A Hierarchical Process Execution Support for Grid Computing

Fábio R. L. Cicerre
Institute of Computing
State University of Campinas
Campinas, Brazil
fcicerre@ic.unicamp.br

Edmundo R. M. Madeira
Institute of Computing
State University of Campinas
Campinas, Brazil
edmundo@ic.unicamp.br

Luiz E. Buzato
Institute of Computing
State University of Campinas
Campinas, Brazil
buzato@ic.unicamp.br

ABSTRACT

Grid is an emerging infrastructure used to share resources among virtual organizations in a seamless manner and to provide breakthrough computing power at low cost. Nowadays there are dozens of academic and commercial products that allow execution of isolated tasks on grids, but few products support the enactment of long-running processes in a distributed fashion. In order to address such subject, this paper presents a programming model and an infrastructure that hierarchically schedules process activities using available nodes in a wide grid environment. Their advantages are automatic and structured distribution of activities and easy process monitoring and steering.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Design, Performance, Management, Algorithms

Keywords

Grid computing, process support, distributed middleware

1. INTRODUCTION

Grid computing is a model for wide-area distributed and parallel computing across heterogeneous networks in multiple administrative domains. This research field aims to promote sharing of resources and provides breakthrough computing power over this wide network of virtual organizations in a seamless manner [8]. Traditionally, as in Globus [6], Condor-G [9] and Legion [10], there is a minimal infrastructure that provides data resource sharing, computational resource utilization management, and distributed execution.

Specifically, considering distributed execution, most of the existing grid infrastructures supports execution of isolated tasks, but they do not consider their task interdependencies as in processes (workflows) [12]. This deficiency restricts better scheduling algorithms, distributed execution coordination and automatic execution recovery.

There are few proposed middleware infrastructures that support process execution over the grid. In general, they model processes by interconnecting their activities through control and data dependencies. Among them, WebFlow [1] emphasizes an architecture to construct distributed processes; Opera-G [3] provides execution recovering and steering, GridFlow [5] focuses on improved scheduling algorithms that take advantage of activity dependencies, and SwinDew [13] supports totally distributed execution on peer-to-peer networks. However, such infrastructures contain scheduling algorithms that are centralized by process [1, 3, 5], or completely distributed, but difficult to monitor and control [13].

In order to address such constraints, this paper proposes a structured programming model for process description and a hierarchical process execution infrastructure. The programming model employs structured control flow to promote controlled and contextualized activity execution. Complementary, the support infrastructure, which executes a process specification, takes advantage of the hierarchical structure of a specified process in order to distribute and schedule strong dependent activities as a unit, allowing a better execution performance and fault-tolerance and providing localized communication.

The programming model and the support infrastructure, named *Xavantes*, are under implementation in order to show the feasibility of the proposed model and to demonstrate its two major advantages: to promote widely distributed process execution and scheduling, but in a controlled, structured and localized way.

Next Section describes the programming model, and Section 3, the support infrastructure for the proposed grid computing model. Section 4 demonstrates how the support infrastructure executes processes and distributes activities. Related works are presented and compared to the proposed model in Section 5. The last Section concludes this paper encompassing the advantages of the proposed hierarchical process execution support for the grid computing area and lists some future works.

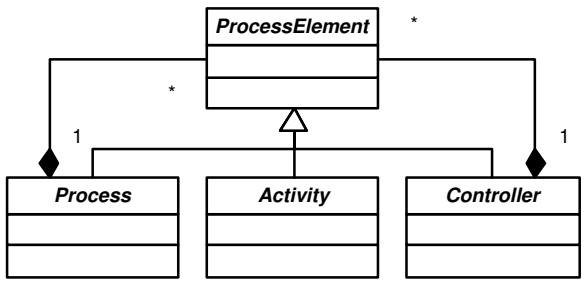

Figure 1: High-level framework of the programming model

2. PROGRAMMING MODEL

The programming model designed for the grid computing architecture is very similar to the specified to the Business Process Execution Language (BPEL) [2]. Both describe processes in XML [4] documents, but the former specifies processes strictly synchronous and structured, and has more constructs for structured parallel control. The rationale behind of its design is the possibility of hierarchically distribute the process control and coordination based on structured constructs, differently from BPEL, which does not allow hierarchical composition of processes.

In the proposed programming model, a process is a set of interdependent activities arranged to solve a certain problem. In detail, a process is composed of activities, subprocesses, and controllers (see Figure 1). Activities represent simple tasks that are executed on behalf of a process; subprocesses are processes executed in the context of a parent process; and controllers are control elements used to specify the execution order of these activities and subprocesses. Like structured languages, controllers can be nested and then determine the execution order of other controllers.

