

PROCESSING SCIENTIFIC APPLICATIONS IN A JINI-BASED OGSA-COMPLIANT GRID

Yan Huang
School of Computer Science, Cardiff University
PO Box 916, Cardiff CF24 3XF
United Kingdom
Email: Yan.Huang @cs.cf.ac.uk

Abstract: This paper describes extensions to SWFL and JISGA, the need for which has become apparent in experimenting them with a variety of scientific applications, particularly in the support of services that incorporate MPI-based parallel processing. Extensions are made to the current SWFL to support messaging passing between parallel processes, allow specified shared variables to be accessed in a mutually exclusive way, and provide “single-to-multiple” data decomposition and “multiple-to-single” data merging mechanisms. As a consequence of these extensions to SWFL, changes need to be made in the capabilities of JISGA. In the last section of the paper, several implementation issues are addressed.

Key words: workflow, Web Services, Grid Services, SWFL, JISGA

Introduction to JISGA

JISGA (Jini-based Service-oriented Grid Architecture) is a lightweight Grid infrastructure based on Jini. It extends a Jini system into an OGSA-compliant system for Grid computing by introducing Web service techniques into the Jini system. A major component of JISGA is a workflow engine that takes an application specified in Service Workflow Language (SWFL) as input, converts it into a Java program, and then handles all aspects of its distributed execution. Figure 1 displays the architecture of JISGA. JISGA processes Grid applications that are composed of interacting services and described in an SWFL application description document. The workflow engine serves as the execution environment for the SWFL-described Grid applications. By using multiple distributed Job Processors and JavaSpaces as shared memory, JISGA supports parallel processing for compute-intensive applications.

In JISGA, a job is submitted by providing the workflow engine with a SWFL document, the input data for the job, and a

flag indicating whether an attempt should be made to exploit parallelism in accessing the services of which the job is composed. The job submission may be blocking or non-blocking. In a blocking call the submitting client must wait for the job to finish before continuing. In the non-blocking case the submitting client does not have to wait for the job and may continue with other tasks. The client can then check later whether the job has completed.

The tasks performed by the workflow engine differ depending on whether the job submitting is flagged as sequential or parallel. In the sequential case the workflow engine uses the SWFL2Java tool to generate the Java code, and then compiles and runs it. In the parallel case, a job partitioned recursively into sequential sub-jobs, based on its workflow model. These sub-jobs are placed into a FIFO queue in which each sub-job represents a part of the original workflow model. It is these sub-jobs that are processed in parallel. The order in which the sub-jobs are queued ensures deadlock-free execution. Job Processors in which a sub-job processing daemon is running on different machines

then extract sub-jobs from the queue and process them.

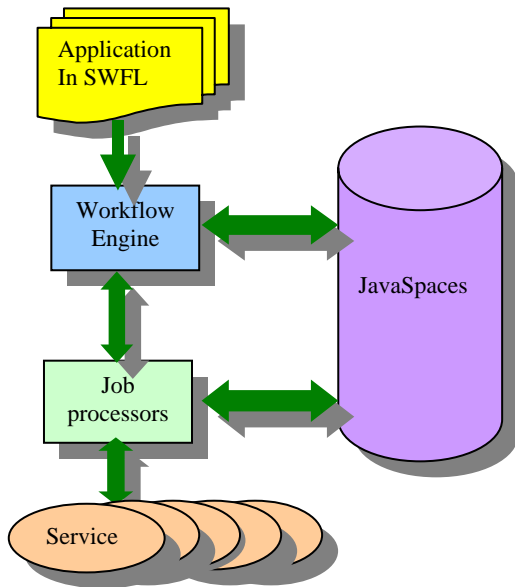


Figure 1: The architecture of JISGA

In the implementation of JISGA, JavaSpaces is used as a distributed shared memory. It is used mainly in processing parallel jobs; however it is also used in both sequential and parallel non-blocking job submission calls to temporarily store the result object that carries the result data of a job. A JavaSpaces Housekeeper service is used to deal with certain tasks relating to the use of JavaSpaces, such as removing objects that have not been recently used, and updating the state of certain objects. The details of JISGA are described in^{1, 2}.

