Virtualization in Parallel Processing

Nachappa A P
Computer Science Department
San Jose State University
San Jose, CA 95192
408-317-8720
nachappa.ap@sjsu.edu

ABSTRACT

We are living in a time of dramatic change in the computing world. In just a few years, computers featuring multiple processors have gone from being a high-end solution to a standard commodity. Still, despite the additional processing power that multiple cores have provided, there are some important questions to consider as we move into the future with this technology.

With an ever-increasing number of cores on a single processor, how do we make use of all of this parallel processing power? How will the engineering systems of the future make use of 16-, 32-, or 64- processor cores effectively? Though parallel programming is one step in the right direction, most applications with a limited number of parallel sections will fail to fully take advantage of processors with many cores. Virtualization – a technology that makes it possible to run multiple operating systems on a single computer – is one solution to these challenges that promises to facilitate more efficient and consolidated designs than previously possible.

With the prevalence of multicore processors in computer systems, many soft real-time applications, such as media-based ones, use parallel programming models to utilize hardware resources better and possibly shorten response time. Meanwhile, virtualization technology is widely used in cloud data centers.

Specific problem solving environments are isolated at the operating system level where real executions are performed in virtualization domain. Virtualization technology helps not only increasing utilization of computing resources but also reducing configuration workload, administrative cost, application porting and energy saving. Multiple commodity operating systems are allowed to share conventional hardware in a safe environment.

This paper aims to cover how virtualization can be of aid in parallel processing and the various paradigms used to parallelize software in virtual environment.

1. INTRODUCTION

With the prevalence of multicore processors in computer systems, many soft real-time applications use parallel programming models to utilize hardware resources better and possibly shorten response time. We shall refer this kind of applications as parallel real-time ones, abbreviated as PRT applications.

At the same time, virtualization technology enables multiple operating systems to run on the same physical machine simultaneously. Hence, more servers use virtualization to host various kinds of applications. Cloud data centers, such as Amazon's Elastic Compute Cloud (EC2), use virtual machines (VMs) to host different applications on the same hardware platform. However, when running in virtualized environment, PRT applications do not behave well and obtain inadequate performance

PRT applications have synchronization requirements due to their parallel feature. In virtualized environment, PRT applications always run in VMs with multiple VirutalCPUs (VCPUs). These VMs will be treated as symmetric multiprocessing (SMP) ones. SMP machines are those where all VCPUs in a SMP VM are usually not online at all times. This is because hypervisors use asynchronous CPU scheduling to manage VCPUs. The asynchronous CPU scheduling cause synchronization problems, such as lock-holder preemption (LHP), which results in a waste of CPU time doing useless work, and which adversely affects the performance of PSRT applications.

Also specifying a parallel program can be treated as a multistage process. Firstly, it involves deciding which actions to be performed in parallel, next deciding which processor will execute each action (and what data will be housed on each processor), followed by deciding the sequence in which each processor will execute the work assigned to it, and lastly, expressing these decisions using primitives provided by the particular parallel machine on which the program will run.

Parallelizing compilers aim at automating the entire process. For large machines and complex applications it has proved to be inadequate. Thus automating the last step, and providing a generic machine independent parallel programming interface.

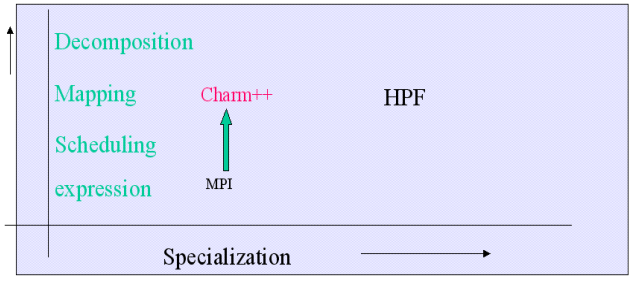

Figure 1: Technical Approach [5]

In the above figure, the programmer specifies the decomposition into parallel parts, while the system maps the parallel parts to processors automatically. We think that this

approach leads to an optimum division of labor between the programmer and the system.