Data are exchanged among process elements through parameters. They are passed by value, in case of simple objects, or by reference, if they are remote objects shared among elements of the same controller or process. External data can be accessed through data sources, such as relational databases or distributed objects.

2.1 Controllers

Controllers are structured control constructs used to define the control flow of processes. There are sequential and parallel controllers.

The sequential controller types are: `block`, `switch`, `for` and `while`. The `block` controller is a simple sequential construct, and the others mimic equivalent structured programming language constructs. Similarly, the parallel types are: `par`, `parswitch`, `parfor` and `parwhile`. They extend the respective sequential counterparts to allow parallel execution of process elements.

All parallel controller types fork the execution of one or more process elements, and then, wait for each execution to finish. Indeed, they contain a *fork* and a *join* of execution. Aiming to implement a conditional join, all parallel controller types contain an exit condition, evaluated all time that an element execution finishes, in order to determine when the controller must end.

The `parfor` and `parwhile` are the iterative versions of the parallel controller types. Both fork executions while the iteration condition is true. This provides flexibility to determine, at run-time, the number of process elements to execute simultaneously.

When compared to workflow languages, the parallel controller types represent structured versions of the workflow control constructors, because they can nest other controllers and also can express fixed and conditional forks and joins, present in such languages.

2.2 Process Example

This section presents an example of a prime number search application that receives a certain range of integers and returns a set of primes contained in this range. The whole computation is made by a process, which uses a parallel controller to start and dispatch several concurrent activities of the same type, in order to find prime numbers. The portion of the XML document that describes the process and activity types is shown below.

```
<PROCESS_TYPE NAME="FindPrimes">
  <IN_PARAMETER TYPE="int" NAME="min"/>
  <IN_PARAMETER TYPE="int" NAME="max"/>
  <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
  <IN_PARAMETER TYPE="int" NAME="numActs"/>
  <BODY>
  <PRE_CODE>
    setPrimes(new RemoteHashSet());
    parfor.setMin(getMin());
    parfor.setMax(getMax());
    parfor.setNumPrimes(getNumPrimes());
    parfor.setNumActs(getNumActs());
    parfor.setPrimes(getPrimes());
    parfor.setCounterBegin(0);
    parfor.setCounterEnd(getNumActs()-1);
  </PRE_CODE>
  <PARFOR NAME="parfor">
    <IN_PARAMETER TYPE="int" NAME="min"/>
    <IN_PARAMETER TYPE="int" NAME="max"/>
    <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
    <IN_PARAMETER TYPE="int" NAME="numActs"/>
    <IN_PARAMETER
      TYPE="RemoteCollection" NAME="primes"/>
    <ITERATE>
      <PRE_CODE>
        int range=
          (getMax()-getMin()+1)/getNumActs();
        int minNum = range*getCounter()+getMin();
        int maxNum = minNum+range-1;
        if (getCounter() == getNumActs()-1)
          maxNum = getMax();
        findPrimes.setMin(minNum);
        findPrimes.setMax(maxNum);
        findPrimes.setNumPrimes(getNumPrimes());
        findPrimes.setPrimes(getPrimes());
      </PRE_CODE>
```

```
      <ACTIVITY
        TYPE="FindPrimes" NAME="findPrimes"/>
    </ITERATE>
  </PARFOR>
  </BODY>
  <OUT_PARAMETER
    TYPE="RemoteCollection" NAME="primes"/>
</PROCESS_TYPE>

<ACTIVITY_TYPE NAME="FindPrimes">
  <IN_PARAMETER TYPE="int" NAME="min"/>
  <IN_PARAMETER TYPE="int" NAME="max"/>
  <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
  <IN_PARAMETER
    TYPE="RemoteCollection" NAME="primes"/>
  <CODE>
    for (int num=getMin(); num<=getMax(); num++) {
      // stop, required number of primes was found
      if (primes.size() >= getNumPrimes())
        break;
      boolean prime = true;
      for (int i=2; i<num; i++) {
        if (num % i == 0) {
          prime = false;
          break;
        }
      }
      if (prime) {
        primes.add(new Integer(num));
      }
    }
  </CODE>
</ACTIVITY_TYPE>
```

Firstly, a process type that finds prime numbers, named `FindPrimes`, is defined. It receives, through its input parameters, a range of integers in which prime numbers have to be found, the number of primes to be returned, and the number of activities to be executed in order to perform this work. At the end, the found prime numbers are returned as a collection through its output parameter.

