments, we will discuss SOAP::Clean is more detail below.

3.1 SOAP::Clean

SOAP::Clean was original developed for two purposes, to enable legacy applications to be easily deployed as Web Services, and to provide a command-line tool for invoking Web Services.

On the server-side, SOAP::Clean enables existing, command-line oriented applications to be made into Web Services with no modification. In order to deploy an application as a Web Service, a user must write a small CGI script using the SOAP::Clean library. An example of such a script is shown in Figure 2.

When this script is executed, a number of methods in the SOAP::Clean library are executed. The `urn`, `name`, and `full_name` methods are used to specify the namespace and name of the Web Service. The `descr` method is used to specify the command line that is to be run when the Web Service is invoked. The text that appears within `[...]` describes the parameters to the command line. Each parameter specification includes at least four properties,

- The directionality of the parameter, i.e., "in", "out", or "in_out".
- Whether the parameter value should appear directly on the command line ("val") or whether the parameter value should be placed in a file whose name appears on the command line ("file").
- The name of the parameter, e.g., "x", "y" and "out".
- The type of the parameter value, i.e., "int", "float", "string", "raw" (arbitrary binary file), "xml" (a structured XML file).

Finally, the `go` method is invoked to inform the SOAP::Clean library that the Web Service is completely

specified, and that the request message should be read and processed.

```
#! /usr/bin/env perl

use SOAP::Clean::CGI;

new SOAP::Clean::CGI
  ->urn('urn:test')
  ->name('arithmetic-test')
  ->full_name('Arithmetic Test Server')
  ->descr("./test.sh -x[in val x:int] ".
          "    -y[in val y:int] ".
          "    > [out file result:int]")
  ->go();
```

Figure 2. Sample SOAP::Clean Server

```
% wsdl-client.pl \
    http://somewhere.com/test.cgi?WSDL \
    call x=2 y=3 out:output.txt
```

Figure 3. Sample invocation of a remote Web Service

On the client-side, SOAP::Clean provides the `wsdl-client.pl` program, which is intended to make Web Services look like legacy, command-line oriented applications. Figure 3 shows showing `wsdl-client.pl` being used to invoke the Web Service from Figure 2.

It may seem odd to want to run a Web Service as if it were a traditional command-line-oriented program, but there are a number of benefits that come from this. For instance, if Web Services appear as programs, then a computational scientist can write a script using, for example, Bourne Shell or UNIX Make that composes several Web Services into a larger application. This is, in fact, how we have implemented the pipe problem simulation described in Section 2.

Of course, since Web Services are based on standard protocols, there are many other ways of invoking our components. First, many Web Services infrastructures (such as SOAP::Clean and Microsoft's .NET) allow services to be accessed via conventional web browsers. Second, libraries for accessing Web Services are available for many languages. These include, gSOAP ([12]) for C/C++, SOAPpy ([24]) for Python and SOAP::Clean and SOAP::Lite ([16]) for Perl.

3.2 Potential performance overheads

There are a number of places in the pipe simulation system and SOAP::Clean infrastructure where our current implementation is suboptimal. We highlight these deficiencies here, and discuss their impact on the overall system performance in Section 5.

As noted in Section 2, we have established a number of XML-based file formats that we use for encoding data objects, like geometric models, unstructured and semi-structured meshes, and field results, between each of the components. As we will discuss in Section 5, there is an overhead associated with using XML instead of a more traditional binary data encoding.

There are two basic approaches to parsing XML objects, DOM and SAX. The DOM approach involves parsing an entire XML file and constructing a data structure representing its content in memory. The SAX style approach involves calling user-supplied call-back functions during XML parsing. By doing so, this approach can avoid having to store the entire XML file in memory. SOAP::Clean uses the DOM approach, which is probably the less efficient of the two. In addition, SOAP::Clean actually parses each XML file *twice* in order to circumvent a bug in the underlying XML parser.

Because SOAP::Clean serves as an interface to existing application codes, each XML file is parsed and copied several more times than would occur in an optimized Web Service. For example, if each component in our system had been originally implemented as a Web Services, then the intermediate XML files used to pass data between SOAP::Clean and the application would not be needed. Also, it is possible that SOAP::Clean could be made to copy unparsed XML files directly into these intermediate files, but it does not currently do so. At present, both SOAP::Clean and the application parse each XML file.