In this approach, the number of parts a computation is broken into is typically not dependent on the number of processors (P), and is often much larger than P. One can consider these parts as virtual processors. Hence this approach can be loosely called virtualization. In the past, we have called it concurrent-object-based approach, message driven programming, or multi–partition decomposition.

2. PARALLEL PARADIGMS

We can categorize parallel architectures into three main classes; namely, shared memory processor (SMP), cluster parallel computers and hybrid. In SMP – also referred to as symmetric multiprocessors – multiple processors have access to a single local memory. Therefore, a program running on a SMP is divided into independent subtasks, or threads, running on multiple cores simultaneously. However, all the threads share the same memory address space and the subtasks communicate by sharing variables in the common memory.

On the other hand, cluster parallel computers consist of multiple independent interconnected computer nodes where each node has its own processor and local memory. Therefore, processes should send messages to each other when they need to exchange variable values. Accordingly, processes must be coded to send and receive messages.

Finally, a hybrid parallel computer consists of interconnected SMPs. Communication of processes within a SMP node is performed by exchanging variable values in the common memory address space and this is usually performed through message passing.

SMPs are most often considered when there is a lot of data dependency between the threads; whereas clusters outperform SMPs in case of little or no data dependencies. On the other hand, clusters are highly scalable; whereas SMPs are restricted to the physical memory size. OpenMP is a standard application program interface (API) which consists of library subroutines and a set of directives. Also, OpenMP is used to execute parallel programs on SMPs. The programmer divides the sequential algorithm into independent tasks, and assigns each subtask to a processor and Message Passing Interface (MPI) is used to communicate between computer nodes in a cluster.

3. VIRTUALIZATION

The basic idea involved in virtualization is to let the programmer divide the program into a large number of parts, independent of the number of processors. The programmer does not explicitly consider processors, but instead they program is considered only in terms of the interaction between these virtual entities. This is illustrated in Figure 2.

Under the hood, the runtime system (RTS) is aware of processors and maps these virtual processors (VPs) to real processors. In particular, it can also change the mapping at runtime, without the user program having to specify it. On the other hand, virtualization provides better resource utilization: virtual machines are installed on an underutilized physical machine to increase the utilization performance. Also, to increase applications flexibility different operating systems may be installed on virtual machines. A virtual machine manager (VMM), or a hypervisor, has the role to manage the virtual machines in terms of the physical resources.

VMM is a layer that separates the virtual machines from the underlying physical hardware. Virtualization also provides more security by protecting the hardware is against any malware that target the virtual machines. This means that the hypervisor takes care of the privileged instructions, such as I/O commands, issued by a virtual machine rather than allowing them to access the hardware directly.

The virtualization can help to achieve uniform resource distribution, uniform management and usage mode, and to improve the performance, programmability, portability and robustness of the system.

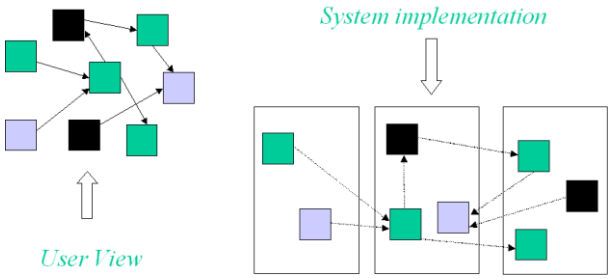

Figure 2: Object-based Parallelization [5]

3.1 The Degree of Virtualization

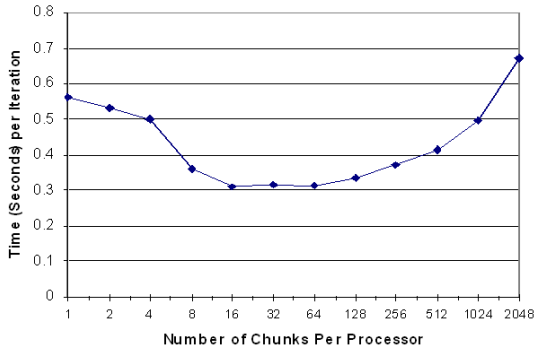

Figure 3: Overhead of Multi-partitioning in application [5]

