Web Service Call Parallelization Using OpenMP

Sébastien Salva[1], Clément Delamare[2], and Cédric Bastoul[3]

[1] LIMOS, Université d'Auvergne Clermont 1
Campus des Cézeaux
F-63173 Aubière, France
salva@iut.u-clermont1.fr

[2] DGI, Direction Générale des impôts, France
86 allée de Bercy, 75012 Paris
clement.delamare@dgi.finances.gouv.fr

[3] LRI, ALCHEMY, Université Paris-Sud
F-91405 Orsay, France
cedric.bastoul@lri.fr

Abstract. Since early works, web servers are designed as parallel applications to process many requests at the same time. While web service based applications are performing more and more, larger and larger transactions, this parallelization culture still not reached the client side. Business to Business (B2B) applications are becoming intensive users of web transactions through Service Oriented Architecture Standard. As multicore systems are now widely available, parallelization seems the right way to fit the need of such applications. In this paper, we describe an API based on OpenMP for transparently parallelizing web service calls, i.e. the serialization, deserialization and connection processes. Our API is mainly based on a software pipeline which splits web service calls into several tasks. Both synchronous and asynchronous modes can be used with this API to call web services. We present experimental evidence demonstrating the ability of our API to achieve improved performance.

1 Introduction

SOA (Service Oriented Architecture) is emerging as the standard paradigm to develop business applications over Internet, like Business to Business (B2B) and Business to Consumer (B2C) applications which involve the transaction of goods or services. Such applications are mainly based on web service interactions. Web services represent objects whose methods can be called through Internet. They generally accept parameters (objects) and generate a response. Both parameters and responses are serialized/deserialized by using an XML based protocol, named SOAP (Service Oriented Architecture Protocol).

Business applications, like banking systems, require a constantly growing number of large data transactions. Those data flood capacity as the throughput is increasing by ten times nowadays. As a consequence, such applications require more and more performance, especially for the costly data serialization/deserialization processes. For B2B applications which perform a lot of web service transactions, the use of monocore systems is becoming a limitating factor. Because of the inherently parallel nature of web service calls, the trend is to benefit from multicore systems to improve server capacity, as Sun T1 (Niagara) architecture shows. However, a trivial parallelization scheme does not reach the maximal service performance because of the limited number of cores and a suboptimal use of resources when server threads stall, waiting for other server responses.

In this paper, we present a solution using OpenMP to efficiently parallelize web service based applications on multicore systems. We describe an API for transparently parallelizing web service calls on multicore systems. This API is easy to use: the developer gives the web services to call and its code to manage web service responses (they can be stored or analyzed to produce other calls).

Our API is mainly based on a software pipeline whose stages split web service calls into independent tasks and parallelize their execution. We show that this solution performs better than simple parallelization schemes when using both synchronous web service which responds in some milliseconds, and asynchronous ones whose responses may be received from some seconds to some days later. Because Java is widely used for coding web service based applications, our API is written using this language. We have also used the OpenMP Jomp API [1] for the parallelization. OpenMP brings many advantages, especially for the software pipeline generation and for the parallelization of the serialization/deserialization processes, which are mainly based on loops.

The remainder of this paper is structured as follow: Section 2 provides an overview of the web service paradigm and of a web service based project, Copernic [2]. Then, we present some related works. Section 3 describes our API mainly based on a software pipeline. Section 4 shows some experimentations. Finally, Section 5 gives some perspectives and conclusions.

2 Web Services Overview and Related Work

Web services are "self contained, self-describing modular applications that can be published, located, and invoked across the web" [3]. The Web Service framework, which is a platform independent standard, is composed of three major axes:

- The *service description* which models how to invoke the service by defining its interface and its parameters. This description, called WSDL (Web Services Description Language) file [4], gathers several parts describing types, accessible methods, query and response message structures, binding and transport options.
- The *definition and the construction of XML messages*, based on the Simple Object Access Protocol (SOAP). SOAP aims to invoke service operations (object methods) on the Internet and to serialize/deserialize data. When SOAP is used over the HTTP layer, the web service is said to be called in *synchronous mode.* When it is used over the SMTP layer, it is said to be called in *asynchronous mode.*
- The *discovery of the service.* Web services are gathered into UDDI (Universal Description, Discovery Integration) registries. These registries can be

consulted manually or automatically using dedicated APIs to find specific web services. The WSDL files are grouped into these registries.

