Dynamic Parallelization of Grid–Enabled Web Services*

Manfred Wurz and Heiko Schuldt

University for Health Sciences, Medical Informatics and Technology (UMIT),
Information and Software Engineering Group,
Eduard-Wallnöfer-Zentrum 1, A–6060 Hall in Tyrol, Austria
{manfred.wurz, heiko.schuldt}@umit.at

Abstract. In a grid environment, it is of primary concern to make efficient use of the resources that are available at run-time. If new computational resources become available, then requests shall also be sent to these newly added resources in order to balance the overall load in the system. However, scheduling of requests in a service grid considers each single service invocation in isolation and determines the most appropriate provider, according to some heuristics. Even when several providers offer the same service, only one of them is chosen. In this paper, we provide a novel approach to the parallelization of individual service requests. This approach makes dynamic use of a set of service providers available at the time the request is being issued. A *dynamic* service uses meta information on the currently available service providers and their capabilities and splits the original request up into a set of simpler requests of the same service types, submits these requests in parallel to as many service providers as possible, and finally integrates the individual results to the result of the original service request.

1 Introduction

Grid computing aims to establish highly flexible and robust environments to utilize distributed resources in an efficient way. This can be, for example, computational resources, storage capacity, or various external sensors. An essential feature of grid environments is to make use of the resources that are available at run-time. While data grids focus on the exploitation of storage resources, *service grids* mainly consider computational resources for scheduling. In particular, if new computational resources become available, then requests will also be sent to these newly added resources in order to balance the overall load in the system.

The advanced resource management of service grid infrastructures seamlessly considers web service standards and protocols for making application logic accessible. Web services can be invoked by common web protocols (e.g., SOAP over

* This work has been partially funded by the EU in the 6^{th} Framework Programme under the network of excellence DELOS (contract no. G038-507618) and by the Austrian Industrial Research Promotion Fund (FFF) under the project HGI

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 173–183, 2005.

HTTP) and are described in a platform-independent way by XML and WSDL. All this has led to the recent proliferation of web service technology which has also gained significant importance in the grid computing community.

However, scheduling of (web) service requests in a grid considers each single service invocation in isolation and determines the most appropriate (web) service provider, according to some heuristics that, for example, take into account the current load of all providers of this particular service in the overall system. Although several providers offer the same service or at least semantically equivalent services, only one of them is chosen at run-time and the request is submitted to this provider for processing. This distribution is even independent of the complexity of the service request in question.

In this paper, we provide a novel approach to the parallelization of individual web service requests by making dynamic use of a set of providers of services of the same type which are available at the time the request is being issued. This work specifically focuses on powerful new services using composition, self-adaptability, and parallel service execution. The contribution of this paper is to introduce an architecture of a service seeming to be an ordinary, callable service to the outside world, which is able to adopt its behavior based on some quality of service criteria attached, and the resources available on a grid. In short, this *dynamic* service uses meta information on the currently available service providers and their capabilities (taken from a service repository) and splits the original request up into a set of simpler requests (in terms of the data that has to be processed) of the same service types, submits these requests in parallel to as many service providers as possible, and finally integrates the individual results from these service providers to the result of the original service request.

The following scenario illustrates in which way dynamic services can be used to solve real world problems, and how easily they can be integrated in existing infrastructures. The applicability to large scale digital library systems within a healthcare environment has been presented in [1].

Clustering: *Data mining in high dimensional feature spaces is a commonly used approach to gain new knowledge in medical informatics and bioinformatics. In the field of functional metabolomics, it is, for example, used to support the identification of disease state metabolites without any prior knowledge and permits the construction of classification rules with high diagnostic prediction [2]. In this type of applications, clustering is important to understand the natural structure or grouping in a data set. Clustering, in particular, aims at partitioning the data objects into distinct groups, maximizing the similarity within that group, while minimizing the similarity between groups. Finding clusters in high dimensional feature spaces is a computationally intensive task (more details can be found in [3]). SURFING (SUbspaces Relevant For clusterING) [4], a sample clustering algorithm, computes a distinct quality measure per subspace and then ranks them according to the interestingness of the hierarchical clustering structure they exhibit. In worst case, this algorithm has to consider all* 2^d *subspaces (where d is the dimensionality of the feature space). Using SURFING, the number of relevant subspaces within the whole data set can be significantly reduced:*