How can one select the degree of virtualization independent of the number of processors (at least for most applications)? Essentially, the answer lies in the overhead associated with each virtual processor. Scheduling each execution (of a message or method invocation, for example) requires less than a microsecond on today's processors. The amount of computation done with each message must be "substantially larger" than this overhead to justify a lower granularity. This definition does not depend on the number of processors, since the communication overhead as opposed to latency does not depend on the number of processors. Other factors influencing the decision are cache effects. With beneficial result of smaller objects on cache performance, one actually gets better performance with a higher degree of virtualization, instead of being dominated by higher overheads, as shown in Figure 3.

Somewhat more negative are situations where large ghost regions are required around each virtual processor. Here, the degree of virtualization must be constrained by the memory overhead of the extra space also. This situation can be mitigated somewhat by using techniques for storing ghost regions transiently in dynamic storage, and using schemes to fuse the "touching" objects that happen to reside on the same processor during a load balancing era. We next describe Charm++, a C++ based programming system that supports this programming model.

3.1.1 Charm++

The basic unit of parallelism in Charm++ is a C++ object containing methods which may be invoked asynchronously from other such objects. These objects may also reside on other processors. Such objects are called chares in Charm++.

Although important constructs such as specifically shared variables, prioritized execution, and object groups are supported by Charm++, the most important construct in Charm++ is an object array.

A chare-array has a single global ID, and consists of a number of chares, each indexed by a unique index within the array. Charm++ supports 1D through 6-dimensional arrays (sparse as well as dense). In addition, users may define a variety of other index types on their own.

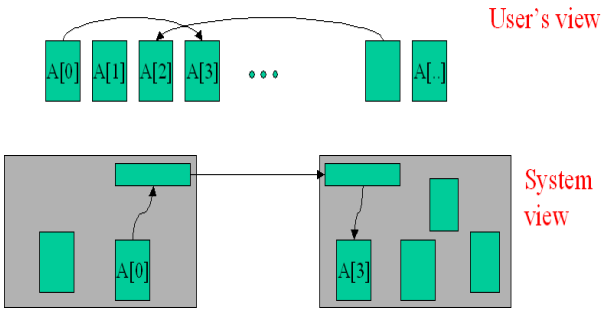

Figure 4: Object Arrays

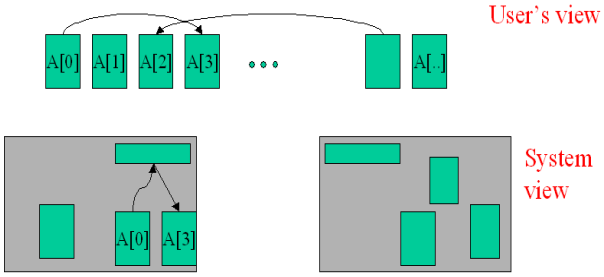

Figure 5: Object Arrays [6]

As shown in Figure 4, the user considers the array as a set of objects, and the code in these objects may invoke methods of other array members asynchronously. However, the system acts as an intermediate stage for communication between the objects. As a result, if the object moves from one processor to another, other objects don't need to know this. The system has automatic mechanisms that efficiently forward messages when needed, and also cache location information so as to avoid forwarding costs in most situations. In addition to method invocation, chare arrays support broadcasts and reductions, which work correctly and efficiently in the presence of dynamic creation and deletions of array members as well as their dynamic migration between processors.

3.1.2 Adaptive MPI (AMPI)

MPI is the prevalent and popular programming model. Although virtualization in Charm++ was useful, the split phase programming engendered by its asynchronous method invocation was difficult for many programmers, especially in Science and Engineering. MPI's model had certain anthropomorphic appeal. The processor sends a message, waits for a message, and does some computation, and so on. The program specifies the sequence in which a processor has to perform these actions.