Web services allow to externalize functional code of companies in a standardized way. Web Services are mainly used for B2B or B2C transactions, to exchange data and externalize functional code.

An example of advanced SOA based project is Copernic [2]. Copernic is a modernization program that will recast and upgrade the entire fiscal information system in France. At the moment, it covers 70 individual projects, managed by the Direction Générale des Impôts and the Direction Générale de la Comptabilité Publique. Its objective is to enable the French tax administration to offer new, citizen-centered services and to boost the efficiency of its internal processes. A lot of data is managed with Copernic, like the PERS reference which models a citizen or the TOPAD reference which models all the French roads with postal addresses. Such data are accessed by web services to modify or consult them. Copernic produces large web service transactions, especially with specific tasks, like the edition of the 33 millions income tax returns which uses many references (PERS, TOPAD and many others ones).

Experiences, to parallelize web servers using OpenMP, have been gained by Balart et al. [5], but few solutions have been proposed to parallelize web service calls even in the general case. Kut and Birant suggested an approach which uses some threads to call several web services in the same time [6]. We show, in Section 3.1 that this method should not be used when the response receipt delay is long (especially with asynchronous web services). Wurz and Schuldt consider another approach for web service parallelization: if the original request is composed of several independent sub-requests, this one is split and the sub-requests are executed in parallel [7]. This solution offers good results, however most of web services use atomic requests (i.e., they are not composed of sub requests).

3 Web Service Call Parallelization

SOA based applications (for e-commerce, B2B or B2C applications) perform large transactions with web services. There are several advantages to parallelize such services: from being able to call several other services at the same time, up to reduce the time necessary to serialize/deserialize data.

Here, we focus on the client side, i.e., applications that call successively web services which may produce themselves other web service calls. In this section, we propose and describe an API which aims at easily and efficiently constructing such applications. For a developer, this API is mainly based on a class named ParallelClient, which is used to set the number of working threads and to add web service calls. The developer has only to focus on the implementation of the web service responses management: data received from web services can be either stored or analyzed. The rest of the client application, i.e., the data serialization/deserialization processes and the web service calls, is made in parallel transparently thanks to the API.

Web service based applications are typically written in C# or in Java. To construct a parallel client API, we chose Java and the Jomp OpenMP API [1].

In the following, we consider that a complete web service call is composed of the following steps:

1. the S step: a data serialization into a SOAP-XML document,
2. the C step: a web service connection, that is a TCP connection with an Internet service provider to send the serialized data and to receive a SOAP-XML response,
3. the D step: a response deserialization whose memory size may be important (100MB or even more)
4. the P step: obtained data are stored in a persistent infrastructure like a database, or may be analyzed.

3.1 A Naive Solution: Using the Task Pool Paradigm

Because of the inherently parallel nature of web service calls, a basic solution would be to use a task pool where one task represents one web service call (including all the steps previously defined). Therefore, with this solution, if n threads are available, n calls can be easily done in parallel, assuming n cores are available. The task pool algorithm is described in Figure 1.

```
pool is a FIFO containing tasks;
setNumThreads(n);

// omp parallel shared(pool) private(....){
// omp critical {
task=read(pool);}
while(task != NULL){
     SOAPdata=Serialize(task.data);
     Response=WebService.call(SOAPdata, endpoint);
     Object[] obj=Deserialize(Response);
     Persistency(obj);
     // omp critical {
     task=read(pool);}
     }
}
```

Fig. 1. The task pool pseudocode

Figure 2 illustrates a task pool execution with only two threads. After each web service connection, each thread must wait for the web service data receipt before deserializing it. It follows, for n threads after n web service connections, the application may be blocked. When web services are called in synchronous mode, the application is blocked during some seconds. However, in asynchronous mode, the client application may be blocked during some days. Furthermore if the target architecture has less than n cores (or n processors for symmetric multiprocessor architectures) this policy will result in thread interleaving, with a potentially high cost induced by thread management and context switching. Hence this solution does not appear as the best one to parallelize web service calls.