for most complex data sets, only 5% *of all* 2^d *subspaces have to be examined [4]. This benefit is achieved by calculating a quality measure based on the k-nearest neighbor distances (k-nn-distances), which however has to be done in* $O(n^2)$ *(n being the number of feature vectors to examine), leading to an overall complexity of* $O(2^d \cdot n^2)$*. Since these* $O(n^2)$ *calculations are independent, they are good candidates for (dynamic) parallelization. Assuming the availability of m instances of a service to calculate k-nn-distances, the total effort for the subspace clustering can be reduced to* $\frac{0{,}05 \cdot O(2^d \cdot n^2)}{m}$ *(ignoring the effort to distribute the data set examined). When knn-distance calculation is available as service by different providers, it is highly beneficial to dynamically incorporate as many instances as possible for improving the complexity of a clustering algorithm.*

In this scenario, we assume that the implementation of the actual service is already completed, and focus on the provision of dynamically adapting services that parallelize execution. These services will provide added value to existing applications which can profit from parallel execution without touching the conscientiously tested business logic itself.

The remainder of this work is organized as follows. Section 2 introduces the concepts and components involved in dynamic call distribution. In Section 3, a concrete implementation is presented and Section 4 provides first experimental results. Related work is discussed in Section 5. Section 6 concludes.

2 A Dynamically Adapting Service for Parallel Execution

The combination of individual, generally usable services to solve complex and specialized problems is of great importance in most applications. These efforts mainly concentrate on abstract workflow definitions which can than be deployed to the grid, and be bound to specific resources at the latest possible time. A particular workflow step, a task that has to be processed, is then always mapped to the service or resource, which best fits the requirements. If more than one service is able to fulfill the requirements, one of them is chosen, following the one or the other load balancing algorithm.

In our proposed architecture, we put the emphasis on improving the usability of a single service, as well as on enabling faster and less error prone development for grid environments. This approach is based on the observation, that following the current proliferation of service oriented architectures, the number of services and service providers in a grid will significantly increase. Especially services which are provider independent and are not bound to special resources can be distributed fast and widely in a grid environment or be deployed numerously on demand.

The task of partitioning request parameters and reintegrating results afterwards is highly application specific and, from our perspective, can not be solved in a generic way. Although we see the potential to identify classes of applications according to the mechanism they partition and reintegrate requests which allows to have pre-built splitter and merger services, an expert in the problem domain

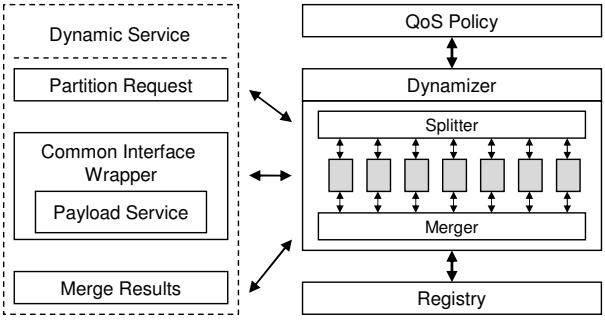

Fig. 1. Overall Architecture

will be necessary to tailor them for the specific need or perform some additional, application domain specific work.

In our approach, splitting requests and merging results, as well as looking up available services in a registry, is transparent to the user. The service that is i.) enhanced with knowledge of how to partition the requested task into subtasks, and ii.) how the partial results can be re-integrated, can still be accessed as before. We call a service enhanced that way a *dynamic service*.

As shown in figure 1, the following logical units can be identified for dynamic services.

The box in the center of the left side, labeled 'Payload Service', represents the actual service. It is responsible for providing application semantics, e.g., a complex computation or a database lookup. This is usually a piece of business logic that has existed beforehand, which is now supposed to be opened to the grid and enabled for parallel execution. To achieve this goal, it is surrounded by another box, labeled 'Common Interface Wrapper', which encapsulates the 'Payload Service' and enhances it with a common interface.

