vGrid: A Framework For Building Autonomic Applications

Bithika Khargharia[1], Salim Hariri[1], Manish Parashar[2], Lewis Ntaimo[1], Byoung uk Kim[1]

[1]Autonomic Computing Laboratory
Department of Electrical and Computer Engineering
The University of Arizona
1230 E. Speedway, Tucson, AZ 85721-0104
Email: {bithika_k, hariri,byoung}@ece.arizona.edu
ntaimo@email.arizona.edu

[2]Applied Software Systems Laboratory
Department of Electrical and Computer Engineering
Rutgers University
94 Brett Road, Piscataway, NJ 08854
Email: parashar@caip.rutgers.edu

Abstract

With rapid technological advances in network infrastructure, programming languages, compatible component interfaces and so many more areas, today the computational Grid has evolved with the potential of seamless aggregation, integration and interactions. This has made it possible to conceive a new generation of realistic, scientific and engineering simulations of complex physical phenomenon. These applications will symbiotically and opportunistically combine computations, experiments, observations, and real-time data, and will provide important insights into complex phenomena. However, the phenomena being modeled are inherently complex, multi-phased, multi-scaled, dynamic and heterogeneous (in time space and state). Furthermore, their implementations involve multiple researchers with scores of models, hundreds of components and dynamic compositions and interactions between these components. The underlying Grid infrastructure is similarly heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. The combination of the two results in application development, configuration and management complexities that break current paradigms that are based on passive components and static compositions. In fact, we have reached a level of complexity, heterogeneity, and dynamism that our programming environments and infrastructure are becoming unmanageable/insecure [3].

In this paper we attempt to explore an alternative programming paradigm and management technique that is based on strategies used by biological systems to deal with complexity, heterogeneity and uncertainty. This approach is referred to as autonomic computing [1]. We discuss key technologies to enable the development of autonomic Grid applications. We also present a middleware architecture that sits on top of the existing Grid middleware, intelligently managing and executing autonomic applications with huge computational requirements over limited Grid resources. We discuss in detail how the proposed vGrid middleware can be used to dynamically control and manage large-scale forest fire simulation.

1. Design Motivations

Let us look at a scenario where an application with huge resource requirements, needs to be executed over limited Grid resources. Our approach to handle such a scenario draws its motivation from Virtual Memory and how it handles huge problem execution with limited available main memory. By injecting autonomous properties into the application, we can ensure smart management of the issues associated with this kind of allocation and de-allocation of resources. We will talk about this more in the subsequent sections of this paper.

Virtual Memory is the simulation of storage space so large that programmers do not need to rewrite programs, and authors do not need to rewrite documents when the content of a program module, the capacity of a local memory or the configuration of a network changes [5]. It is memory that is physically not there but behaves as if it is there. In the world of Virtual Memory, resources are provided on demand taking care of achieving optimum performance without leading to too many “swaps” of the program modules. It has a “working set” to prevent the program from thrashing. It has a number of allocation and de-allocation strategies to handle different scenarios. Conceptually, we could map the concepts of virtual memory to solve our problem at hand, that of bridging the supply - demand gap.

In the perspective of virtual memory, we can think of virtual compute resources that can be allocated to the application for execution. Since the resources are virtual, we can allocate as many of them as required by the application, and have some kind of an intelligent manager that maps these virtual resources to actual physical resources, much like the operating system in the context of virtual memory. We term that central intelligence in our architecture as the vGrid manager. The vGrid manager will be responsible for collecting resource requests from the application, dividing the application into virtual computational units (VCUs) much like the pages and segments in virtual memory, applying allocation strategies to translate virtual resources to physical resources (much like the translation from virtual address space to physical address space for a piece of code) and applying de–allocation strategies to reallocate the same resources to other VCUs (much like the operating system throwing away used pages and allocating new pages to that part of the memory).

The organization of the paper is as follows. In Section 2, we give an overview of a forest fire simulation application that will be used to explain and demonstrate the novel features of the vGrid middleware. In Section 3, we describe in detail the architecture of the vGrid middleware and its main components.

2. Brief Description of Forest Fire Simulation Model

The Forest Fire simulation model has been developed using the concepts of cellular automata.