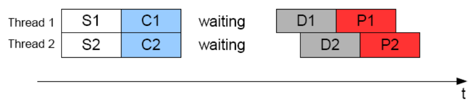
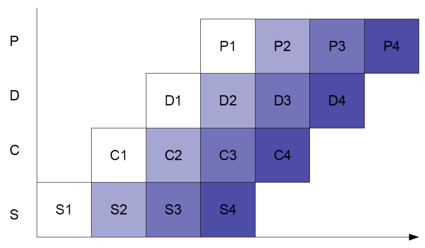


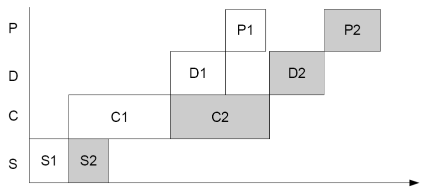
Fig. 2. Two threads executing two web service calls

3.2 A Better Resource Management Using the Pipeline Paradigm

The web service call steps are executed successively and are completely independent. As a result the pipeline paradigm can be used to parallelize web service calls. We use a pipeline of four stages, one for each step described previously: a serialization stage (S), a web service call stage (C), a deserialization stage (D) and a stage for the persistency (P). This scheme is depicted in Figure 3(a).



(a) A software pipeline for web service calls



(b) The software pipeline blocked by web service connections

Fig. 3.

This solution requires at least four cores, each of them being dedicated to a thread processing a given step. This solution avoids thread interleaving and achieves a better use of resources since the system overlaps service call stalling time (while waiting for a response) with processing.

However, such a pipeline architecture also has some drawbacks:

```
FIFO fifo1, fifo2, fifo3;
//omp parallel sections {

//serialization code
task= read(fifo1);
task2.serializeddata=serialize(task.data);
write(task2,fifo2);
}
//omp section {
//web service connection code
...
}
//omp section {
//deserialization code
...
}
//omp section {
//data storage code or data analysis
...
}
}
```

Fig. 4. The pipeline pseudocode

- The web service connection and the waiting of its response is usually a blocking operation. As the response receipt delay may be long (until more than one day in asynchronous mode), the pipeline may be blocked as well. Therefore, the two last stages may stay empty until a response is received. In Figure 3(b), we illustrate two web service calls: the C stage must receive the response of the first web service before calling the second one. As a consequence, the pipeline is blocked. In this case, there are few advantages to parallelize this step.
- The serialization and deserialization processes may require long time executions. For instance, a deserialization of 100Mbytes*** of SOAP-XML data may require more than 30 minutes with a recent monocore processor. As a consequence, the load balancing between the pipeline stages may be catastrophic if there is a lot of data to serialize/deserialize. So these two steps must be also parallelized to reduce the execution time of the serialization/deserialization stages.

Despite these drawbacks, the present solution appears to be better than the trivial one. Indeed the S stage can always work until there are web services to call, since it is never blocked.

3.3 An Optimized Pipeline for Web Service Calls

A refinement of the previous four stage pipeline solution is to extend the first three stages. We do not develop the Persistency stage here because this one depends mainly on the data storage.

The call stage For the web service call stage, we use the ProviderConnection architecture proposed by SUN [8]. In this one, the web service connection is not made directly by the client application but by using a service (daemon) which can be localized in the same computer than the Client or in a gateway. In our case, this service is composed of two servlets, the first one is used for calling the web services, and the other one receives the web service responses. So, with such an architecture, we perform non blocking web service calls.

The call stage, illustrated in Figure 5, is finally composed of a task pool where a task is an object {call identifier, URL, method, Message} which models a web service call. An available thread executes a task (1). And it calls the web service by using the Send servlet (2). The Send servlet makes a connection to the web service and sends SOAP-XML data (3). The Response servlet receives the response (4). Each response is given to the Deserialization stage (5).

