JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Evaluation of Cluster Middleware in a Heterogeneous Computing Environment

MASTER'S THESIS

for obtaining the academic title

Master of Science

in

INTERNATIONALER UNIVERSITÄTSLEHRGANG
INFORMATICS: ENGINEERING & MANAGEMENT

composed at ISI-Hagenberg

Handed in by:
Stefan Georgiev, 08024

Finished on:
17 July 2009

Scientific Advisor:
A.Univ.Prof. DI Dr. Wolfgang Schreiner

Hagenberg, July, 2009

Abstract

Deploying a Beowulf-type high-performance cluster is a challenging task. Many problems, regarding the underlying hardware infrastructure and the used software components need to be solved in order to make a set of machines work as a single computer. The task becomes even more complicated when heterogeneous commodity hardware platforms are utilized. This research is provoked by the idea of using a company's old desktop computers for achieving computing performance at a low cost. While the existing cluster middleware provides many solutions for utilizing distributed resources, it still fails to be suitable for some cluster setups. This thesis represents the results of an evaluation in the field of heterogeneous high-performance computing. A systematic analysis of available tools for high-performance clustering aims to reveal both their strong and their weak sides. In addition, a detailed evaluation is provided from the point of view of administrators and users. The thesis describes the process of building a fully-functioning parallel environment using three different tools for cluster deployment - OSCAR v6.0.2, ROCKS v5.1 and CAOS-NSA v 1.0. These tools are compared in detail, so that the most suitable one can be determined. As a result two test environments are built. Numerous examples of tests runs, executed on the two environments, are described. Usability, with regards to different tools like resource managers, schedulers, and MPI, is assessed taking into consideration the underlying hardware platforms. The capabilities of the clusters are evaluated against a parallel application for route optimization.

Contents:

Chapter 1
Introduction

Ever since the invention of the first calculating machines the need for more processing power has been the driving force for further development and innovation. Today's computer hardware evolves rapidly in order to be able to keep up with the extreme demands of the applications and the users. Even though, in the fields of science and research there is still a growing need for more computational power. What is more, professional business applications are becoming more and more advanced and, thus, start requiring more computing resources as well. For years, the solution has been utilization of specialized massive computers that have proprietary hardware and architecture in order to achieve topmost computing performance. However, these machines are still rather costly to be used in everyday business life. An alternative solution, that proves to be also cheaper, is using the joint computing power of collection of more simple computers.

This thesis focuses on studying the capabilities of such architectures. A collection of computers that work together towards achieving a common goal define a parallel environment. The main idea behind parallel environments is to split a heavy computational task into smaller ones and then distribute them to multiple processing units for calculation. This technique sets the foundations of introducing massive speed-ups that can be equal to the number of processors used. An application running concurrently on ten processors can execute ten times faster than on a single one. Today's multi-core processors aim to achieve parallelism by dividing the workload of a single machine between the separate cores. While using a single machine proves to be rather effective, it also suffers from hardware constraints that limit the reachable performance. Thus, an alternative solution is creating a parallel environment that consists of separate machines, each of which has its own processor and memory, connected together via means of networking. This architecture is referred to as a cluster of machines. What brings them together to work as a single computer is the middleware software. It is the gluing component that lies between the operating system and the user applications. In the common case, clusters are completely different from the massively parallel computers that utilize special hardware and interconnection methods between the nodes. What is more, clusters which are built using commodity hardware prove to be quite productive as well. The process of designing, building and using computers for solving extremely advanced computational problems is referred to High-Performance Computing (HPC). Clusters are often used in the area of HPC because they show to be able to provide performance at a lower price. This thesis focuses on analyzing the HPC capabilities of clusters built from commodity hardware machines that are of heterogeneous type. Recent research in the field of HPC shows that there are certain advantages in utilizing different computers in cluster environments. Machines with different processors and different memory capacity show to be more flexible in solving heavy computational tasks and, what is more, they open broader horizons for improvements in the field of parallel programming.