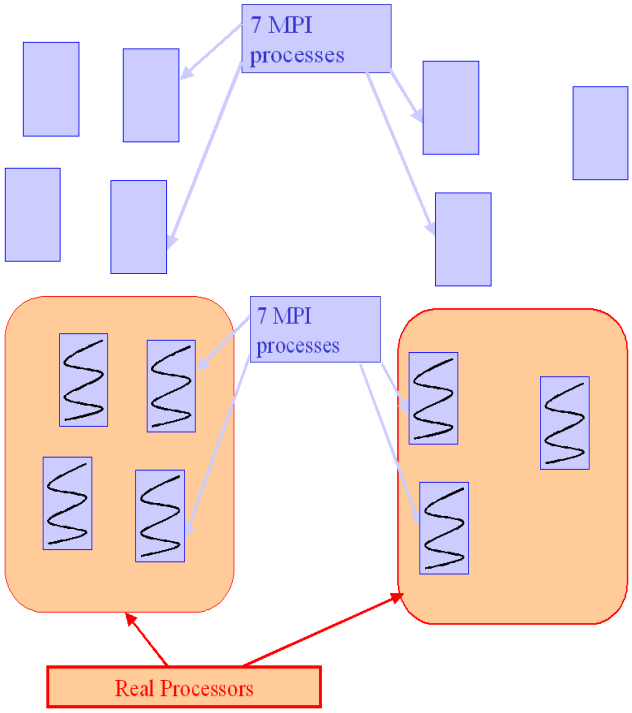

Figure 6: AMPI implements MPI processes as user-level migratable threads [4]

Given this fact, the virtualization aspect of the Charm++ model, was considered and applied to MPI, so MPI programs can be easily converted to our framework. The resultant system is called AMPI. In AMPI, as in Charm, the user programs in terms of a large number of MPI processes, independent of the number of processors. AMPI implements each process as a user-level light-weight and migratable thread as shown in Figure 6. We can have tens of thousands of such MPI threads on each processor, if needed, and the context switching time is of the order of 2-5 microseconds on today's machines. It is technically challenging to migrate user level threads which contain their own stacks and heaps, and which may contain pointers to stack and heap data, but has been efficiently implemented. Migrating existing codes to AMPI requires some simple, mechanical transformations to code because global variables cannot be shared between two threads running on the same processors. AMPI has been used to parallelize several applications, including codes at the DOE

supported Rocket Simulation center at Illinois, and early versions of a Computational Cosmology code.

3.2 Virtualized Self-Adaptive Parallel Programming Model (VAPPM)

High productivity programming model needs to meet the following basic requirements: productivity, scalability and programmability.

Virtualized Adaptive Parallel Programming Model (VAPPM) is the interface between the application development and the architecture of virtualization-based heterogeneous high productivity computers; it is an abstract machine of storage and execution provided to programmers.

With the support of virtualization provided by OS layer, programmers through VAPPM achieve an effective abstraction to conceal the task and data division, mapping between the computations and the processors, communication and synchronization. This mechanism can extricate programmers from architecture details of the system. Programmers can do their jobs in spite of heterogeneous architecture, super-scaled parallelism, automatically parallel optimization, runtime management and fault tolerance, and hand these complex works to compiler and Virtualization-Based Runtime System.

Firstly the, programmers describe the program blocks which can be parallel executed. After the first step, the system, rather than programmers, will appoint the mapping relationships between computing and processors, data storage address and communication and so on. In another words, programmers just need to extract a serial of program blocks which can be parallel executed according to specific application, and assume that each block can be executed by a virtualized processor and don't need to consider processor quantity, memory and topology. The specific execution processor and other things are controlled by Virtualization-Based Runtime System (VRTS). Virtualization-Based Runtime System maintains the system-level resources view and performance model, so it can partition sources and map tasks according to present computing sources condition and specific computing source features of the system. Even more, VRTS can dynamically schedule the mapping relationships during execution. Hence forth, this method obviously shows better performance, scalability and programmability compared to conventional models.