2.1 Cellular Automata

A cellular automaton is an array of identically programmed automata, or "cells", which interact with one another. Essentially, it is a 1-dimensional string of cells, a 2-D Grid or a 3-D solid and has three important features – state, neighborhood and its program [6]. Just as every living cell contains all of the instructions for its replication and operation [2], each individual cell in a cellular automata can be programmed with a set of rules that define how its state changes in response to its current state and that of the neighbors. Thus by building appropriate rules into a cellular automaton we can simulate many kinds of complex physical behavior, ranging from the motion of fluids governed by the Navier-Stokes equations to outbreaks of starfish on a coral reef [6].

2.2 Description of the Model

This model is developed based on DEVS and Cell-DEVS formalisms [7], [12]. It predicts fire spread (the speed, direction and intensity of forest fire front) as the fire propagates, based on both dynamic and static environmental and vegetation conditions. In addition, with the knowledge of fire spread in a given direction, the model can estimate the time the fire would take to reach a given location. This model considers non-uniform fire spread parameters in order to address the issue of spatial/temporal variability of forest fire propagation variables and follows along the line of work of Vasconcelos [10] that introduces and illustrates the conceptual basis for a discrete event hierarchical modular fire spread model.

In this model, the forest is represented as a 2-D cell-space composed of cells of dimensions l x b (l: length, b: breadth). For each cell there are eight major wind directions N, NE, NW, S, SE, SW, E, W

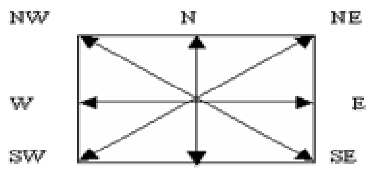

Figure 1: Cell with major wind directions

In our architecture, a group of such individual cells will together constitute, a Virtual Computational Unit (VCU). The weather and vegetation conditions are assumed to be uniform within a cell, but may vary in the entire cell space. A cell interacts with its neighbors along all the eight directions as listed above, using input and output ports (the DEVS-Java model).

A cell is programmed to undergo state changes from "unburned" to "burning" if it is hit by an igniter or gets a notification message to compute its fire-line intensity value. The cell changes state from "unburned" to "burning" only if the computed fire-line intensity is above a threshold value for "burning".

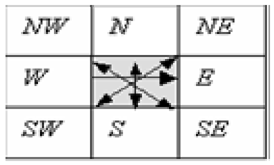

Figure 2: Fire directions after ignition

During the "burning" phase, the cell propagates eight different fire components along the eight directions (refer figure 2 above). The direction and value of maximum fire spread is computed using Rothermel's fire spread model [8]. The remaining seven components are then derived using a different decomposition algorithm. Rothermel's model takes into account, the wind–speed and direction, the vegetation type, the calorific value of the fuel and terrain type in calculating the fire spread.

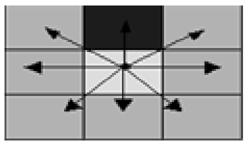

Figure 3: Potential neighbor cells to be ignited by the "burning" center cell

2.3 The Challenge of Parallelizing Cellular Automata

Unlike embarrassingly parallel applications, which are easily distributed on a network, large-scale models of living systems will require ongoing communication between the computers participating in the computation [2]. To parallelize the application that we have discussed above, we need to study and think extensively about the behavior of the application. The operating system utilizes “locality of reference” as one criterion in dividing the huge code into smaller pages or segments. In our design, the initial partitioning of the problem space is performed based on the application specific information provided by the user. After execution has started, the vGrid manager may decide to reconfigure the VCUs to optimize the performance of the application autonomously. Note that this runtime autonomic management of the application is done without any user intervention.

Finally, these VCUs will be scheduled for execution over the Grid, with heterogeneous resources in different domains, geographically dispersed. As explained in the application above, there are a lot of neighbor interactions possibly happening at the same time. So when the VCU’s are scheduled to execute on different machines, the bordering cells in one VCU will have to communicate with the bordering cells in another VCU, physically amounting to exchange of messages between machines on which the VCUs reside. While developing the simulation model above, the amount of communication among neighbors was already reduced by implementing the concept of “transmit when there is a real state change” (DEVS Java approach). The application sees all neighboring cells sitting close to one another as if on a single machine. It is the responsibility of the vGrid manager to route messages between neighboring cells and smartly handle underlying communication and synchronozation issues.