The work in this thesis is inspired by the challenges, which a local company in upper Austria is confronted to. RISC Software GmbH is a company that brings to the market software solutions that combine up-to-date science with most recent research. It is a spin-off of the RISC Institute

(Research Institute of Symbolic Computation). The current research in the area of parallel programming provoked this analysis of clustering technologies and methods. Numerous projects in the company solve advanced computational problems that require vast amount of processing power. In addition, others need to be parallelized in order to achieve reasonable computational time and provide solutions fast. All these, require a parallel environment to be built so that tests runs can be performed. Up to this point, RISC Software GmbH was using remote distributed resources for testing as no alternative was available locally, at the company. What is more, a concrete application dealing with route optimization for industrial logistics was being developed at the time of writing. This application could only be tested effectively on a parallel cluster environment. Building such a testing set-up could not be achieved previously because the crew at RISC Software GmbH was missing necessary experience in the field of clustering. Additionally training was not an option because of the limited size of the development team and their current workload.

Cluster middleware presents numerous challenges that need to be studied previous to deploying a production cluster. This thesis presents a solution for creating a fully-functioning parallel environment by providing a systematic evaluation and comparison of different techniques and tools. Existing achievements in the field of HPC computing are studied. Appropriate tools were chosen for creating a heterogeneous computing environment based on different criteria like, for instance, ease of utilization and provided functionality. Cluster middleware in the form of the cluster deployment tools ROCKS, OSCAR, CAOS-NSA, is thoroughly analyzed. Detailed description of the experience gained during the installation process aims to provide a comparison between these tools and assess their qualities. Furthermore, usability of the achieved environments is tested by evaluating the tools for job distribution, resource management and resource monitoring included in the software of the clusters. Then, the functionality of the achieved clusters is compared using the route-optimization application implemented by RISC Software GmbH. Finally, conclusion are drawn that aim to provide a guideline for creating an effective production cluster for high-performance computing.

The thesis has the following structure. Chapter 2 describes in detail what cluster computing is and what problems it faces when a heterogeneous hardware is used. In addition, an overview of the most recent achievements in the area aims to present different options for clustering tools. In Chapter 3 high-level middleware is studied by focusing on the experience gained from installation and configuration of tools for cluster deployment. Additionally, different techniques for building a cluster are discussed and compared. Chapter 4 provides users with guidelines how to benefit from the tools a cluster provides. Here, series of examples show how both parallel and sequential jobs can be submitted to the cluster. Chapter 5 analyzes the behavior of the built environments when confronted to the challenges of a real-life production application. Different conclusions regarding parallel execution are discussed here. Finally, Chapter 6 summarizes all gained experience, presents conclusions and opened possibilities for further development.

Chapter 2
State of the Art

This chapter describes the state of the art in cluster computing and discusses the tools mostly used for building a heterogeneous cluster environment. Section 2.1 gives a basic overview of clusters and defines some different kinds of clusters. Section 2.2 focuses on programming for clusters, describing some common difficulties. Finally, Section 2.3 is dedicated to cluster middleware, which is described in the frame of four subsections: low level middleware, parallel file systems, high level middleware, and grid middleware.

2.1 Introduction to Clusters

The following section is mainly based on [Buyya vol.1, 1999].

Parallel computers introduce a way to overcome constraints of traditional sequential computers. They divide workload, distribute it among nodes and carry out calculations simultaneously. Large computational problems can be divided into smaller ones, which then can be solved concurrently in an environment of multiple processors. Often, proprietary parallel computers implement such an environment on a single large machine. A simple alternative is to connect multiple single processor machines and coordinate their work. That way similar result can be achieved at a lower cost.

Parallel computers are known to have a number of different architectures [Buyya vol.1, 1999]. The Massively Parallel Processor (MPP) is a large parallel system. Normally, it consists of several hundred processing elements (nodes). The nodes are connected via high-speed network and each of them runs a separate copy of an operating system. MPP implements a “shared-nothing” architecture. Often, each node consists only of main memory and one or more processors. Additional components, like I/O devices, could be added. In contrast, the Symmetric Multiprocessor (SMP) system has a “shared-everything” architecture. Each node can also use the resources of the other nodes (memory, I/O devices). A single operating system runs on all the nodes. Finally, a distributed parallel architecture defines a network of independent machines, usually over a wide geographical area. Each node is a completely separate machine with its own operating system. Furthermore, each node can be of completely different architecture. Any combination of MPPs, SMPs or plain computers could be added to a distributed system.

[Buyya vol.1, 1999] defines “A cluster is a type of parallel or distributed processing system which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource”. A computer node can be a single or multiprocessor system (PC, workstation, or SMP) with memory, I/O facilities, and an operating system. Different terms arise to define different cluster “flavors”: Networks of Workstations (NOW) or Cluster of Workstations (COW), clusters of PCs (CoPs) or Piles of PCs (PoPs).