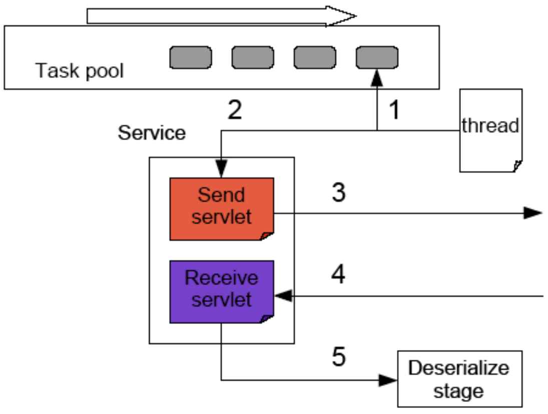


Fig. 5. The call stage architecture

By using such an architecture, the stages are never blocked. The S stage can work until there is web services to call. And the C stage can now establish all the connections. When a response is received, this one is deserialized by the D stage and stored by the P one. Each web service call is numbered by an unique identifier, thus responses may be received in any order. In the synchronous mode, all the stage works. However, in asynchronous mode, if the receipt delay is long, some stages may become idle. So, in this case, the load balancing is not good. We give two approaches to solve this problem in the perspectives.

The Serialization/Deserialization stages Serialization and deserialization stages are both parallelized using a team of threads. As the data (objects) to serialize are independent, we use one thread team to serialize in parallel all the objects, then the final XML-SOAP document is generated. Similarly, all the elements of a SOAP-XML document can be extracted and then deserialized in parallel by another team of threads. The algorithms of these stages are given in Figure 6:

```
Serialization Stage
Input: List of nb objects
Output: SOAP_XML_document document
A list Lfinal of nb empty elements is defined
// omp parallel for private(element,i) shared (List,Lfinal)
for(i=0,i<nb;i++){
     Read List[i];
     (XML element)Lfinal[i]=Serialize L[i];}
for(i=0;i<nb;i++)
     document.add(Lfinal[i]);

Deserialization Stage
Input: SOAP_XML_document document
Output: List_object of nb objects
nb=numbers of elements(objects) in the SOAP document
A list L of nb empty elements is defined
for(i=0;i<nb;i++)
     L[i]=document.get_element(i);
//omp parallel for private(object,i) shared (List,L)
for(i=0,i<nb;i++){
     List_object[i]=Deserialize L[i];}
```

Fig. 6. The Serialization/Deserialization stage pseudocode

We show in the following section that the performance of the optimized pipeline is better than the task pool solution. And more precisely, the longer the web service data receipt is, the better the performances are. This is a consequence of the unblocked connection to web services that the pipeline brings. For a task pool solution, if we use n threads for $2n$ transactions, the first n transactions are performed, then the next ones are done too when the first n transactions are finished. With the optimized pipeline solution, all the transactions are executed in succession and responses are given to the deserialization stage in succession too. By considering only the web service connection time, if this one takes on average 5 minutes, the task pool solution needs at least 10 minutes to finish $2n$ transactions. The pipeline solution needs 5 minutes.

4 Experimentation Based on the Copernic Project

Our experimental setup was based on two dual processor Xeon systems taking advantage of hyperthreading. Each system is able to run 8 threads in parallel. Our API has been applied onto a modified version of the Copernic Framework [2]. Operating Systems were based on RedHat Advance Server 4 with JAVA applications running thanks to Sun JDK 1.5.0.9 and Jomp API 1.0b. The first system is dedicated to service, provided by JBOSS 4.0.4-GA, while the other one is on the client side.

Not surprisingly, parallelizing web service calls lead to major performance improvements on our multithreading-capable systems. For transactions with very small response time, the cost of thread management and probably the limitations of the hyperthreading technology limits the global speedup to a factor of 3.5 instead of an ideal factor of 8. The optimized pipeline solution gives by far the best results compared to the naive scheme.