3. vGrid Architecture

An overview of the vGrid middleware architecture is shown in the figure 4 below:

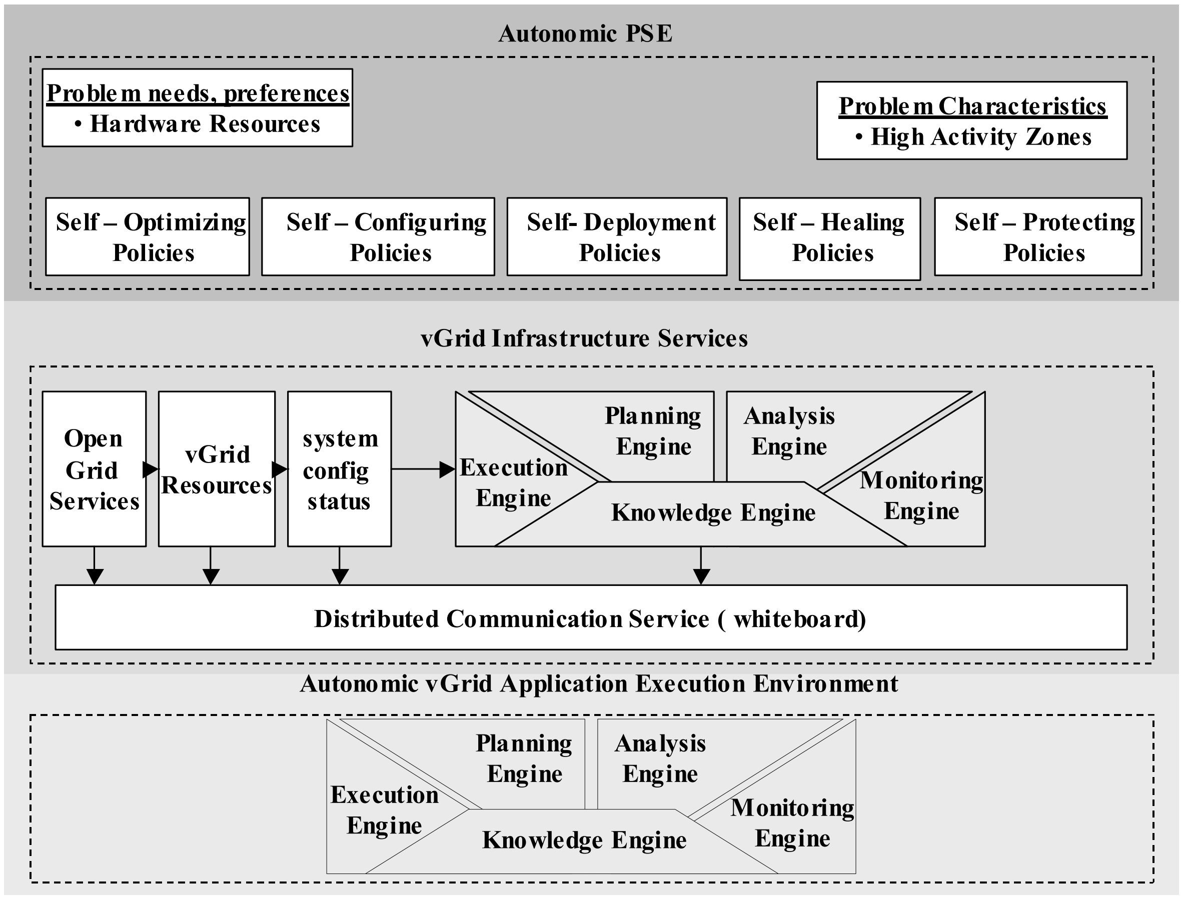

Figure 4: vGrid Middleware Architectural Overview (vGrid Manager Symbol [4]).

3.1 Layer I: Autonomic Problem Solving Environment (PSE)

The autonomic PSE will provide application developers with a general software development environment to design and construct large scale scientific and engineering applications, specify high level policies that capture different aspects of autonomic behavior in the application (like self – optimization, self – healing etc, refer figure 4 above) and submit the application for execution over the Grid.