Originally, clusters were only used for processing computation-intensive tasks in the frame of high-performance computing. Later on, they showed to be useful in other areas as well. Today, except *high performance* (HP) clusters, there are also *high-availability (HA) clusters* and *load-balancing (LB) clusters*.

High-availability (also called *failover*) clusters implement the concept of redundancy [Sloan, 2004]. They are used for mission-critical applications. An example is a web server at a company that must not fail. HA is achieved by having multiple secondary servers that are exact replicas of a primary server. Constantly, they monitor the work of the primary server waiting to take over if it fails. In this basic form, only a single machine (server) is in active use while the remaining ones are in stand-by mode.

Load-balancing clusters provide better performance by distributing workload among nodes in a cluster [Sloan, 2004]. Consider a web server. If load-balancing is implemented, different queries are handed to different nodes for processing. Concurrent processing of queries results in faster overall response time of the server. LB is accomplished by a number of techniques. A simple round-robin algorithm or more complex algorithms that rely on feedback from the individual machines can determine which machine is best suited for handling the next task.

The term "load-balancing" bears different meaning in different scenarios. LB for a high-performance cluster is completely different from LB for a web server. In this thesis we focus entirely on high performance computing utilizing a cluster of commodity-hardware machines. In this case load-balancing is used distributing computational tasks among different nodes in accordance to their processors speed, current load, and memory capacity. The goal is to achieve overall speed-up of tedious calculations.

Clusters and high performance clusters in particular can be composed of computing elements of different ownership and architecture. We distinguish dedicated and non-dedicated nodes, homogeneous and heterogeneous types of nodes. In practice, these two categorizations are tightly dependent because, as it turns out, dedicated nodes are of homogeneous nature and non-dedicated are heterogeneous: in the case of large supercomputers comprised of thousands of dedicated nodes administration is much easier if all nodes are of homogeneous architecture. And, with non-dedicated clusters sometimes it is just not possible to have a network of homogeneous nodes. Nevertheless, the purpose of this study is to show that heterogeneous dedicated nodes prove to achieve better results in the field of high performance computing.

Dedicated nodes of a cluster are devoted entirely to the computational tasks of the cluster [Buyya vol.1, 1999]. They utilize a shared set of resources in order to perform parallel computation across the entire cluster. Usually, nodes are situated in a controlled environment with high-speed interconnections. There is one terminal for the whole cluster.

The alternative, non-dedicated nodes are owned by individuals. Cluster tasks are executed by "stealing" unused CPU cycles [Buyya vol.1, 1999]. Consider an office building or a NOW. Most workstation CPU-cycle capacity remains unused, even during peak hours. Workstation clusters are easier to integrate into existing networks than special parallel computers. They are highly scalable, meaning that new stations can be added any time. Workstation clusters are cheap because they use commodity hardware. Last, but not least, the development tools for workstations are more mature compared to the proprietary solution for parallel computers.

On the other hand heterogeneity of nodes can be defined both, at hardware and software level [Buyya vol.1, 1999]. Often nodes in a company network differ in the utilized CPU types and speeds, memory capacity, network interconnect speed. Also, operating systems may vary. But different types of nodes can perform differently on a set of tasks. Matching tasks to suitable nodes is a problem in such an environment. Other problems regard administration and development. Different hardware architectures define different interfaces and techniques to be managed and programmed for. Additionally networks introduce bottlenecks, as processors are still faster in computations than communications on commodity low-cost networks. Inevitably, latency is introduced by network delays and synchronization in the system.

In this thesis we focus on building a cluster of dedicated heterogeneous workstations. We will show that utilizing such a cluster for enterprise needs proves to be a cost effective alternative for achieving high performance.

2.2 Cluster Software Development - Mapping and Scheduling

Achieving high performance on a heterogeneous cluster requires a lot more than setting up a proper hardware and software environment. Speed-up needs to be achieved in the first place, i.e. a program must execute faster on a high-performance computer than on a sequential one. Utilizing a parallel environment to achieve this requires programs to be parallelized as well, i.e. to be broken down to tasks which can be executed concurrently on different processors. Only when all processors fully utilized, real speed-up is achieved. Before tasks are executed they need to be assigned to processors. The process of assigning tasks to processors is referred to as *mapping* [Buyya vol.1, 1999]. Proper assignment can show great influence on the speed of a parallel program. This is especially true in the case of heterogeneous parallel computers. Take into account the different architecture of CPUs, their different speeds, and interfaces. Some processors are faster in computing floating point operations while others like graphical processors are faster working on vector input data. The problem is to define which processor is most suitable for executing a task in a way that the overall execution of a program is minimized. In contrast, in a homogeneous environment, where all the processors are the same, the problem reduces just to constructing and distributing the tasks.