At present, `wsdl-client.pl` requests the WSDL interface from a Web Service prior to performing any method invocation. This doubles the number, but not size, of the messages required to perform an invocation. It should be possible to cache the WSDL on the client and use it over multiple method invocations.

4 Performance Results

The following machines were used for the experiments below,

- The *ASP* cluster is housed in the Cornell Computer Science department and consists of 5 Dell PowerEdge 1650's, each with Pentium III's at 1.26GHz (1 dual and 4 single). Each node had 512MB-1GB RAM and ran Red Hat Linux 8.0.
- Web Services at the Cornell Theory Center, or *CTC*, are implemented using a number of machines. CTC-STAGER hosts the web server (IIS 5.0) that receives the SOAP requests. LSQLSRV03 hosts the databases (SQL Server) that are used for storing the input and output data files. The computation was performed on

the CMI cluster, which has 32 dual nodes (Dell 1550), each with 2 PIII at 1GHz. Each node has 2 GB RAM. All machines run Windows 2000 Advanced Server.
- Web Services at Mississippi State University, or *MSU*, were executed on an IBM x330 server, with dual 1.266GHz Intel Pentium III CPUs and 1.25GB RAM running Red Hat Linux version 7.3.
- The machine used at the University of Alabama at Birmingham, or *UAB*, is an IBM x335, with dual 2.4GHz Xeon and 2GB RAM, and runs Red Hat Linux release 7.3.

Except where noted, the components used in these experiments were deployed on the ASP cluster.

4.1 Overhead of using XML files

As we described in Section 2, we designed and used common formats based upon XML for our intermediate files. Since many of our component applications were developed before this project, we have implement converters to translate between their original formats and our common XML-based formats.

	Runtime (seconds)		Overhead	
	Base Appl.	w/XML	Abs.	Rel.
Surface Mesher	58.68	61.93	3.25	5.54%
JMesh	101.59	109.71	8.12	7.99%
Fluid/Thermal	1527.86	1627.15	99.29	6.50%

Table 1. Overhead of using XML files

We can easily measure the cost of this file conversion for three components in our system. Both JMesh and the Fluid/Thermal solver have to convert their input and output files between XML and their original file formats, while the Surface Mesher only has to convert its output. The costs of these conversions are shown in Table 1. The columns labeled "Base Appl." and "w/XML" show the running time in seconds of each component without and with, respectively, XML file conversion. The next two columns show the absolute and relative overhead of the file conversion, with respect to the "Base Appl."

4.2 SOAP Message Sizes in bytes

Table 2 shows the sizes of SOAP messages that are exchanged between the client and servers. There are four sets of columns, one for each of the four operations required in order to invoke a component. For the SOAP::Clean component, these operations are,

- The client requests the WSDL for the component from the server, which is labeled *Retrieve WSDL*.
- The client uses the *Spawn* operation to upload arguments to the server and request that it start executing the component. The server returns a UID, which serves as a handle for the component instantiation.
- The client uses the *Running?* operation to query the server to see if the component is still running.
- After the component has completed, the client uses the *Results* operation to download the results from the server.

The operations for the Mechanical solver, which is the only .NET-based component, are similar.

The size, in bytes, of the SOAP request (client-to-server) and response (server-to-client) messages are shown within each set of columns.

4.3 Overhead of using Web Services

Table 3 shows the running times and Web Service overheads for our system. These results are from the first two time steps, $t = 0$ and $t = 1$, of a simulation of the pipe problem. Each row of the table shows the results for the individual components within each time step, except for the last row, which shows the aggregate results for the entire system. The column labeled "Base Running" shows the running time in seconds of each component when it is executed directly on the server, without using the Web Services infrastructure. The overheads are based on these times.

Because they can take an unpredictable amount of time to run, we run most components in an asynchronous manner. The column labeled "Polling Freq." gives the time in seconds between subsequent calls to a component's "Running?" operation. A few components are known to take very little time to run and are run synchronously; these components are denoted with a polling frequency of 0.

The columns labeled "Local WS" contains the execution times obtained by running the client and all but one of the servers on the ASP machine. These times differ from "Base Running" in that all of the communication between the client and the servers is forced to go through the HTTP server and SOAP::Clean libraries. In a sense, they represent the "pure" overhead from using Web Services, as they do not include network latency.