The general problem-solving environment or Autonomic PSE, that appears as the topmost layer in the vGrid architecture, (refer Figure 4 above), is based on a previous project called Adaptive Distributed Virtual Computing Environment (ADViCE) [9]. The software architecture of the ADViCE problem-solving environment has a web-based graphical user interface that provides users with a set of task libraries to solve one class of problems.

Now let us look at the process of configuring and executing the forest fire application within the autonomic PSE. The application developer will compose the forest fire problem space with the individual cells or fine computational units (FCUs). Within the PSE task library for forest fire simulation application, these FCUs exist as individual modules of code. There exist, within the task library, one FCU of each type (with different forest cell characteristics, different fire spread algorithms etc) .When configuring the application, the user can select FCUs of different types, drag and drop them in the editor area and link them in the manner in which they want to form the entire forest cell space.

A right click on the application brings up a menu bar with items like self-optimizing, self – healing etc. Clicking on the self-optimization menu item for example, will open up a small box where the user can key in the policies for self-optimization of the simulation. Let us assume he key's in –"load balanced execution with minimum execution time". For ease of explanation, we will use this one objective to show how the intelligent middleware operates to self-optimize the simulation as specified by the user.

The PSE also provides appropriate visualization tools for the application developer to view different results of the simulation (like execution time on each machine where parts of the application are running, load on each individual machine, how the fire front is advancing etc) as it proceeds.

Autonomic Components

An autonomic component is a self – governing, self – managing module that will maintain and adjust its operation in the face of changing workloads, demands, hardware failures both innocent and malicious. Our approach here is to develop an autonomic application as a dynamic and opportunistic composition of autonomic components. In the forest fire application, the entire forest is composed of tiny cells or FCUs such that the properties of the forest remain uniform within that cell. A group of FCUs are then managed in a collection called Virtual Computational Unit (VCU), which is the amount of the problem to be given to a single Grid resource unit. The total sets of VCUs make up the complete parallel problem. In our architecture, autonomic properties reside at the level of VCUs, which is nothing but an agglomeration of numerous cells with certain common properties.

Cell or Fine Computational Unit (FCU)

Each individual cell has the following information – data (like threshold value of fire line intensity to change state from "unburned" to "burning", Rothermel's equations, the current state of the cell and how long will it remain in that state), rules for its operation (eg. what happens when a burning cell receives an ignition input from its burning neighbors) and knowledge about its neighbors. Each cell has a unique cell id in the cell space. A cell has input and output ports for communication. Note that a cell or FCU is "atomic" in the sense that we cannot further divide the FCU into smaller units during computations.

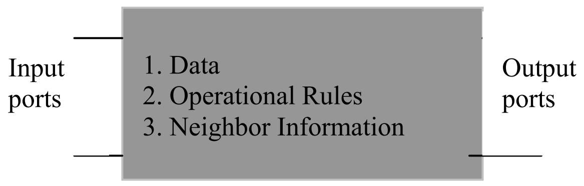

Figure 5: An individual forest cell

Virtual Computational Units (VCUs)

The granularity of an autonomic component lies at the level of a Virtual Computational Unit or VCU. VCUs are capable of exporting their information and policies about their behavior, resource requirements, performance, interactivity and adaptability to the vGrid manager. The vGrid manager uses that information to autonomously change its configuration as and when necessary.

The VCU performs the above mentioned functionalities with the help of an Autonomic Wrapper (AW) associated with the VCU. AW maintains information about the operational, functional and control aspects of the VCU in the following tables:

Os_Table or Operating State Table.

This table maintains the operating state information for the VCU. Just as the operating system saves process state information while doing context switching, the AW will maintain VCU operating state information in the Os_Table. If a VCU gets de-scheduled due to a resource allocation decision by the vGrid Manager for example, its entire status at the time of getting de-scheduled will be updated and saved in the Os_Table. When this VCU gets scheduled later, execution can resume from the point at which this VCU left it. It contains the following information: VCU_id, VCU state (running, ready, blocked, free), value of data structures.