On top, 'Partition Request' encapsulates knowledge on how incoming parameters for the 'Payload Service' have to be partitioned, so that the original request can be decomposed into numerous new 'sub'- requests. Each of these 'sub'- requests can than be distributed on the grid, and be processed by other instances of the originally targeted service. The box at the bottom ('Merge Results') integrates (partial) results returned from the grid to provide the original service requester with a consolidated result. It can therefore be seen as the reverse operation to the 'Partition Request' service. The combination of these elements is referred to as 'Dynamic Service'.

To find the instances of the originally targeted service (e.g., services where the functional description equals the one of the 'Payload Service'), a registry is used (depicted in the lower right corner of figure 1) . This registry provides information on which services are available, how they can be accessed, and what their properties are (in terms of CPU load, connection bandwidth, access restrictions, etc).

The 'Dynamizer', depicted on the right hand side, makes use of the services mentioned above. It glues together the previously described services by making the parallel calls and coordinating incoming results. It has also to provide appropriate failure handling. It is, in contrast to 'Partition Request' and 'Merge Results', application independent and generally usable. The 'Dynamizer' can interact with all services that adhere to a common interface, as ensured by the 'Common Interface Wrapper'. It can be integrated in environments able to call and orchestrate services, or it can be packaged and deployed together with specific services.

To make the best possible use of the 'Dynamizer', the user can send a description of the desired service quality along with the mandatory parameters needed to process the request. In this QoS (Quality of Service) policy, the user can, for example, describe whether the request should be optimized in terms of speed (select high performance nodes, and partition the input parameters accordingly), in terms of bandwidth (try to keep network usage low) or if it should aim for best accuracy (important for iterative approaches or database queries, where there is an option to use different data sources). Since these specifications can be contradictory, adding preferences to rank the users requirements is of importance. To better illustrate the mechanisms within the 'Dynamizer' regarding the user specified QoS policy, we consider the following example: A scientist wants to use the SURFING algorithm as described in section 1 to examine a set of metabolomic data. In the QoS policy file, he specifies that network usage should be kept low (because his department has to pay for each megabyte sent through the wire), and as a second preference to have his call optimized in terms of speed. The 'Dynamizer' has three powerful computational services at hand, which would be able to deliver the result within approximately two hours at the cost of three times transferring the data set, or, alternatively, 400 less powerful services, which would be able to deliver within 20 minutes but at the obvious cost of much higher network usage. The algorithms on how to reconcile the users specifications, the details of the QoS description language and how to integrate this best with our existing implementation is currently under investigation.

3 Implementing Virtual Dynamic Web Services

3.1 Dynamizer

The most vital part, the hub, where all the main control flow, the intelligence and the failure handling, is located, is within the so called 'Dynamizer' (shown in figure 1). It is responsible for issuing the calls in parallel, in our implementation each one in a separate thread, collecting the results, and combining them to match the original request.

If a request to a 'dynamic' service (consisting of the actual service and enhanced with a 'Common Interface Wrapper', the 'Partition Request' and 'Merge Results' services) is issued, it is redirected to the 'Dynamizer'. The registry is queried for a list of available instances that are as well able to process the re-

quest. In case there are any, the original request can be decomposed using the 'Partition Request' service attached to the *'dynamic'* service. In our case, this is implemented as an additional operation within the originally called web service, named *splitParameters*. This operation takes a list of input parameters, the ones specified by the issuer of the original call, along with a parameter indicating how many partitions should be created. The decision on how many partitions should be created is made by the 'Dynamizer', based on information provided by the registry. If the *splitParameter* operation can not produce as many partitions as asked for by the 'Dynamizer', it is empowered to adapt that parameter in favor of producing an error. Along with the partitions of parameters, it creates a re-assembly plan, which is later on used to reconstruct the overall result.

Having a number of available services, as well as the partitioned parameter set at hand, the 'Dynamizer' issues parallel calls to those services, each with one of the partitions as an input. If the number of available services does not match the number of partitions, not all available services are used, or some are used more then once, respectively. Alternatively, a surplus of services can be used to backup others, in case the nodes hosting the services fail or are disconnected (or are known to be less reliable than others).