The Generalized Mesher component resided on the MSU machine, so its times always include the use of Web Services and network latency. The "Base Running" times measure the cost of executing this Web Service synchronously, while the "Local WS" times measure asynchronous execution.

The columns labeled "Remote WS" contain executions times obtained by running the client on the UAB machine in Alabama, while the servers remained on the MSU and ASP machines in Mississippi and New York, respectively.

	Retrieve WSDL		Spawn		Running?		Results	
	request	response	request	response	request	response	request	response
Surface Mesher	85	6,708	135,074	875	720	744	720	377,992
JMesh	76	7,176	512,386	908	744	744	744	1,334,431
T4→T10	80	7,004	1,468,764	880	720	744	720	4,454,170
Generalized Mesher	72	7,678	513,015	1,305	1,171	724	1,171	2,824,138
Fluid/Thermal	84	8,490	7,416,514	1,017	861	744	861	1,038,255
Crack Insertion	87	6,556	135,017	851	698	745	698	141,467
Fracture Mechanics	89	7,302	6,621,903	977	826	744	826	3,201
Crack Growth	83	6,552	141,299	851	694	745	694	180,465
Mechanical	81	17,031	5,500,714	670	619	662	640	1,861,450

Table 2. SOAP message sizes in bytes

t	Component	Base Running (secs.)	Polling Freq. (secs.)	Local WS running (secs.)	Local WS absolute (secs.)	Local WS relative	Remote WS running (secs.)	Remote WS absolute (secs.)	Remote WS relative
0	Surface Mesher	60.74	10	76.45	15.71	25.86%	75.05	14.31	23.56%
	JMesh	91.52	10	101.13	9.61	10.50%	100.01	8.49	9.28%
	T4→T10	64.88	10	79.88	15.00	23.12%	81.84	16.96	26.14%
	Generalized Mesher	32.03	10	37.19	5.16	16.11%	31.14	-0.89	-2.78%
	Fluid/Thermal	1536.95	60	1570.48	33.53	2.18%	1570.80	33.85	2.20%
	Crack Insertion	0.30	0	10.46	10.16	3386.67%	8.77	8.47	2823.33%
1	Surface Mesher	62.69	10	76.80	14.11	22.51%	75.20	12.51	19.96%
	JMesh	123.56	10	135.71	12.15	9.83%	137.06	13.50	10.93%
	T4→T10	72.90	10	80.19	7.29	10.00%	82.02	9.12	12.51%
	Generalized Mesher	34.39	10	39.12	4.73	13.75%	30.12	-4.27	-12.42%
	Fluid/Thermal	1628.59	60	1680.90	52.31	3.21%	1678.98	50.39	3.09%
	Crack Growth	35.52	0	45.02	9.50	26.75%	51.77	16.25	45.75%
	Fracture Mechanics	0.33	0	11.15	10.82	3278.79%	9.49	9.16	2775.76%
	Total	3744.40		3944.48	200.08	5.34%	3932.25	187.85	5.02%

Table 3. Overhead of using Web Services

Any difference between these times and the "Local WS" times should reflect changes in network latency.

We have not included running times for the Mechanical Solver, which was the only component that required executing via a batch queue. We found that the time spent waiting in the queue dominated the total running time and varied from minutes to days. As a result, we had no way of directly measuring the Web Services overhead of this component.

5 Performance Analysis

In this section, we analyze the experimental results of the previous section and discuss the performance bottlenecks identified by previous work in the context of our system.

5.1 Using XML

[9] states that an XML data representation can be 4-10 times larger than an equivalent binary representation and that over 90% of the CPU cycles for processing SOAP messages can be spent performing ASCII-to-double conversions. Furthermore, our own results show that the Fluid/Thermal solver takes 99 seconds to convert approximately 2 megabytes of XML data to and from a native format.

While these results may appear very alarming, our observed overhead from using XML formats was less than 8%. The reason why the overhead was so low is because it is offset by the longer running times of the base components. This is clearly shown by Table 1. We conclude that the short-term benefit from having standard, human-readable file formats has greatly outweighed the small performance penalty.

5.2 Using Web Services

The overhead that we observed from using Web Services is about 5%. This is significantly less than what was observed by previous studies. How do we account for this?

Is it because our Web Services infrastructure is highly optimized? Quite the contrary. Section 3.2 enumerated a number of limitations of our current implementation that

might be potential performance bottlenecks. In fact, we would go so far as to say that we violated every performance recommendation made in [9] and [26]!