Ci_Table or Control Information Table

This table maintains all control information about the VCU. It contains the following information: VCU Complexity, Block_Time, Total Ex_Time, % CPU utilization, Available Memory.

Fi_Table or Functional Information Table

This table maintains functional information about the VCU like fire spread algorithms etc.

After a configurable number of cycles, this information is made available to the vGrid manager [2]. The vGrid manager uses this information, and decides on appropriate course of action to reconfigure that VCU if needed. We will explain this process in the following sections. The vGrid manager interacts with the VCU through the AW. So configuration changes prescribed by the vGrid manager is communicated to the AW and enforced by the AW on the VCU by using the actuator port. Similarly, the AW uses the sensor port to sense VCU data [2].

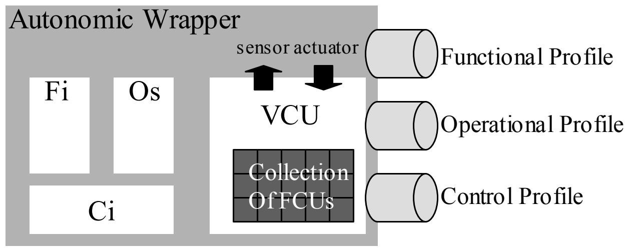

Figure 6: Virtual Computational Unit (VCU) [2] [3]

3.2 Layer II: vGrid Infrastructure Services

The vGrid middleware services implement key enhancements to existing Grid middleware and runtime services to support autonomic Grid Applications [3].

The main components of the vGrid infrastructure architecture include vGrid Manager – Knowledge Engine, Monitoring Engine, Analysis Engine, Planning Engine and Execution Engine, vGrid Resources, Open Grid Services, and Distributed Communication Service. In what follows, we will describe how the simulation application submitted by the user will use the vGrid infrastructure services to use the heterogeneous Grid resources and achieve autonomic runtime management.

3.2.1 vGrid Manager (VGM)

The main administrative component in the vGrid architecture is the vGrid Manager (VGM) which is the autonomic application manager that sets up and configures the application execution environment, manages and controls all the autonomic requirements (e.g., self-optimizing, self-healing, self-configuring, self-protecting, etc.).The vGrid Manager consists of the following five engines:

Monitoring Engine (ME)

Once the user submits the application for execution, the ME reads the application information and generates a cell_properties array for each forest cell constituting the application. The cell_properties array stores information like moisture content, terrain topography, vegetation type, wind speed and wind direction. The ME uses the Grid services (like Metacomputing Directory Service) to generate and update a resource_properties array for each physical resource in the Grid

Analysis Engine (AE)

The AE reads in the cell_properties array and generates a Work Capability Index (WCI) for each cell in the forest cell space. The WCI is a number that indicates the compute intensity of a particular cell within the forest cell space. In addition to the cell properties, the WCI also depends on cell position in the cell space. This means that neighboring cells have more or less the same WCI. This is also justified by reality because properties of the forest only change gradually and there are normally no observable discrete changes within a forest. So WCI generation must respect the natural boundaries. It also generates a Resource Capability Index (RCI) for each available resource in the Grid by reading in information from the resource_properties array.

Planning Engine (PE)

The task of the planning engine is to partition the cells into VCUs by using the WCI as already computed by the AE. While doing this initial partitioning, if the PE discovers WCI that donot conform to geometry of the cell space, it will send an error message to the AE with the cell_id and neighboring cell ids. The AE re-computes the WCI. If it arrives at the same result, it will raise an error condition to the application developer requesting reconfiguration of the application in and around that cell_id. The PE also generates the Working Set of VCUs by reading in fire-spread rules stored in the Knowledge Engine (KE).

Concept of Working Set (WS)

In the world of virtual memory, working set is the set of pages that must be there in memory at any point of time in order to ensure that the program does not thrash. In our forest fire example, the nature of the problem is such that the area of intensive computation is defined by the fire-front. As the fire front advances, all the cells behind the fire front have already burned out and the resources allocated to them can be taken away and allocated to the cells within and ahead of the fire front.