The problem of mapping complicates furthermore by the need of communication between tasks. As the tasks are a part of one program they need to synchronize and pass values between each other. This is often referred to as communication overhead in parallel programming. Mapping has to take into account the communication load when deciding where to execute tasks. In order to achieve full parallelism, tasks should be spread as evenly as possible and fully exploit all processors. However, distributing the processes among the parallel computer leads to elevating communication between them and thus slows down the whole execution. One solution to this problem is to place tasks that communicate most in one processor or in closely located processors. This way, however, complicates the mapping problem even more as it already goes into conflict with the choice of the most appropriate processor. Hence, mapping has to keep balance between computation and communication load.

On the other hand, scheduling is the process which determines when tasks are executed. [Buyya vol.1, 1999] defines “scheduling is to solve a set of tasks serviced by a set of processors to get the best result according to a certain policy, which can be described in a number of different ways in different fields.” Scheduling techniques are implemented by most modern operating systems today.

The difference is that in this case they focus only on allocating CPU cycles to local processes. In a parallel environment scheduling manages a set of processes that are created and executed on different machines. While mapping determines the best machine for a task, scheduling controls the order of which tasks are executed. Furthermore, it determines if a task should be suspended, moved to another machine, or resumed. All this is done in a way that maximizes the total speed at which parallel computations are carried out. In the general form the scheduling problem is known to be NP complete. However, optimal solutions can be found for a number of situations. Well known scheduling algorithms are FIFO, round robin task distribution, shortest-job-first, and the shortest-job-remaining-time.

According to [Buyya vol.1, 1999] "The processing capacity of a heterogeneous computing system cannot be efficiently exploited unless the resources are properly scheduled". Load balancing is a scheduling technique that takes care of idle CPU cycles. It tries to distribute work evenly between processors, so that no processor remains idle. Load balancing can be of significant importance to heterogeneous computing environments. As work is distributed to most appropriate processors it could happen that faster processors are congested with tasks while slower ones remain idle for longer periods. Of course, scheduling should take care of this situation, allocating processes to less powerful machines. Additionally, some techniques could be adopted that classify tasks as "easy" or "hard" so that hard tasks are given to faster processors and the easy ones - to slower processors. An example is producing many fine-grain tasks. In this way the number of tasks could determine the level of difficulty that each processor works with.

Task granularity is defined as the ratio between computation and communication [Buyya vol.1, 1999]. The granularity of tasks refers to the "independent" parts of an application that can be processed in parallel. They determine the overall execution speed of a parallel program. Fine grained tasks are small; relatively simple and most importantly they are a large number. So, they introduce large communication overhead as they are inevitably very dependant of one another and need constant exchanging of synchronization information. Also, tasks will be completed very fast resulting in constant context switches and also message passing with the scheduler of the system. Large-grained tasks, in contrast, are difficult to schedule and limit parallelism.

As discussed, parallel programming introduces a number of difficulties regarding proper distribution of work load among nodes in order to minimize overall execution time. There exist some open source cluster schedulers like MAUI.

2.2.1 MAUI Scheduler

According to its home page MAUI is a highly configurable open-source job scheduler [MAUI, 2009]. It determines where, when and how to execute jobs on a cluster or supercomputer. It is well suited for high performance computing (HPC) incorporating a large set of features. It relies on configurable scheduling policies, priorities, and limits to maximize resource use and minimize response time. MAUI controls an external resource manager such as Torque, OpenPBS, PBSPro, or Sun Grid Engine (SGE) and operates upon the information gathered by it. Users usually submit their jobs through the resource manager. The scheduler then decides how these jobs are executed and forces its decisions upon the cluster using the resource manager.