This process contains a `PARFOR` controller aiming to execute a determined number of parallel activities. It iterates from `0` to `getNumActs() - 1`, which determines the number of activities, starting a parallel activity in each iteration. In such case, the controller divides the whole range of numbers in subranges of the same size, and, in each iteration, starts a parallel activity that finds prime numbers in a specific subrange. These activities receive a shared object by reference in order to store the prime numbers just found and control if the required number of primes has been reached.

Finally, it is defined the activity type, `FindPrimes`, used to find prime numbers in each subrange. It receives, through its input parameters, the range of numbers in which it has to find prime numbers, the total number of prime numbers to be found by the whole process, and, passed by reference, a collection object to store the found prime numbers. Between its `CODE` markers, there is a simple code to find prime numbers, which iterates over the specified range and verifies if the current integer is a prime. Additionally, in each iteration, the code verifies if the required number of primes, inserted in the `primes` collection by all concurrent activities, has been reached, and exits if true.

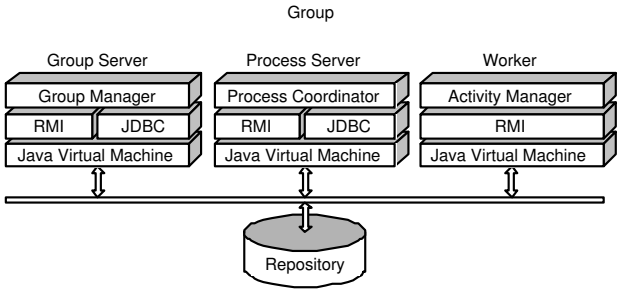

Figure 2: Infrastructure architecture

The advantage of using controllers is the possibility of the support infrastructure determines the point of execution the process is in, allowing automatic recovery and monitoring, and also the capability of instantiating and dispatching process elements only when there are enough computing resources available, reducing unnecessary overhead. Besides, due to its structured nature, they can be easily composed and the support infrastructure can take advantage of this in order to distribute hierarchically the nested controllers to different machines over the grid, allowing enhanced scalability and fault-tolerance.

3. SUPPORT INFRASTRUCTURE

The support infrastructure comprises tools for specification, and services for execution and monitoring of structured processes in highly distributed, heterogeneous and autonomous grid environments. It has services to monitor availability of resources in the grid, to interpret processes and schedule activities and controllers, and to execute activities.

3.1 Infrastructure Architecture

The support infrastructure architecture is composed of groups of machines and data repositories, which preserves its administrative autonomy. Generally, localized machines and repositories, such as in local networks or clusters, form a group. Each machine in a group must have a Java Virtual Machine (JVM) [11], and a Java Runtime Library, besides a combination of the following grid support services: group manager (GM), process coordinator (PC) and activity manager (AM). This combination determines what kind of group node it represents: a group server, a process server, or simply a worker (see Figure 2).

In a group there are one or more group managers, but only one acts as primary and the others, as replicas. They are responsible to maintain availability information of group machines. Moreover, group managers maintain references to data resources of the group. They use group repositories to persist and recover the location of nodes and their availability.

To control process execution, there are one or more process coordinators per group. They are responsible to instantiate and execute processes and controllers, select resources, and schedule and dispatch activities to workers. In order to per-

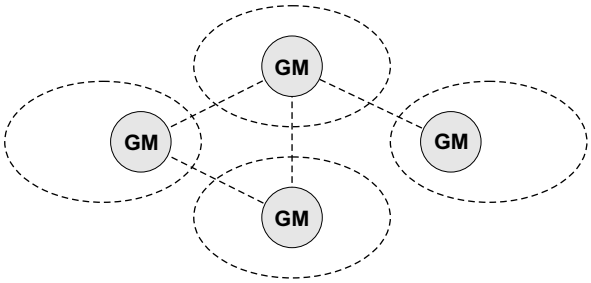

Figure 3: Inter-group relationships

sist and recover process execution and data, and also load process specification, they use group repositories.

Finally, in several group nodes there is an activity manager. It is responsible to execute activities in the hosted machine on behalf of the group process coordinators, and to inform the current availability of the associated machine to group managers. They also have pendent activity queues, containing activities to be executed.

3.2 Inter-group Relationships

In order to model real grid architecture, the infrastructure must comprise several, potentially all, local networks, like Internet does. Aiming to satisfy this intent, local groups are connected to others, directly or indirectly, through its group managers (see Figure 3).

Each group manager deals with requests of its group (represented by dashed ellipses), in order to register local machines and maintain correspondent availability. Additionally, group managers communicate to group managers of other groups. Each group manager exports coarse availability information to group managers of adjacent groups and also receives requests from other external services to furnish detailed availability information. In this way, if there are resources available in external groups, it is possible to send processes, controllers and activities to these groups in order to execute them in external process coordinators and activity managers, respectively.