A JISGA system can be pictured as a virtual computer with multiple processors that is composed of multiple distributed machines connected by a network. JavaSpaces works as the memory and the JSHousekeeper works as a kind of memory management unit. In this virtual machine, each JobProcessor can be regarded as a CPU, and the workflow engine provides an interactive interface for the clients to submit jobs. Figure 2 depicts such a virtual machine.

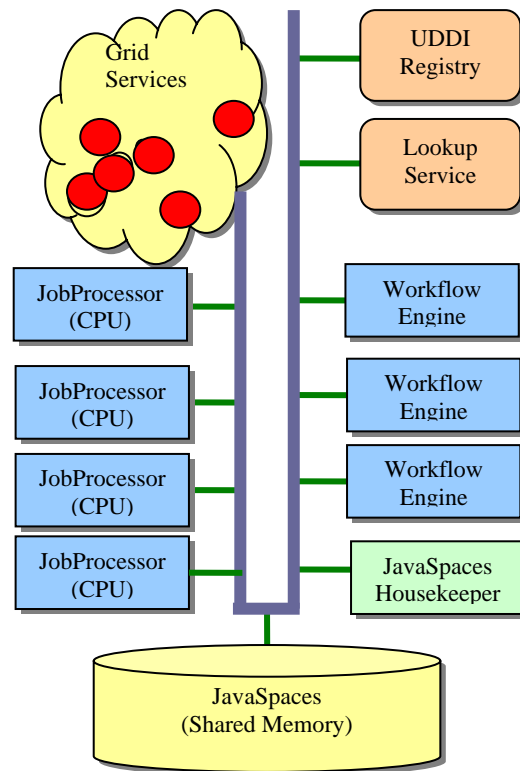


Figure 2: A JISGA Virtual Machine

SWFL and Its Extension

Both Web Service and OGSA systems present an implementation-free interface for their services by encapsulating all of the necessary information for accessing them service in a platform-free, XML-based language such as WSDL³. Applications in are generally service-composite jobs involving interacting services. Such applications can be represented directly in a programming language such as Java and C, or in a standardized, implementation-free and platform-free language, such as XML. Representing such applications in a traditional programming language limits the applications to a range of specific services and a specific time because once the service environment is changed, the corresponding implementation details in the application program have to be changed correspondingly. Representing applications in an XML-document describes an application by giving its workflow structure

and references to the WSDL documents of the services involved in the application. Described in an implementation-free language, the application is not limited to any particular service instance and can be run in any environment where a set of services are available, which implement the interfaces described in the WSDL documents and are referred by the application description document.

JISGA processes Grid applications that are composed of interacting services and described in an SWFL application description document. SWFL⁴ is an XML-based Service Workflow Language and is used to describe composite services and applications in a standardized way. There are two main approaches in describing the workflow in a service-based composite job: a graph-based approach and an execution-order-based approach. In a graph-based approach, a job is represented by specifying the data and control flows between different services in terms of its workflow graph. In an execution-order-based approach, the order of the execution of the services is predefined. However, having an execution-order-based approach, as in XLANG⁵ and BPEL4WS⁶, does not give the same flexibility as having a graph-based approach, as in WSFL⁷, in which the only constraints on the order of execution of services are implicit in the workflow graph. Having a predefined service execution order is not suitable for representing a distributed application, where the ability to dynamically partition and schedule services at runtime is important in order to exploit potential parallelism and to make best use of the available distributed resources. WSFL does allow this capability, and hence provides a flexible and effective basis for representing a Grid application.

SWFL is an extension of WSFL, and the main motivations for developing it were to describe Java-oriented conditional and loop constructs, to permit sequences of more than one service within conditional clauses and loop bodies, and to overcome the limitations inherent in WSFL's data

mapping approach. SWFL extends WSFL in two important ways:

1. SWFL improves the representation of conditional and loop control constructs. Currently WSFL can handle *if-then-else*, *switch*, and *do-while* constructs and permits only one service within each conditional clause or loop body. SWFL also handles *while* and *for* loops, and permits sequences of services within conditional clauses and loop bodies.
2. SWFL permits more general data mappings than WSFL. SWFL can describe data mappings for arrays and compound objects.