MAUI is an advanced batch scheduler that implements different mechanisms for optimal utilizations of available resources. Some of these mechanisms are advance reservations, QOS levels, backfill, and allocation management. *Advanced reservations* dedicate certain resources to specific users over a given timeframe. Reservations are managed through reservation-specific access control lists. *Quality of service* (QOS) mechanism gives special privileges to particular users that may include extended access to resources and services, special policy exemption, or job prioritization. *Backfill* is a scheduling approach that increases system utilizations by executing jobs "out-of-order" from the scheduler's priority queue. Jobs with lower priority are run together with the highest priority jobs in the queue without delaying them. Essential to this technique is estimating beforehand how long a job will run in order to determine whether it will be delayed or not. This estimate should be provided by the user. In addition to these techniques, MAUI provides an extensive administrative control allowing configurations to be enforced on scheduling, job priorities, and reservation policies.

2.3 Middleware

When a pool of interconnected computers appears as a single unified computing resource we can say that these machines have a Single System Image (SSI) [Buyya vol.1, 1999]. The SSI is achieved by a software layer that lies on top of the operating system and actively interacts with it. This software is referred to middleware. It actually resides in the middle between the operating system and the user applications. Middleware "glues" resources by message passing, moving processes across machines, monitoring and synchronizing work of nodes.

The following sections describe middleware that is used for this project. Tools and standards are divided into three categories according to level of abstraction. Low level defines cluster middleware that lies closest to the machine level. Parallel file systems describe a way of building a single file system over the disks of interconnected computers. High Level middleware is based on tools of the previous two levels. Finally Grid middleware is discussed concerning the possibility for further extension of the project and including the cluster into a large-scale network.

2.3.1 Low Level Middleware

2.3.1.1 MPI

The Message Passing Interface (MPI) defines a standard for data movement across interacting processes in a distributed system [Gropp, 1999]. MPI is not a programming language or an implemented library. It describes how basic functions for message exchange should look like. It defines the names, calling sequences, and results. Some defined functions are for point-to-point communication between processes, for collective operation execution, and for process management. MPI is typically realized as a communications interface layer that resides on the facilities of the underlying operating system. Bindings are defined for C, FORTRAN, and C++ as well as for various other languages. Programming with MPI requires explicit parallelization of code. The programmer is responsible for identifying which areas of the code can be paralyzed and then implementing a parallel algorithm using the MPI functions.

On the other hand MPI suffers some drawbacks [Dongarra, 2004]. The number of tasks working on a parallel program has to be defined in beforehand and cannot change during runtime. Another problem is the lack of interoperability between MPI implementations. One vendor's implementation of MPI cannot exchange messages with another vendor's implementation.

Additionally, in its basic form MPI does not define fault tolerance. The only specification is that if an error occurs during execution the application should be able to exit. An implementation of MPI that focuses on this problem is Fault-Tolerant MPI (FT-MPI). It offers both user and system level fault tolerance.

The second issue of the MPI standard (MPI-2) addresses some problems and introduces solutions for them [Barley, 2009]. The key areas of new functionality are Dynamic Processes, One-Sided Communications, Extended Collective Operations, External Interfaces, and Parallel I/O. Dynamic processes remove the static process model of MPI. One-sided communications provides routines for one directional communications. External interfaces define routines that allow developers to layer on top of MPI, such as for debuggers and profilers.

2.3.1.2 Open MPI

We will describe Open MPI based on the information provided on [Open MPI, 2009]. Open MPI is an open source complete implementation of the MPI 1.2 and MPI-2 standards. Its primary goal is to create a high-efficient, production-quality MPI library for high-performance computing. The project allows and encourages involvement of the HPC community with external development and feedback. Thus, it provides a better quality peer-reviewed implementation. Open MPI combines research and experience gained from previous implementations. It merges some of the well-known MPI implementations: FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI. The driving motivation behind Open MPI is to bring together good ideas and technologies from the individual projects and form one open source MPI implementation.

Open MPI is very portable. It implements a variety of communication protocols for the most popular interconnection networks used in today's parallel machines. Additionally, it supports network heterogeneity and fault tolerance. These ideas were first explored in LA-MPI and further developed in Open MPI. Other features include multi-threaded programming and thread safety. Open MPI prevents the "forking problem" common to other MPI projects.

Open MPI relies on new design architecture to implement MPI - it uses the Modular Component Architecture (MCA). MCA defines internal APIs called *frameworks* that are particular services such as process launch. Each framework contains one or more components which are specific implementations for a framework. Components can be dynamically selected at runtime. Open MPI provides point-to-point message transfer facilities via multiple MCA frameworks.