Finally, the (partial) results returned are integrated using the service 'Merge Results'. It is, like the *splitParameters* operation, included in the *'dynamic'* service. In our implementation, it was realized as an operation named *mergePartitialResult*. It accepts as an input a reference to the overall result, the (partial) result returned by one of the service instances called and the re-assembly plan produced during partitioning. According to this re-assembly plan, the partial result is inserted into the final result. When all partial results are returned, the 'Dynamizer' forwards the overall result to the original issuer of the call. Figure 2 shows a sequence diagram to illustrate the call sequence among the described services.

3.2 Dynamic Service

In contrast to the 'Dynamizer' who plays a managing and coordinative role, the *'dynamic'* service exposes a piece of business logic to the grid, enhanced with the possibility to dynamically execute incoming calls in parallel. It achieves that by adding two additional operations, *splitParameters* and *mergePartitialResult*, and interacting with the 'Dynamizer' (as described above). To present all possible *'dynamic'* services to the 'Dynamizer' with one common interface, the actual service is wrapped within an operation we termed *doOperation* in our implementation. In the architectural view depicted in Figure 1, it is labeled with 'Common Interface Wrapper'. It accepts partitions of parameters as an input, can do some type conversions if necessary and maps the partition to the parameters of the actual service.

Revisiting the scenario from chapter 1, the splitting can easily be achieved by assigning each available knn-distance service a fraction of distances to calculate. Let there be m distance calculation services available and a set of n feature vectors, then each calculation service will have to return the knn-distances for

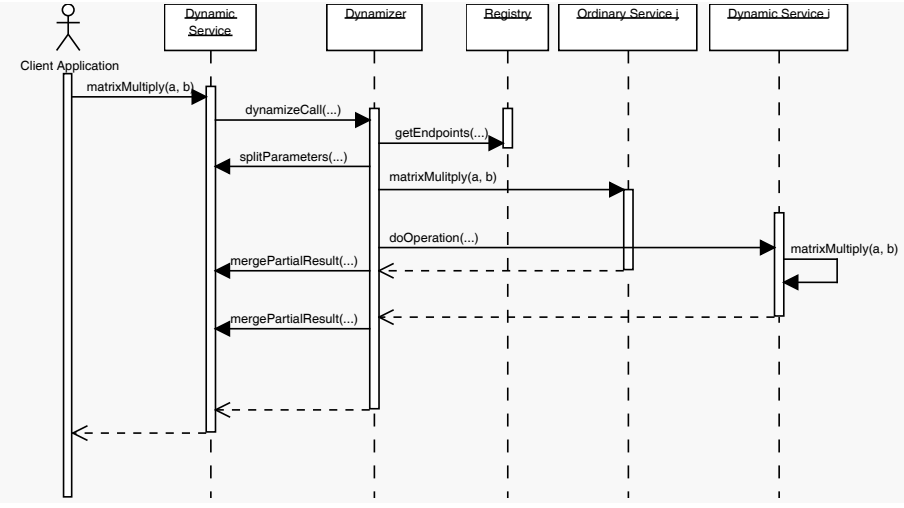

Fig. 2. Sequence Diagram of Service Invokation

n/m feature vectors. This ordered result vectors can then be merged into the overall result vector of n knn-distances.

Depending on the actual service(s) in terms of number of available instances, load, speed of instances available etc., and the QoS policy specified by the user, different kinds of decompositions might be fruitful. If, for example, bandwidth usage has to be kept small or the network used is slow, partitioning might occur differently at the cost of less computational speed. The intelligence about which way of decomposition is best is currently left to the implementation of the 'Split Request' service – but better support by the infrastructure is subject of ongoing research.

3.3 Registry

As registry, any catalog service that is capable of storing information about services available in the grid together with some metadata about their potential behavior can be used. We have implemented a simple registry to store data about the services available in our test bed, but others as GT3's IndexService or a UDDI server could of course be used as well.