In experimenting with SWFL and JISGA for a variety of scientific applications some limitations of SWFL have become apparent, particularly in the support of services that incorporate more sophisticated parallel processing. The original SWFL schema supports parallelism by permitting parallel control and data flow between activities, and has a *parallel* attribute to enable parallelism in processing a *for* loop. However, this definition is not sufficient for describing the complexities of services that communicate via message passing. In the original definition of a *parallel for* loop, there is no mechanism provided to allow message-passing between the parallel processes, which assumes that all the processes are running the same code but on different data with no communication between them. But in a more practical parallel model, special processes are needed for tasks such as broadcasting or merging data, and point-to-point communication between pairs of processes representing for loop iterations should be allowed. In the extended SWFL, a label is specified for each parallel process. Normally, the process is labeled by given a consecutive positive integer starting at "1". By specifying a label for each parallel process, a particular job can be assigned to a particular parallel process to support MIMD (Multiple-Instruction Multiple-Data) parallelism, and a message-passing path can be specified. This extension enables SWFL to describe a variety of scientific applications requiring

parallel processing based on message passing.

Another extension made to the SWFL is to allow user specify shared variables or data. This allows SWFL to support shared-memory applications. A new element type called *Variable* is added to *Flowmodel* in SWFL. A *Variable* element defines a variable by specifying its name, its data type, its initial value and its properties such as *shared/non-shared*. Both *shared* and *non-shared* variables can be accessed by multiple distributed processes, but accessing a shared variable requires mutual exclusion to ensure that only one process can access it at a time.

In order to adequately support SPMD (Single-Program Multiple-Data) parallelism, and allow a data structure to be distributed among processes that execute the same instructions on their parts of the data structure, an extension needs to be made in SWFL to specify if and how a data structure is to be distributed across the processes. The current data mapping mechanism in SWFL does not address the problem of mapping data structures to parallel processes. Hence, a further extension is needed to support this function. Mapping data structures to parallel processes can be implemented in two steps. In the first step, a source data structure is divided into a set of data sub-parts which are parts of the source data structure. Then, in the second step, each of the data sub-part is assigned to each processor to be processed (See Figure 3).

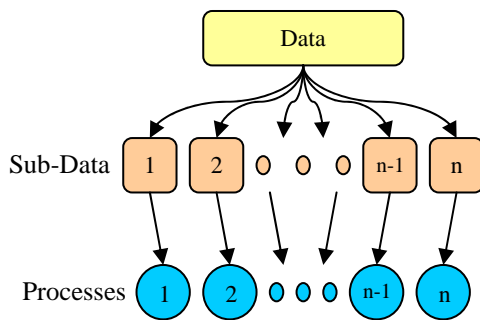


Figure 3: Mapping a data structure to parallel processes.

The data sub-parts from the division of a source data structure are of the same data type. They may be all linear arrays of *double* or all matrices of *int*. The number of the sub-parts may not be fixed because the division of the source data structure is based on how many parallel processors are available. In the original SWFL specification, it is not possible to specify such a data mapping, because the original specification of a *part* element in a *message* allows only reference to an individual variable of a certain data type. It turns out that if there are n data sub-parts of the same type in a message, the part specification has to be repeated n times to complete the specification of the message. Also n has to be a certain number, but in many cases, because the number of the processors may change based on the size of the source data structure, n could be a variable changing each time the application is executed. In the extended version of SWFL, the specification of a *part* element is extended to support the specification of a set of variables of the same data type. An optional “*multiple*” attribute is attached to the *part* element to specify whether it is a single data part or a set of data parts of the same kind. A “*number*” attribute is used to specify the number of the data parts specified in this single specification of the *part* element. A variable can be assigned to the “*number*” attribute.