Open MPI works on three abstraction layers. The Open MPI (OMPI) layer provides standard MPI functions to the user. Below it lies the Open Run-Time Environment (ORTE) that implements a parallel run-time interface that is platform independent. Finally, on the lowest level resides Open Portable Access Layer (OPAL) which interacts with the operating system and the hardware providing an abstraction layer that hides system specific particularities.

2.3.1.3 PVM

PVM (Parallel Virtual Machine) is an integrated set of tools and libraries that form a framework for building a single parallel computer from a collection of interconnected heterogeneous computers [Geist, 1994]. The primary goal of PVM is to build a flexible, cost-efficient solution to large computational problems relying on the aggregate power of many computers. PVM's design allows it to interconnect machines of different architecture - from laptops to CRAYs. Transparently to the user it handles message passing, data conversion, and task scheduling across such a network.

Primary goal to PVM is to form a Single System Image or a "virtual machine" from a set of heterogeneous nodes connected by a network. Computers form a pool of resources and the user can specify which computers are to be used for executing a current set of tasks. PVM handles changes to the pool allowing machines to be added or deleted from the virtual machine at runtime. Computations are carried out by system processes. PVM API allows processes to be started or stopped according to different criteria, imposed also by external schedulers or resource managers. It supports also exchange of messages that check if a process is alive or inform when a process leaves the system. However, PVM allows only blocking send. Non-blocking send is available with MPI. Additionally, PVM is favored for its fault tolerance that allows users to write long running applications that resist task failure and changes in the resource pool [Dongarra, 2004].

Basically PVM systems include two parts [Geist, 1994]. First part is a daemon resides in every node that is part of the virtual machine. A daemon is a program that constantly runs in background. In the case of PVM daemons monitor system's resources and exchange messages with other daemons. Second part is a library interface routines that contains a complete set of primitives to handle cooperation between tasks. PVM relies on the notion that an application is divided into tasks. Each task is responsible for a part of the computational workload of an application. Functional and data parallelism of tasks are supported, as well as a mixture between the two.

2.3.2 Middleware for Parallel File Systems

2.3.2.1 MPI-IO

MPI I-O provides parallel I/O capabilities for the Message Passing Interface. It was developed in 1994 in the IBM's Watson Laboratory [MPI-IO, 2003]. Now it is a part of the MPI-2 standard. MPI-IO is designed to work with distributed data sets similarly to exchanging messages in native MPI. Writing files is similar to sending MPI messages and reading is similar to receiving MPI messages. All basic file manipulation actions are supported namely opening, closing, deleting, and resizing of files. MPI-IO incorporates its own file-representation strategy as a collection of *etype* units (elementary data type). Etypes control file access and positioning. Additionally, views are another concept which defines which parts of a file are visible to a task and how a task should interpret them.

MPI-IO implements different ways of accessing data which introduces flexibility to the I/O process. Data access is characterized by positioning, synchronization, and coordination. These define whether tasks should share a common file pointer in a collective manner or not, whether a process is blocked till data is written/read or not, whether access is coordinated or "free-for-all".

Data representations (data sizes, byte ordering, etc.) may vary between different platforms. A feature called File Interoperability makes MPI-IO very portable. Interoperability guarantees that data written to a file could be read afterwards obtaining the initial meaning. MPI-IO supports external and user-defined data representations. These make sure of proper handling of data coming from a machine with a different architecture or another MPI environment. MPI-IO defines internal data representations modes which ensure that file data written by one process can be read by any other process within a single MPI environment [MPI-IO, 2003].

MPI-IO has proven to be effective in a heterogeneous computing environment. Parallel programming using MPI benefits from ease of integration of MPI-IO.

2.3.2.2 PVFS

The Parallel Virtual File System (PVFS) is a parallel file system for clusters of workstations [PVFS, 2009]. The main goal of PVFS is to provide high-performance data management over a distributed environment, where concurrent access to files is common. It provides dynamic distribution of I/O workload and in that way can scale even to high-end systems. Additionally, PVFS is designed for managing large data sets - often of hundreds of terabytes. Data is dividing across the discs of the cluster nodes giving applications multiple different paths to reach a file through the network. Thus, bottlenecks are eliminated and total bandwidth is increased.