This mechanism can achieve better partition between execution efficiency and programming complexity of parallel programming. Programmers can program according to previous parallel programming concept and experience, and no need to understand the system topology and different processor architecture. Moreover, programming complexity will decrease obviously, for problems such as processor mapping and communication can be ignored.

VAPPM program model is not only compatible with conventional data parallel, but also support parallel tasks. Moreover, with the concept of Domains and Virtualized Processors (VP), Virtualization-Based Runtime System can map between data and processor according to system global view and performance model.

It has the following features:

(1) Global view: Because Virtualization-Based Runtime System maintains system-level view of all the main functional units, programmers can program based on the single system image.

(2) Data parallel: Domains, as primary index, can not only be used to define the array size and value range, but also to support loop and array operation, more important, it can also help to map implicitly between data and processors.

(3) Task parallel: VAPPM doesn't have the concept of threads; it achieves task parallel by parallel execution of sub-computations. Removing the thread concept from programming model can eliminate source management influence, more important, thread management is no longer the interface of code modules. Every module can express the concurrent computations freely according to it's natural behavior. Data consistency can be guaranteed by adding serials of atomic read-write operations, synchronization, and marking a serial of operations as an atomic transaction.

(4) Data affinity scheduling: VAPPM predefines a Locale class in corresponding to a sub-computation, which includes both storage and code instead of bond the computation and processor. Any storage and computation can be related to a Locale object. The data and computation in the same Locale have affinity.

4. ANALYSIS

The main goal of parallel computing is to reduce the execution time and make better resources utilization, thus increasing the overall performance. This allows for the design of huge problems that need intensive computations to be implemented in reasonable time. Sequential programs are studied with the aim of extracting independent tasks that may run concurrently. The greater is the number of processors, the more opportunity is provided to assign independent tasks concurrently and as a result more speedup is achieved. However, concurrency may sometimes impose overhead due to the required synchronization between subtasks. Developers in parallel coding should therefore design their programs in such a way to achieve minimum synchronization overhead. A hypervisor schedules the available physical resources to allocate them to different installed virtual machines. In order to achieve this goal as efficiently as possible many factors are taken into consideration in the scheduling algorithm. Such factors include the number of physical CPUs (pCPUs), whether hyper-threading is applied or not, caches assignment to virtual CPUs (vCPUs) and the ratio of vCPUs with respect to the number of pCPUs. The most efficient scheduling algorithm minimizes the waiting time of virtual machines to allocate the required physical resources. Therefore, it makes a virtual machine unaware of the existence of any competition from other VMs on the available physical resources. Consequently, a VM has the illusion that it is a physical one and it is not forced to wait for a non-idle resource.

More challenges are faced by the hypervisor's scheduler in a multi-core virtualized environment. One of such difficulties is the tendency of a VM to dominate all physical cores in order to complete its job as fast as possible. Hence, all other virtual machines installed on the same physical machine are denied to make use of the available physical resources. In addition, the dynamically changing demand of each virtual machine complicates further the design of the hypervisor scheduler. Therefore, vCPUs are continuously moved around cores during

execution in an attempt to achieve the best possible configuration regarding the allocation of physical resources.

5. METHODOLOGY

In an experiment conducted, a Dell quad-core laptop is to be used; the processor is Intel® Core i7-3610 QM and the cache capacity is less than 10 MB. In order to measure the mutual effect of parallelism and virtualization more accurately, two programs with different criteria are to be tested in order. The first code (A) is characterized by being a computing intensive program whereas the other one (B) uses the memory intensively.