To implement the first step in mapping a data structure to parallel processes, a more advanced data mapping schema is needed to support the data mapping from a *part* element of a single data structure to a *part* element of a multiple data structures. In the extension, an optional attribute “*type*” is attached to the part mapping element in a SWFL data mapping specification. By specifying “*type=*”*single-to-multiple*”, it distinguishes a single-to-multiple data decomposition mapping from a general single-to-single data mapping and a *multiple-to-single* data merging mapping. In the specification of a “*single-to-multiple*” data mapping, a “*rule*” element is used to specify how a single data structure to be divided in to multiple data structures. An

attribute named “*method*” allows the choice of one of four basic mapping methods: block, cyclic, block-cyclic and customized mappings. A block mapping partitions the source data structure (which is usually an array), into blocks of consecutive entries. A cyclic mapping assigns the first elements of the source data structure to the first target sub-data structure, the second element to the second, and so on in round-robin fashion. A block-cyclic mapping first partitions the source part into blocks of consecutive elements as in the block mapping, the blocks are then mapped to the target sub-parts in the same way that elements are mapped in the cyclic mapping. A “*blocksize*” element is used to specify the size of the blocks in both block and block-cyclic mappings; and a “*targetNum*” element is presented to specify the number of target sub-parts in both cyclic and block-cyclic mappings. A customized mapping is used for irregular data mappings. By specifying which specific range of the source data part maps to which specific target data sub-part, a customized mapping can be specified.

The extended version of SWFL labels data sub-parts with consecutive positive integers starting at “1”. Because both data sub-parts and parallel processes are labelled in the same way, in assigning each of the data sub-parts to a particular parallel process, SWFL could, by default, map a data sub-part to the process with the same label. This saves the effort of specifying the mappings explicitly. However, SWFL must support a mechanism to specify a customized mapping of data sub-parts to processes to allow a specific process to handle a specific data part. This may be required by MIMD (Multiple-Instruction Multiple-Data) processing, in which the parallel processes execute different sets of instructions. In this case, each process is specified in a separate element called “*process*” with a different “*label*”, and has its own *input* and *output* message elements. In the specification of the *input* message, the data sub-part that is assigned to the process can be specified by its *part* name and its *label* value.

The last issue addressed here is the data merging support in SWFL. Data merging is required mostly in SPMD processing. Because all the parallel processes execute the same code, the output data of each process should have the data structure and data type, and these normally need to be merged back into one single data structure to be used by the following activities (see Figure 4). One of the possibilities for data merging is data reduction in which all the data sub-parts are combined using a binary operation such as addition, multiplication, max, min, logical and, etc. Another possibility of the data merging is combining a data sub-parts into a larger data structure by allocating each element in the sub-parts to a specific location in the target data structure.

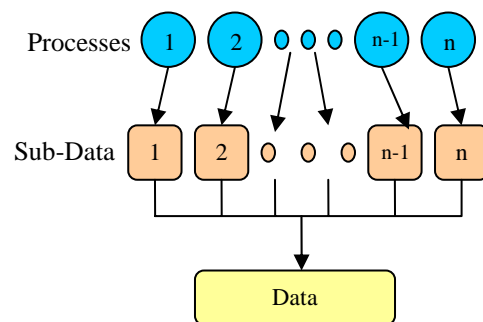


Figure 4: Data merging

Data merging is of type “*multiple-to-single*” in specifying its part data mapping. It can be as the inverse of the “*single-to-multiple*” data mapping. Data reduction mappings are also classified as “*multiple-to-single*” mappings. So in addition to the mapping method choices: block, cyclic, block-cyclic and customized, the “*multiple-to-single*” data mapping supports a “*reduction*” data mapping. An “*operation*” element is used to specify the reduction operation.