The direction of fire-front depends on initial ignition point and wind direction. The KE already has rules which dictate the direction of fire-front depending on the initial ignition point and wind direction. The PE reads those rules and determines the cells that will be under the fire front initially and subsequently. It finds out the VCU ids for those cells and thus generates the initial WS for the application.

Note that the initial wind direction is capable of changing as the simulation proceeds thereby steering the fire front in a direction different from the originally predicted direction. This situation will be taken care of as the simulation proceeds, from the feedback information received from the AW for each active VCU. Let us assume that the wind direction has changed during simulation and an active VCU sends notification message to an inactive VCU to compute fire-line intensity and check if it can transit from “unburned” to “burning”. Since this VCU is currently inactive, the active VCU will be in blocked state till the message reaches the desired VCU. This VCU has to block because its future state might be affected by inputs coming in from the currently inactive VCU. The Block_Time parameter in the Ci_table will be used by the AE to discover and deduce the bringing in of the currently inactive VCU into the WS.

Execution Engine (EE)

The EE will simply read in VCU ids from the WS and allocate VCUs to physically available resources by a way of mapping the WCI to the RCI. Note that the EE also sends VCUs that are currently not in the WS, to different Grid domains, because these VCUs have been marked as the next potential entries in the WS. The approach to allocate the VCUs to Grid resources is carried out in a two level hierarchy.

Inter-Grid Allocation

At this level, the global EE divides VCUs into several Grid domains, based on the current state of the Grid resources, their availability (RCI) and the pre-determined computational intensity of the VCUs (WCI). The EE picks up VCUs both from the WS and outside the WS and attempts to loosen the communication/synchronization messages between inter-Grid domain resources as much as possible by grouping adjacent VCUs to one Grid domain. The EE creates Ci, Os and Fi tables for each AW associated with a VCU and updates current information in those tables. EE also generates an I_table or interaction table that contains information about VCU location in the Grid. Finally the EE sends these tables for local and remote VCUs together with the VCUs, to that domain.

Intra-Grid Allocation

This allocation is taken care of by the local vGrid manager (EE) responsible for that domain. Within each Grid domain, the number of VCUs allocated to the domain Grid resources are much larger than what can be handled by the existing resources (e.g., say we have 25 Million VCUs to run over 250 computing nodes (128 IBM SP2, 64 Bewoulf cluster, 58 ATM Cluster, etc.). We already have the WCI computed for each VCU that reflects the computational requirements for that VCU. The local EE will simply pick VCUs that were marked to be in the WS and allocate them to the resources by using the WCI and RCI values.

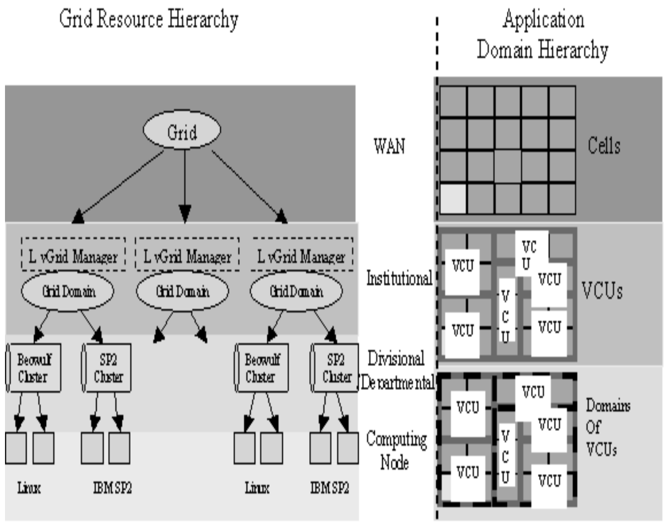

Figure 7: Grid Resource Hierarchy Vs Application Domain Hierarchy

Knowledge Engine (KE)