Our results are depicted in Figure 7. We compared the task pool and the optimized pipeline solutions for various response delays and number of threads. We consider for each experiment 100 service calls with 4000 data to serialize/deserialize and a response time of 1s, 5s and 10s. The reference speed for speedup computation is the naive scheme using a single thread. For small response times and at least 5 threads, the pipeline solution is a 10% improvement over task pool solution thanks to a lighter thread management cost. The speedup of the pipeline solution over task pool solution grows dramatically with response waiting time, from 30% for 5s response time to more than 60% for 10s response time, since the optimized solution is less challenged by blocked threads.

5 Conclusion and Perspectives

In this paper, we propose an API to efficiently parallelize web service calls. This API is mainly based on a software pipeline which splits web service calls into several tasks, taking advantage of OpenMP for a better use of resources. We have shown that any web service can benefit from our API (both synchronous and asynchronous ones). Using OpenMP to design such an API showed high productivity. To analyze which level of performance and productivity can be reached using different solutions based on, e.g., pthreads is left for future works. The API development is still in progress, and there is room for many improvements:

- Others stages could be added in the pipeline to perform optional tasks: for example, a web service search stage which would aim to find specific web services in several UDDI registries, or an XML data compression stage for reducing the network load.
- The stage load between is not necessarily balanced. For example, if the web service responses are large, the deserialization stage load will be higher than the others. The load can be balanced by setting more threads to this stage. However, the problem is that each web service call can be different, thus the load balancing should be changed at each call. One approach could be the

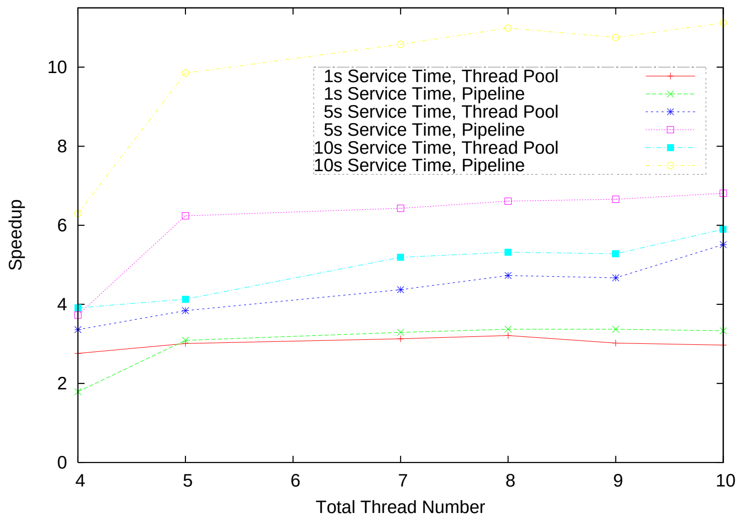


Fig. 7. Speedup for Various Waiting Times and Solutions

load balancing prediction by estimating the response sizes. Another approach could be to consider each couple (web service task, stage) as a global task in a task pool. In this case, the load does not depend on the pipeline stages but on the threads (cores) used by this task pool. Developing and experimenting with these solutions are the object of ongoing work.

References

1. Bull, J., Kambites, M.: JOMP, an OpenMP-like interface for Java. In: Proc. of the ACM 2000 Conf. on Java Grande, San Francisco, San Francisco (2000) 44–53
2. Direction Générale des Impôts: The copernic tax project. (2006) http://en.wikipedia.org/wiki/Copernic_tax_project.
3. Tidwell, D.: Web services, the web's next revolution. In: IBM developerWorks. (2000)
4. Consortium, W.W.W.: Web services description language (wsdl). (2001)
5. Balart, J., Duran, A., Gonzalez, M., Martorell, X., Ayguade, E., Labarta, J.: Experiences parallelizing a web server with OpenMP. In: First International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon (2005)
6. Kut, A., Birant, D.: An approach for parallel execution of web services. In: ICWS'04: Proceedings of the IEEE International Conference on Web Services (ICWS'04), Washington, DC, USA, IEEE Computer Society (2004) 812–813
7. Wurz, M., Schuldt, H.: Dynamic parallelization of grid enabled web services. In: Advances in Grid Computing, EGC 2005. (2005) 173–183
8. Sun Microsystems: Web Services made easier: the Java APIs and architectures for XML. Sun Microsystems (2002) http://java.sun.com/xml/webservices.pdf.