Implementation Issues

As a consequence of the extensions to SWFL described above, changes have been made in the capabilities of JISGA. Two

mechanisms are provided for the processing of scientific parallel applications. One way is to make use of an existing MPI implementation, such as MPICH. An MPI service is used to accept jobs that require MPI parallel processing, create machine-dependent MPI parallel code, compile it, and then run it in a particular MPI environment. The other mechanism uses the original parallel processing features provided in JISGA. In this scenario, a job is first partitioned recursively into sequential sub-jobs. These sub-jobs are placed into a FIFO queue. Job processors running on different machines then extract sub-jobs from the queue and process them in parallel by using the SWFL2Java tool to generate the Java harness code for the sub-job, which is then compiled and executed. To support MPI message passing, the processes created for each parallel sub-job have an identifying label attached to them that is used to identify the source and destination of any point-to-point messages.

In the current implementation of JISGA, the sub-jobs are treated the same way no matter whether they are general sub-jobs or identical sub-jobs which have the same sequence of instructions but processes different input data. Because each sub-job only represents an individual job, processing a SPMD parallel application will put a number of descriptions of identical sub-jobs into Sub-Job Queue. This is ineffective because it increases the network traffic by repeatedly writing the same document to JavaSpaces, and wastes the space used by JavaSpaces. Also, no guarantee could be made to ensure the sub-jobs of the same SPMD parallel block are executed synchronously. To overcome these limitations related to parallel processing, changes have been made to JISGA to allow multiple identical sub-jobs to be represented in a single sub-job description and submitted to the Sub-Job Queue. To distinguish this kind of sub-job from ordinary sub-jobs, we called a sub-job representing a set of identical parallel sub-jobs an “*m-sub-job*”. The Job Processor that takes an *m-sub-job* from the Sub-Job Queue to execute will play the role as the master

process in the parallel processing for the *m-sub-job*. It accomplishes the *m-sub-job* in the following steps.

1. It searches for available Job-processors by sending a query message and then wait for the replies.
2. Once a minimal required number of Job-processors are ready, it assigns sub-jobs to those Job-Processors and keeps a record (saved in a HashMap) of which sub-job has been assigned to which Job-Processor.
3. In each Job-Processor that has taken one of the sub-jobs to execute, once the job has completed, its result will be send to JavaSpaces, and a “*successful*” message will be sent to the master process to inform it which sub-job has successfully finished.
4. If in any case, a specific “*successful*” message hasn’t been received in a specified time period, the master process will assume the processing of the specific sub-job failed, and then a new assignment will be made.

As for the supporting of shared/non-shared data variables, only one copy of the shared data is allowed in JavaSpaces and only “*take*” and “*write*” operations are allowed on it. By strictly applying these rules, mutual exclusion is efficiently supported to ensure a shared data is only accessed by one process at a time.

As for data decomposition and data merging, a special sub-job is allocated to accomplish these tasks. It is treated in the same way as any other sub-job – it is placed into the Sub-Job queue and waits for the next available Job-Processor to execute it.

Some applications have been chosen to run in the JISGA environment to demonstrate the functionalities of JISGA. One of them is a mathematical application involving a large-scale numerical linear algebra computation. It solves a set of loosely coupled dense linear systems by using the Parallel Diagonal Dominant (PDD) algorithm, and the application is presented as one of the demonstrations in the Welsh

e-Science Centre booth at the All-Hands Meeting.

References:

¹ Yan Huang. JISGA: A Jini-Based Service-Oriented Grid Architecture. In The International Journal of High Performance Computing Applications, 17(3):317-327, Fall 2003. ISSN 1094-3420

² Yan Huang. The Role of Jini in a Service-Oriented Architecture for Grid Computing. Ph.D. thesis. June 2003. School of Computer Science, Cardiff University

³ E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. Web Service Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>. March 2001

⁴ Yan Huang and David W. Walker. Extensions to web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid. In Proceedings of Computational Science-ICCS 2003. Part 3:254-263. LNCS 2659

⁵ S. Thatte. XLANG: Web Services for Business Processing Design. Available at http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

⁶ F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte and S. Weerawarana. Business Process Execution Language for Web Services. Available at <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>, August 2002.

⁷ F. Leymann. Web Service Flow Language (WSFL 1.0). Available at <http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, May 2001.