The running speeds of both sequential programs (A) and (B) are measured in a virtualized environment with different number of virtual machines. The execution times are denoted as $T_{I,J,K}$. In this notation I is equal to A or B which is the program name. J is the number of cores which is equal to 1 for the sequential programs. Finally, K denotes the number of installed virtual machines. A hypervisor with the worst performance compared to other known hypervisors is to be used in the experiment. Considering the worst case scenario allows revealing the maximum possible imposed overhead.

OpenMP compiler directives are then to be inserted into the sequential program in order to have two versions of the parallel programs running on a dual-core and quad-core. The execution time is then to be measured for both A' and B', the parallel versions of the sequential programs A and B respectively. Let these be $T'_{I,J,K}$. I, J, and K designate the program name, the number of cores and the number of installed virtual machines respectively. Then, the gained speedup (S) for each case is calculated as shown in formula 1.

$$S_{I,J,K}=T_{I,J,K}/T'_{I,J,K} \quad (1)$$

The same programs are then rewritten using MPI to have two different parallel versions of the programs A and B; let these be A" and B". The same procedure is to be repeated with A" and B" to arrive at a conclusion about the performance of MPI programs in a virtualized environment.

The following points describe the detailed methodology to study the mutual impact of virtualization and parallelism:

• Write a computing intensive sequential program A.

• Develop a sequential program B that accesses the memory heavily.

• Measure the running speed and the memory bandwidth of both A and B in a native non-virtualized environment. Let these be $T_{A,1,0}$ and $T_{B,1,0}$ respectively. Since the programs are sequential, then the number of cores is equal to 1. Since execution is performed in a native environment, then the number of virtual machines is equal to 0.

• Install the virtual machine with worst performance. Measure the running speed and memory bandwidth of both A and B in each case with various numbers of virtual machines. These will be $T_{A,1,X}$ and $T_{B,1,X}$ respectively; where X is the number of the virtual machines. Record the overhead imposed by virtualization in each case, if any.

• Plot the results graphically with the number of virtual machines on the horizontal axis and the throughput/memory on the vertical axis. These graphs depict the overhead imposed by virtualization. Let these be Graphs 1 and 2.

• Insert OpenMP directives into A and B with the aim of being executed on a dual core machine. Let these be $T'_{A',2,X}$ and $T'_{B',2,X}$ respectively. For every number of virtual machines x, measure the corresponding throughput and memory bandwidth. Also, calculate the speedup as shown in formula (1).

• Repeat the previous step for a quad-core machine. These will be $T'_{A,4,X}$ and $T'_{B,4,X}$. Also, calculate the speedup as shown in formula (1).

• Plot the results of the throughput/memory bandwidth of the two previous steps graphically on Graphs 1 and 2 respectively.

• Also, plot the results of the gained speedup for both dual- and quad- cores in both native and virtualized environments; with the Number of cores on the horizontal axis and the Speedup on the vertical axis.

• Repeat the whole experiment with MPI instead of OpenMP.

6. Benefits of Virtualization

Charm++ and AMPI are only examples of the virtualization approach. One can imagine creating virtualized programming models from existing programming models just as AMPI virtualizes the MPI model. In this section we outline a large number of benefits that accrue from virtualization. For ease of discussion, we have grouped them into benefits due to:

6.1 Message Driven Execution

Since there are multiple virtual processors which exist on each processor, it is necessary to have a dynamic scheduler on each processor, as shown in Figure 8. The scheduler works with a pool of messages. It picks the next message from the pool in accordance with priorities associated with messages, if any, identifies the charm++ object it is destined for, and invokes the designated method on it. The method runs to completion and produces other messages (method invocations) for objects on this or another processor. The scheduler then continues by selecting the next message. If the message is related with an AMPI thread, it is inserted in the queue of received messages, and the thread is awakened which would be: marked as ready, and inserted in the scheduler's queue.

Thus, no object (or VP) can hold the processor idle while it is waiting for its message. Instead the object that has a message waiting for it is allowed to continue. This execution style is called message-driven execution (MDE), and leads to several key benefits.