The reason why our overhead is so low is because of how we have divided our system into components. As Table 3 shows, most of the running time of our system is taken by two executions of the Fluid/Thermal solver[2]. The Fluid/Thermal solver is an MPI ([29]) program that is executed on a tightly coupled cluster. So, while the execution of this one component may involve a large number of messages being exchanged between processors, this is all done using MPI, a message-passing library designed for this purpose.

To summarize, our system is built on top of a relatively inefficient Web Services infrastructure, but about 95% of its execution time is spent within each of the components. Our conclusion is that the organization of a distributed simulation system makes more of a difference to its performance than the underlying Web Services infrastructure. The key seems to be ensuring that most of the time-consuming computation and communication occur *within* and not *between* components.

6 Related Work

A number of frameworks and standards have been proposed for developing component-based systems. Perhaps the best known are CORBA ([23]) and COM ([21]). We investigated these frameworks, but found that using them would require us to make extensive modifications to our existing applications. We also found that the existing frameworks were primarily designed for deploying applications within a single machine. DCOM ([20]) is one exception to this. It is also interesting to note that existing component frameworks are evolving towards interoperability with Web Services (witness .NET subsuming COM and DCOM, and the OMG's adoption of a specification on CORBA-WSDL/SOAP Interworking).

Ninf [22] and NetSolve [1] are intended to allow existing numerical *libraries* to be executed remotely, while SOAP::Clean and the other elements of our infrastructure are intended to allow existing *applications* to be executed remotely. As a result, the type systems are different. For example, both Ninf and NetSolve provide array and subarray types, while SOAP::Clean only provides simple scalar types ("int", "float", "string"), a binary file type ("raw") or an arbitrary XML file type ("xml").

[2]The running time of the Mechanic Solver, whose results we have not included here, can be anyway from 50% to 5000% of the running time of the Fluid/Thermal solver, depending upon how long a job request has to wait in the batch queue.

7 Conclusions

We have described a multi-physics simulation testbed that consists of a loosely coupled set of distributed components implemented using Web Services. This testbed has enabled us to (a) develop state-of-the-art simulations without having to port codes between each others machines and (b) provides us with a platform on which to study adaptivity in scientific applications. This approach has given us a number of development and software maintenance benefits, and has cost us very little in terms of performance (about 5% overhead).

Our results suggest that even a simple and standard-compliant Web Services infrastructure, such as SOAP::Clean, can be used directly in high performance distributed scientific computing without introducing burdensome performance bottlenecks. We have suggested that this is a result, not of our Web Services implementation, but of how we have organized our system into components.

We believe that our work provides a number of important lessons for other researchers. First, with this sort of infrastructure, it is possible for multi-institutional, multi-disciplinary computational science projects to establish virtual organizations, as envisioned in [14]. This is possible even with primitive Grid technology.

Second, in order to achieve reasonable performance from a distributed simulation system, it is important to carefully chose the functionality that goes into each of its components. Loosely coupled codes that communicate infrequently can be placed in separate components, while tightly coupled codes should almost certainly be placed within the same component. Individual sites will probably have enough resources to do matrix multiplication or solve large systems of linear equations without harnessing computational resources from multiple sites, so the role of Web Services in such projects is to make it possible for large codes to inter-operate with minimal coordination and re-implementation.

We believe that this sort of decomposition is a natural result of, not only our physical problem, but of the fact that we are a multi-disciplinary project. In such a project, each member has a clearly defined research area, and the components seem to natural divide themselves along these lines. Put differently, our components are loosely coupled because our project members are! We expect that this will be true of most other multi-disciplinary projects, and we believe that Web Services may be appropriate for many of these as well.

We are pleased with the performance of our present system, but there is still enormous room for improvement. This paper has enumerated a number of places where our implementation can be improved; as part of our future work, we will address these issues. Another issue that we will address is scalability. Our experience suggests that the overhead

of processing SOAP relative to the cost of the computation goes *down* as the problem size increases. This is inconsistent with previously published results, and we are currently modifying our system so that we can readily generate problems of varying size in order to test our hypothesis.

References

[1] D. C. Arnold and J. Dongarra. The netsolve environment: Progressing towards the seamless grid. In *2000 International Conference on Parallel Processing (ICPP-2000)*, Toronto, Canada, August 21-24 2000.