4. PROCESS EXECUTION

In the proposed grid architecture, a process is specified in XML, using controllers to determine control flow; referencing other processes and activities; and passing objects to their parameters in order to define data flow. After specified, the process is compiled in a set of classes, which represent specific process, activity and controller types. At this time, it can be instantiated and executed by a process coordinator.

4.1 Dynamic Model

To execute a specified process, it must be instantiated by referencing its type on a process coordinator service of a specific group. Also, the initial parameters must be passed to it, and then it can be started.

The process coordinator carries out the process by executing the process elements included in its body sequentially. If the element is a process or a controller, the process coordinator can choose to execute it in the same machine or to pass it to another process coordinator in a remote machine, if available. Else, if the element is an activity, it passes to an activity manager of an available machine.

Process coordinators request the local group manager to find available machines that contain the required service, process coordinator or activity manager, in order to execute a process element. Then, it can return a local machine, a machine in another group or none, depending on the availability of such resource in the grid. It returns an external worker (activity manager machine) if there are no available workers in the local group; and, it returns an external process server (process coordinator machine), if there are no available process servers or workers in the local group. Obeying this rule, group managers try to find process servers in the same group of the available workers.

Such procedure is followed recursively by all process coordinators that execute subprocesses or controllers of a process. Therefore, because processes are structured by nesting process elements, the process execution is automatically distributed hierarchically through one or more grid groups according to the availability and locality of computing resources.

The advantage of this distribution model is wide area execution, which takes advantage of potentially all grid resources; and localized communication of process elements, because strong dependent elements, which are under the same controller, are placed in the same or near groups. Besides, it supports easy monitoring and steering, due to its structured controllers, which maintain state and control over its inner elements.

4.2 Process Execution Example

Revisiting the example shown in Section 2.2, a process type is specified to find prime numbers in a certain range of numbers. In order to solve this problem, it creates a number of activities using the `parfor` controller. Each activity, then, finds primes in a determined part of the range of numbers.

Figure 4 shows an instance of this process type executing over the proposed infrastructure. A `FindPrimes` process instance is created in an available process coordinator (PC), which begins executing the `parfor` controller. In each iteration of this controller, the process coordinator requests to the group manager (GM) an available activity manager (AM) in order to execute a new instance of the `FindPrimes` activity. If there is any AM available in this group or in an external one, the process coordinator sends the activity class and initial parameters to this activity manager and requests its execution. Else, if no activity manager is available, then the controller enters in a wait state until an activity manager is made available, or is created.

In parallel, whenever an activity finishes, its result is sent back to the process coordinator, which records it in the `parfor` controller. Then, the controller waits until all activities that have been started are finished, and it ends. At this point, the process coordinator verifies that there is no other process element to execute and finishes the process.

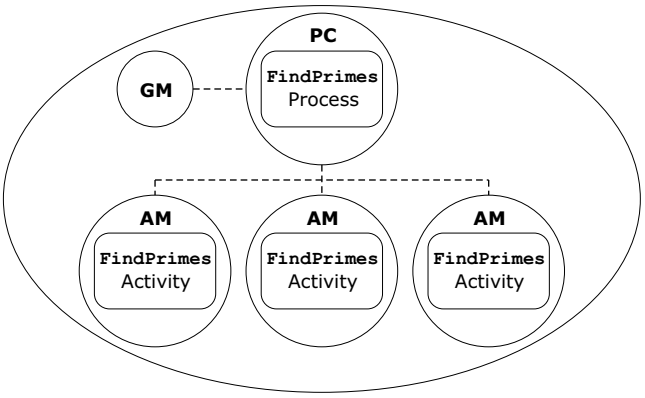

Figure 4: `FindPrimes` process execution

5. RELATED WORK

There are several academic and commercial products that promise to support grid computing, aiming to provide interfaces, protocols and services to leverage the use of widely distributed resources in heterogeneous and autonomous networks. Among them, Globus [6], Condor-G [9] and Legion [10] are widely known. Aiming to standardize interfaces and services to grid, the Open Grid Services Architecture (OGSA) [7] has been defined.

The grid architectures generally have services that manage computing resources and distribute the execution of independent tasks on available ones. However, emerging architectures maintain task dependencies and automatically execute tasks in a correct order. They take advantage of these dependencies to provide automatic recovery, and better distribution and scheduling algorithms.

Following such model, WebFlow [1] is a process specification tool and execution environment constructed over CORBA that allows graphical composition of activities and their distributed execution in a grid environment. Opera-G [3], like WebFlow, uses a process specification language similar to the data flow diagram and workflow languages, but furnishes automatic execution recovery and limited steering of process execution.