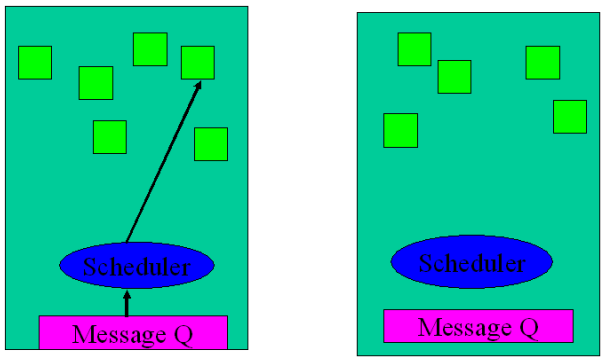

Figure 8: Message Driven Execution [4]

6.2 Principle of Persistence

The Principle of Persistence is a heuristic that can be thought of as the parallel analog of the principle of locality that we discovered while examining the success of our runtime system for dynamic load balancing. Once an application is expressed in terms of its natural objects, as virtual processors, and their interactions, the object computation times and communication patterns, which is number and bytes of messages exchanged between each communicating pair of objects, tend to persist over time.

This rule holds for most parallel applications, including many with dynamic behavior. For instance, some applications such as those using adaptive mesh refinements may involve abrupt but infrequent changes in these patterns. Other applications such as molecular dynamics may involve slow changes in these patterns. In either case, the correlation between recent past and near future holds good, expressed in terms of interacting virtual processors. In rare cases, applications may involve rapid and large changes, which can still be handled by our RTS as best as the application programmer will be able to handle it, by migrating work away from busy processors as they are detected. The obviousness of the principle arises from the fact that most parallel applications are iterative in nature, and the physical system being simulated change gradually, due to numerical constraints.

The real benefit of this principle is that it allows for measurement-based load balancing and optimization algorithms (such as for communication libraries) that can adapt to application behavior.

7. SUMMARY/CONCLUSION

The paper describes the virtualization model, its current state of art in terms of Charm++ and AMPI systems, and applications, and illustrates the benefits of this approach. Virtualization can achieve better software engineering for parallel programs because it helps in organizing applications in terms of the number of physical processors.

Virtualization support leads to message driven execution, which promotes modularity, adaptively overlaps computation and communication and engenders execution predictability. It also allows the RTS to assign and reassign virtual processors (objects or AMPI threads) to processors at runtime, leading to better performance of time-shared clusters, shrinking-and expanding the sets of processors assigned to a job, and support for automatic check pointing.

The Principle of Persistence can be observed to hold for most parallel programs when expressed in terms of their natural pieces (as virtual processors). This leads to measurement based load balancing, and learning algorithms, including communication libraries that can tune their behavior to the application's dynamic and evolving characteristics.

8. REFERENCES

[1] Like Zhou, Song Wu, Huahua Sun, Hai Jin, Xuanhua Shi. "Virtual Machine Scheduling for Parallel Soft Real-Time Applications". 2013 IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems

[2] Hua Cheng, Zuoning Chen, Ninghui Sun. "A Virtualized Self-Adaptive Parallel Programming Framework for Heterogeneous High Productivity Computers". 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications

[3] Soha S. Zaghloul. "The Mutual Effect of Virtualization and Parallelism in a Cloud Environment".

[4] Teng Li, Vikram K. Narayana, Esam El-Araby, Tarek El-Ghazawi. "GPU Resource Sharing and Virtualization on High Performance Computing Systems". 2011 International Conference on Parallel Processing

[5] L. V. Kal´. "The Virtualization Approach to Parallel Programming: Runtime Optimizations and the State of the Art". November 11, 2005

[6] J. E. Moreira and V.K.Naik. "Dynamic resource management on distributed systems using reconfigurable applications". IBM Journal of Research and Development.

[7] Chaves, J.C. "Enabling high productivity computing through virtualization". 2008 DoD HPCMP Users Group Conference

[8] L.V. Kal´e and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor,Proceedings of OOPSLA'93, pages 91–108. ACM Press, September 1993.