[2] J. Cavalcante-Neto, P. Wawrzynek, M. Carvalho, L. Martha, and A. Ingraffea. An algorithm for three-dimensional mesh generation for arbitrary regions with cracks. *Engineering with Computers*, 17:75–91, 2001.

[3] S. Chalasani and D. Thompson. Quality improvements in extruded meshes using topologically adaptive generalized elements. *International Journal for Numerical Methods in Engineering*, (submitted).

[4] S. Chalasani, D. Thompson, and B. Soni. Topological adaptivity for mesh quality improvement. In *Proceedings of the 8th International Conference on Numerical Grid Generation in Computational Field Simulations*, Honolulu, HI, June 2002.

[5] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Symposium on Computational Geometry*, pages 274–280. ACM Press, 1993.

[6] L. P. Chew, S. Vavasis, S. Gopalsamy, T. Yu, and B. Soni. A concise representation of geometry suitable for mesh generation. In *Proceedings, 11th International Meshing Roundtable*, pages pp.275–284, Ithaca, New York, USA, September 15-18 2002.

[7] P. Chew, N. Chrisochoides, S. Gopalsamy, G. Heber, T. Ingraffea, E. Luke, J. Neto, K. Pingali, A. Shih, B. Soni, P. Stodghill, D. Thompson, S. Vavasis, and P. Wawrzynek. Computational science simulations based on web services. In *International Conference on Computational Science 2003*, June 2003.

[8] P. Chew and S. Vavasis. Proposal for mesh representation. Internal draft, January 21 2003. Accessed February 13, 2003.

[9] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, July 2002.

[10] M. Corporation. Microsoft .NET. Accessed February 11, 2003.

[11] C. Douglas, A. Deshmukh, et al. Report from the March 8-10, 2000 NSF sponsored workshop on Dynamic Data Driven Application Systems. Accessed February 8, 2003.

[12] R. A. V. Engelen and K. A. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, page 128, Berlin, Germany, May 21 – 24 2002.

[13] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22 2002.

[14] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[15] Hdf5 - the next generation of the hdf library & tools. Accessed on June 3, 2003.

[16] P. Kulchenko. Web services for perl (soap::lite, xmlrpc::lite, and uddi::lite). Accessed on June 3, 2003.

[17] E. Luke. Loci: A deductive framework for graph-based algorithms. In S. Matsuoka, R. Oldehoeft, and M. Tholburn, editors, *Third International Symposium on Computing in Object-Oriented Parallel Environments*, number 1732 in Lecture Notes in Computer Science, pages 142–153. Springer-Verlag, December 1999.

[18] E. A. Luke. *A Rule-Based Specification System for Computational Fluid Dynamics*. PhD thesis, Mississippi State University, 1999.

[19] E. A. Luke, X. Tong, J. Wu, L. Tang, and P. Cinnella. A step towards "shape-shifting" algorithms: Reacting flow simulations using generalized grids. In *Proceedings of the 39th AIAA Aerospace Sciences Meeting and Exhibit*. AIAA, January 2001. AIAA-2001-0897.

[20] Microsoft, Inc. Distributed component object model (DCOM). Accessed February 13, 2003.

[21] Microsoft, Inc. Microsoft COM technologies. Accessed February 13, 2003.

[22] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of ninf: towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.

[23] Object Management Group, Inc. Welcome to the OMG's CORBA website. Accessed February 13, 2003.

[24] Python web services. Accessed June 12, 2003.

[25] K. Seymour, H. Nakada, S. Matsuoka, D. Dongarra, C. Lee, and H. Casanova. Gridrpc: A remote procedure call api for grid computing. ICL Technical Report ICL-UT-02-06, Innovative Computing Laboratory, Department of Computer Science, University of Tennessee, June 2002.

[26] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating web services based implementations of gridrpc. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, 2002.

[27] R. Srinivasan. Xdr: External data representation standard, Aug. 1995. IETF RFC 1832.

[28] P. Stodghill. SOAP::Clean, a Perl module for exposing legacy applications as web services. Accessed February 11, 2003.

[29] D. W. Walker and J. J. Dongarra. MPI: a standard Message Passing Interface. *Supercomputer*, 12(1):56–68, 1996.

[30] World Wide Web Consortium. Web services activity. Accessed June 11, 2003.

[31] World Wide Web Consortium. Extensible markup language (xml) 1.0 (second edition). W3C Recommendation, October 6 2000.