The KE will contain rules for fire spread based on wind direction and ignition points. At any point of time, the fire front can be predicted by using these rules, the wind direction and ignition point. This knowledge will be used to generate the WS. All high level policies are stored as implementation rules here. For example, a high level self – optimization policy of "load balanced execution with minimum execution time" will be stored here in terms of different load - balancing algorithms (like migration of VCUs, enlarge or compact a VCU etc) and different execution time reduction algorithms (like communication reduction by redistribution of VCUs to resources, overlap communication and computation etc) . It will also contain rules governing resource allocation strategies among VCUs. Let's say, the AE observes that a currently inactive VCU is projected to be active (from the fire spread rules in the KE) in the next iteration and all the resources are currently busy running the other VCUs,. The PE will turn to the KE for rules to allocate resources to this VCU. It could have rules like, de-schedule an active VCU (based on its total execution_time, total block_time or total time_to _completion parameters) and allocated the resource to this VCU, or queue this VCU for execution in an already busy resource till it becomes completely free. Just as the operating system has different de-scheduling algorithms (round robin, elevator, scan etc) for different scenarios, the KE gives estimated / projected execution times, effect on overall performance, load on resource unit etc values for different scheduling/de-scheduling algorithms and this enables the PE to plan the appropriate strategy for a particular scenario.

3.2.2 VCU shifting: Dynamic Problem Partition

To be able to allocate and de-allocate resources depending on the computational intensity and status of available resources, we need to dynamically re-partition the VCUs as the fire front advances and move them among Grid resources as execution proceeds. So, we should have a mechanism to track the activity of the computation in space. Our approach to implement this is as follows.

Once the execution proceeds, the AW keeps updating the Ci and Os tables depending on the execution status of the VCU it manages. AW periodically sends Ci_table and Os_table to the local ME.

As we had mentioned in section 3.1, we will use the self-optimization objective of achieving "load balanced execution with minimum execution time" to explain how the vGrid middleware achieves that objective autonomously without user intervention. Local AE reads Ci_table information, let's say the total execution time parameter from Ci_table and the number of cells in the VCU that are yet to become active. The local AE computes the projected time_to_completion and compares it with the estimated time_to_completion for that VCU. The estimated_time_to_completion, optimal load values etc for this forest fire application are already stored in the KE as obtained from previous runs and from profiling information for this application. It also reads the tables to compare the actual load on that machine with the estimated load. If the AE finds the values to have exceeded the threshold limit, it sends the information to the local PE. The local PE computes the number of cells that need to be removed from the VCU to bring the load and execution time to the estimated values. The local EE checks to see if that plan can be executed within that domain itself. It does that by reading the plans for the other VCUs within that domain as generated by the PE. If it finds it can accommodate that request within that domain, it signals the target AW. The AW then updates the tables, sends information like – affected data structures for those cells to the local EE, deletes those cells from that VCU and updates the tables with the current VCU information. The EE now sends those cells together with the information sent by the old AW, to the selected machine as determined by the PE. The new AW then updates all the tables and starts execution again.

If however, the request cannot be accommodated within that domain, it has to be sent back to the global ME. The global ME will request for execution time and load information from all the domains that have VCUs running at that minute. Once it gets all that information, the global AE and PE together generate plans, either to adhere to the plan sent by the local vGrid manager or generate a new plan of actually moving the entire VCU to a different domain with higher RCI. Note that the local AE and PE make it easier for the global contemporaries to set up plans, but they have no authority in taking decisions outside their own Grid domain. .Since the global ME, AE and PE have the global picture of the entire execution, they are in the best position to decide the correct plan. Next, the global EE simply carries out the plan in a manner similar to the local EE. Of course the I_table might have to be updated by the global EE if the VCU partition or location has changed.

3.2.3 Distributed Communication Service (White Board)

vGrid uses a distributed communication infrastructure with a white board or "Linda" like architecture [11] for coordination of all of the different components. It uses the IBM's TSpaces, which is an intelligent connection-ware component that provides flexible communication, event notification, transactions, and database storage, queries and access control.

Because of its ubiquitous nature (being written in Java), reach-ability characteristics (it uses standard TCP/IP protocols) and loosely connected nature, TSpaces is an ideal middleware component for making network oriented services available to any client, regardless of the computing platform. For example, a service such as printing, email, network fax service or remote device control can be provided in the following way. A client simply sends a message in the form of a tuple to the TSpaces server, specifying the data (e.g. the description of a compute service submitted by an AW) and the service needed (e.g. a problem to run). A service provider application registers an interest with the TSpaces server for all tuples mentioning that particular service. When tuples mentioning that service appear, the TSpaces server notifies the service provider, whereupon the service provider can remove those tuples and process those [2].