4 Experiments and Results

We have implemented a simple matrix multiplication service which can be used for dynamic parallelization. We ran it on varying numbers of personal computers serving as our grid test bed. We have exploited ordinary desktop computers with either Linux or Windows as operating system, and Tomcat with Axis as basis

Table 1. Processing time for multiplication of NxN matrices on up to 4 PC's

	4 Nodes	3 Nodes	2 Nodes	Non Parallel
$N = 60$	1990 ms	2200 ms	2650 ms	3940 ms
$N = 120$	5030 ms	5410 ms	6440 ms	7500 ms
$N = 180$	9580 ms	9720 ms	11390 ms	11250 ms
av. speedup	1.5	1.4	1.2	1.0

for call distribution. The 'Payload Service', the matrix multiplication, was by itself not aware of participation in a distributed infrastructure. It was just an ordinary method implemented in Java, able to multiply two matrices passed in as arguments and return the resulting product.

To enable this method for use through the previously described 'Dynamizer', the two additional services 'Partition Request' and 'Merge Result' have been implemented. To comply with the strategy that the payload services itself should not be changed, all parameter partitions generated by the 'Partition Request' service have to be regular inputs for a matrix multiplication. Therefore, the partitioner cuts the first matrix along its rows into equally sized parts. Assume a matrix having m rows, n columns and k computers offering the multiplication service at the time of partitioning the request, this results in k matrices each having m/k rows and n columns. The second matrix is partitioned analogously. Along with this sub matrices, a description is generated for each partition which stores the information of where to integrate the partial result. The 'Merge Result' service copies the partial results received, according to the re-integration description, into the result matrix.

Table 4 shows the results of multiplying $N \times N$ matrices, consisting of randomly generated integer values out of the set $0, 1$, locally or distributed on up to four PC's using dynamic partitioning and distribution of calculation requests as described. To better shape out the usage of this distribution pattern for computational intensive tasks, the multiplication was interrupted for 50 ms per resulting row.

It can be seen from the results in Table 4 that the speedup by adding nodes to the system has not been tremendous. But we would like to stress that the actual speedup of execution was not the primary goal in this work. Rather, the dynamics of parallel execution, partitioning, and merging have been our main focus. To gain better performance, optimizations can be applied in the algorithm to partition requests. Similarly, the data that has to be transferred can be additionally compressed. The up to four PC's used have been added or removed from the grid without changing a single line of code in our services. Nevertheless, the components adapted themselves automatically to the new environment.

5 Related Work

There are numerous possibilities to decompose an application into smaller parts that can then be executed in parallel. One of the most important and widely

known decomposition pattern is to use a central master coordinating control and data flow, and several slaves executing sub tasks. Although there are other possibilities like divide and conquer or branch and bound, the master/slave paradigm is especially suitable for grid environments [5] and therefore widely used. The master worker tool [6] allows to integrate applications in the grid by implementing a small number of user-defined functions similar to the approach described in this paper, but has a strong focus on problems from the field of numerical optimization [7]. While the master worker tool is tightly integrated in a Globus Toolkit 2 environment, our approach focuses on evolving into a more generally usable framework and is independent of the underlying grid infrastructure.

Similarly, the AppLeS Master-Worker Application Template (AMWAT) [8] offers a mature library to ease the creation of applications which are able to solve a problem by breaking it into subproblems and merging the subproblems results into an overall solution. AppLes emphasizes scheduling and fault tolerance issues. In contrast to the explicit exploitation of AppLeS agents and necessary adaptation of existing applications, we aim to do the parallelization transparently and by wrapping existing code instead of interweaving it. Work with other task parallel models can be found in [9, 10] using divide and conquer mechanisms and [11] for an example of branch and bound type of decomposition.

Using Java [12] to build environments for parallel and distributed environments and research about performance differences to other technologies was, among others, conducted in [13][14]. In [13], the Java Language has been enriched with a set of constructs like remote object creation, remote class loading, asynchronous remote method invocation, and object migration focusing on 'java-only' environments, the evolution of web services enabled us to easily integrate all environments being able to invoke web services. In [14], a good overview on various programming models for parallel java applications can be found.

In addition to the possible registries mentioned in the previous chapters, Miles et. al. describe a service registry which allows to store semantically enhanced services [15]. It extends the standard UDDI interface to provide semantic capabilities by attaching metadata to entities within a service description.