The previously referred architectures and others that enact processes over the grid have a centralized coordination. In order to surpass this limitation, systems like SwinDew [13] proposed a widely distributed process execution, in which each node knows where to execute the next activity or join activities in a peer-to-peer environment.

In the specific area of activity distribution and scheduling, emphasized in this work, GridFlow [5] is remarkable. It uses a two-level scheduling: global and local. In the local level, it has services that predict computing resource utilization and activity duration. Based on this information, GridFlow employs a PERT-like technique that tries to forecast the activity execution start time and duration in order to better schedule them to the available resources.

The architecture proposed in this paper, which encompasses a programming model and an execution support infrastructure, is widely decentralized, differently from WebFlow and Opera-G, being more scalable and fault-tolerant. But, like the latter, it is designed to support execution recovery.

Comparing to SwinDew, the proposed architecture contains widely distributed process coordinators, which coordinate processes or parts of them, differently from SwinDew where each node has a limited view of the process: only the activity that starts next. This makes easier to monitor and control processes.

Finally, the support infrastructure breaks the process and its subprocesses for grid execution, allowing a group to require another group for the coordination and execution of process elements on behalf of the first one. This is different from GridFlow, which can execute a process in at most two levels, having the global level as the only responsible to schedule subprocesses in other groups. This can limit the overall performance of processes, and make the system less scalable.

6. CONCLUSION AND FUTURE WORK

Grid computing is an emerging research field that intends to promote distributed and parallel computing over the wide area network of heterogeneous and autonomous administrative domains in a seamless way, similar to what Internet does to the data sharing. There are several products that support execution of independent tasks over grid, but only a few supports the execution of processes with interdependent tasks.

In order to address such subject, this paper proposes a programming model and a support infrastructure that allow the execution of structured processes in a widely distributed and hierarchical manner. This support infrastructure provides automatic, structured and recursive distribution of process elements over groups of available machines; better resource use, due to its on demand creation of process elements; easy process monitoring and steering, due to its structured nature; and localized communication among strong dependent process elements, which are placed under the same controller. These features contribute to better scalability, fault-tolerance and control for processes execution over the grid. Moreover, it opens doors for better scheduling algorithms, recovery mechanisms, and also, dynamic modification schemes.

The next work will be the implementation of a recovery mechanism that uses the execution and data state of processes and controllers to recover process execution. After that, it is desirable to advance the scheduling algorithm to forecast machine use in the same or other groups and to foresee start time of process elements, in order to use this information to pre-allocate resources and, then, obtain a better process execution performance. Finally, it is interesting to investigate schemes of dynamic modification of processes over the grid, in order to evolve and adapt long-term processes to the continuously changing grid environment.

7. ACKNOWLEDGMENTS

We would like to thank Paulo C. Oliveira, from the State Treasury Department of Sao Paulo, for its deeply revision and insightful comments.

8. REFERENCES

[1] E. Akarsu, G. C. Fox, W. Furmanski, and T. Haupt. WebFlow: High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing. In *Proceedings of Supercomputing (SC98)*, 1998.

[2] T. Andrews and F. Curbera. *Specification: Business Process Execution Language for Web Services Version 1.1*. IBM DeveloperWorks, 2003. Available at http://www-106.ibm.com/developerworks/library/ws-bpel.

[3] W. Bausch. *OPERA-G: A Microkernel for Computational Grids*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[4] T. Bray and J. Paoli. *Extensible Markup Language (XML) 1.0*. XML Core WG, W3C, 2004. Available at http://www.w3.org/TR/2004/REC-xml-20040204.

[5] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[6] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. J. Supercomputer Applications*, 11(2):115–128, 1997.

[7] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[8] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[9] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computational Management Agent for Multi-institutional Grids. In *Proceedings of the Tenth Intl. Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE, 2001.

[10] A. S. Grimshaw and W. A. Wulf. Legion - A View from 50,000 Feet. In *Proceedings of the Fifth Intl. Symposium on High Performance Distributed Computing*. IEEE, 1996.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Second Edition edition, 1999.

[12] B. R. Schulze and E. R. M. Madeira. Grid Computing with Active Services. *Concurrency and Computation: Practice and Experience Journal*, 5(16):535–542, 2004.

[13] J. Yan, Y. Yang, and G. K. Raikundalia. Enacting Business Processes in a Decentralised Environment with P2P-Based Workflow Support. In *Proceedings of the Fourth Intl. Conference on Web-Age Information Management (WAIM 2003)*, 2003.