Inter VCU Communication

In our forest fire application, when a cell completes "burning" it goes to "burned" state after transmitting the notification message to its neighbors. Communication between neighbors in the same VCU is handled using the output ports of DEVS Java model. Communication between neighbors in separate VCUs (whether within a domain or in different domains) is handled as follows. When cell id 'x' for example, needs to communicate with cell id 'y' and both of them are on two different Grid domains, the Local VGM (EE) will get the request from the AW that it could not resolve that cell id within that site .The Local VGM issues a query for that information on the white board and the VGM responds to that query by writing on the whiteboard, the VCU number for that cell and the corresponding communication server. It uses the I_table to respond to this query. The message is then routed to the appropriate communication server.

3.3 Layer III: Autonomic Grid Application Execution Environment

This layer is responsible for monitoring and controlling the actual execution. In terms of Grid Resource Hierarchy (see Figure 7) this layer exists in each Grid resource domain. The main component in this layer is the Local VGM consisting of the local KE, ME, AE, PE, and EE. It performs the second level of resource allocation and de-allocation (Intra Grid) as mentioned above. It also periodically communicates with the VGM using the whiteboard, where it sends information about the VCUs being executed in its domain. The VGM needs this information from all domains, for autonomously managing the VCUs according to the self-healing, self-optimizing, self-protecting, self – configuring and self – deploying properties specified by the application developer. This has already been explained in section 3.2.2.

4. Conclusion and Current Status

The vGrid architecture is a middleware system that is capable of managing Grid applications according to the autonomic properties specified by the application developer during problem configuration stage. It frees the application developer from the issues related with execution and management of huge applications distributed over heterogeneous Grid resources.It appears to the user like an operating system that promises to run your application irrespective of the application memory requirements.

We are currently developing the conceptual architecture for the system to support autonomic cellular automata type and autonomic adaptive mesh refinement applications (AMR) in science and engineering.

References

[1]P. Horn, "Autonomic Computing: IBM's perspective on the State of Information Technology", IBM Corp., October 2001. http://research.watson.ibm.com/autonomic.

[2]James Kaufman, Toby Lehman,"Optimal Grid: Grid Middleware for High Performance Computational Biology", Research Report, IBM Almaden Research Center. Email:{kaufman,lehman}@almaden.ibm.com.

[3]M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang , M. Parashar, B. Khargharia, S. Hariri, "Automate: Enabling Autonomic Applications On The Grid", Proceedings of Active Middleware Services (AMS) 2003.

[4]Jeffery O. Kephart, David M. Chess, "The Vision of Autonomic Computing", published by IEEE Computer Society, Volume:36, Issue:1,Jan 2003 pp 41 -50

[5] Peter J. Denning, "Before Memory Was Virtual", Draft, June 6th 1996. http://cne.gmu.edu/pjd/PUBS/bvm.pdf.

[6] http://life.csu.edu.au/complex/tutorials/tutorial1.html

[7]B.P. Zeigler, H. Praehofer, T.G. Kim, "Theory of modeling and simulation", 2nd Edition, Academic Press, 2000.

[8]Rothermel, R.], "A mathematical model for predicting fire spread in wildland fuels", Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1972.

[9] H. Topcuoglu, S.Hariri, Wojtek Furmanski, D. Kim, Y. Kim, I. Ra, X. Bing, B. Ye, J. Valente, "A Problem Solving Environment for Network Computing", published by IEEE Computer Society, 1997

[10]J.M. Vasconcelos, "Modeling Spatial Dynamic Ecological Processes with DEVS-Scheme and Geographical InformationSystems, Ph.D. Dissertation, Department Renewable and Natural Resources , University Of Arizona, U.S.A, 1993.

[11]N. Carriero and D. Gelernter, "Linda in Context," Communications of the ACM 32, No. 4, 444-458 (April 1989).

[12]G. Wainer, N. Giambiasi, "Timed Cell-Devs: modeling and simulation of cell spaces", in Discrete Event Modeling and

Simulation: Enabling Future Technologies, Springer – Verlag, 2001.