6 Conclusion and Outlook

In this paper, we have presented a novel approach to the dynamic parallelization and automatic adaptation of (web) service calls. Invocations of a service that is extended to be dynamic are split up at run-time into a set of sub-requests which are sent in parallel to different service providers. After execution, the results of the sub-requests are integrated in order to determine the result of the original service call. This allows for the fast development of distributed, standards-based parallel applications. In many cases, it enables parallel execution of readily developed and deployed business logic without changes to existing code.

In further work we plan to implement the assignment of QoS policies to services requests. Currently, we are working on the specification of a language to formulate these policies and on mechanisms that allow to apply these policies,

even in the case of contradictory specifications. The current implementation presented in this paper is searching for identical instances of the same service to distribute a call. An important question in our further work will be to examine ways to extend the search also to semantically equivalent services. Finally, as a next step, we aim to integrate our prototype implementation into OSIRIS (Open Service Infrastructure for Reliable and Integrated process Support) [16], a peer-to-peer process execution engine. This will allow to parallelize not only single service invocations but to consider dynamic parallelization of services in the context of process and workflow execution.

References

1. Wurz, M., Brettlecker, G., Schuldt, H.: Data Stream Management and Digital Library Processes on Top of a Hyperdatabase and Grid Infrastructure. In: Pre-Proceedings of the 6^{th} Thematic Workshop of the EU Network of Excellence DELOS: Digital Library Architectures - Peer-to-Peer, Grid, and Service-Orientation (DLA 2004), Cagliari, Italy, Edizioni Progetto Padova (2004) 37–48
2. Baumgartner, C., Böhm, C., Baumgartner, D.: Modelling of Classification Rules on Metabolic Patterns Including Machine Learning and Expert Knowledge. Journal of Biomedical Informatics, In Press (2004)
3. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Academic Press (2001)
4. Baumgartner, C., Plant, C., Kailing, K., Kriegel, H.P., Kröger, P.: Subspace Selection for Clustering High-Dimensional Data. In: Proc. IEEE Int. Conf. on Data Mining (ICDM'04). (2004)
5. Foster, I., Kesselman, C., eds.: The Grid 2, Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers (2004)
6. Linderoth, J., et al.: An Enabling Framework for Master-Worker Applications on the Computational Grid. In: 9th IEEE Int'l Symp. on High Performance Dist. Comp., Los Alamitos, CA, IEE Computer Society Press (2000) 43–50
7. Anstreicher, K., et al.: Solving Large Quadratic Assignment Problems on Computational Grids. In: Mathematical Programming 91(3). (2002) 563–588
8. Shao, G.: Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources. PhD thesis, University of California - San Diego (2001)
9. Foster, I.: Automatic Generation of Self-Scheduling Programs. In: IEEE Transactions on Parallel and Distributed Systems 2(1). (1991) 68–78
10. v. Nieuwpoort, R., et al.: Efficient Load Balancing for Wide-Area Divide-And-Conquer Applications. In: 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2001) 34–43
11. Iamnitchi, A., et al.: A Problem-specific Fault-tolerance Mechanism for Asynchronous Distributed Systems. In: Int'l Conference on Parallel Processing. (2000)
12. Microsystems, S.: Java Technology. `http://java.sun.com/` (2004)
13. Izatt, M., Chan, P., Brecht, T.: Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. Concurrency: Practice and Experience **12** (2000) 667–685
14. Bull, M., Telford, S.: Programming Models for Parallel Java Applications. Technical report, Edinburgh Parallel Computing Centre, Edinburgh (2000)

15. Miles, S., Papay, J., Payne, T., Decker, K., Mureau, L.: Towards a Protocol for the Attachment of Semantic Descriptions to Grid Services. In: The 2nd European Across Grids Conference, Nicosia, Cyprus, Springer LNCS (2004)
16. Schuler, C., Weber, R., Schuldt, H., Schek, H.J.: Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In: Proceedings of the 2^{nd} International Conference on Web Services (ICWS'2004), San Diego, CA, USA, IEEE Computer Society (2004) 26–34