PVFS supports different access models - collective I/O, independent I/O, non-contiguous and structured access patterns [PVFS, 2009]. It supports the UNIX I/O interface and allows existing UNIX I/O programs to use PVFS files without recompiling. UNIX file tools (ls, cp, rm, etc.) operate on PVFS files and directories as well. However, relying on the native UNIX tools for parallel I/O introduces overhead as commands have to go through the operating-system's kernel. PVFS overcomes this obstacle by introducing a native API - a library which implements a subset of the UNIX operations. It directly contacts PVFS servers rather than passing through the local kernel. Another PVFS interface is ROMIO. It implements the MPI2 I/O calls in a separate library allowing MPI programmers to access PVFS files through the MPI-IO interface.

PVFS defines three different roles of cluster nodes in the system architecture: compute node, I/O node, and manager node [PVFS, 2009]. Normally, there is only one management node in a system and other nodes are dedicated to either computing or data storage. In the case of small clusters management, computation, and I/O can be carried out on the same nodes. The management node maintains all metadata of the file system, controls operations on it, and validates permissions. The metadata describes a file – for example, its name, owners, locations in the system, hierarchy in the file system, etc. When a computing node needs to access a file it contacts first the management node. Then, after all the necessary metadata is obtained, the computing node can start exchanging file data with the I/O nodes. PVFS manages file data scattering and gathering completely transparently. An application only uses the PVSF API. PVFS implements the different roles of nodes through a set of daemons. Management and I/O nodes that run the corresponding daemons exchange data with the computing nodes in a client-server mode.

PVFS is designed mainly to provide high performance parallel I/O. Taking this into consideration together with the easy installation and implementation that relies on commodity network and storage hardware, proves PVFS to be quite applicable for Beowulf-type clusters.

2.3.2.3 Hadoop

Apache Hadoop is a software platform that creates a distributed file system over the discs in a cluster [Hadoop, 2009]. It is designed to unify storage resources of big clusters built of commodity hardware. Based on Java it allows developing applications that process large data-sets. Data manipulation is completely transparent to the user and furthermore, multiple copies of data are maintained over the nodes. This provides applications with reliability and shorter paths to data, resulting in high aggregation bandwidth.

The distributed file system implemented by Hadoop (HDFS) takes care of storing file data and reallocating it in case of failure [Hadoop HDFS, 2009]. HDFS relies on a computational model, named Map/Reduce [Hadoop M/R, 2009], which has two main processing phases and manages to reduce overall amount of data without loss of meaning. Applications are required to specify input/output locations for files and also can define map and reduce functions to be applied over them. HDFS API helps do this with appropriate interfaces and abstract classes. After a *job* is configured in this way, data is divided into independent pieces which are then processed in parallel by the *map tasks*. All computations of mapping are carried out in the memory across nodes and results are then stored as files and handed in to the *reduce tasks*. Reducing collects the results; forms a single file and applies the predefined reduce function on it.

Typically computation and storage jobs are carried out on the same node which facilitates scheduling. Tasks are executed close to the data they operate on following the rule "Moving computations is cheaper than moving data" [Hadoop HDFS, 2009]. Even though, HDFS implements a client-server architecture having a single NameNode or master server that maintains meta-data of files and controls the mapping of files to physical locations in the cluster. The other nodes, named DataNodes, take care of the storage attached to them and serve read and write requests. DataNodes perform data manipulations upon instruction of the NameNode. Clients contact first the master server before they start exchanging data. HDFS maintains a file-system hierarchy and stores data in files. Internally, files are stripped into blocks which are stored on different DataNodes and replicas are also maintained.

HDFS is a Java framework that works on top of GNU/Linux operating system. It is designed to bring up fault-tolerant file system on clusters of low-cost commodity machines. Namely, Bowulf-type clusters could be a possible application.

2.3.2.4 Sector-Sphere

Sector-Sphere is an open source project of the National Center for Data Mining at the University of Illinois at Chicago [Sector-Sphere, 2009]. It is a system for distributed data storage over a single cluster or a network of geographically distributed clusters. It is designed to utilize computers built of commodity hardware. Basically the system has two main components – Sector and Sphere. They take care respectively of storage and computing services.

Sector is a distributed file system that combines resources of nodes and clusters interconnected with high-speed commodity networks [Sector-Sphere, 2009], [Yunhong, 2008]. It provides tools for data access and data manipulation. But mainly it focuses on maintaining file system semantics like file hierarchy, user access control, and common file access APIs. It is designed for read intensive tasks maintaining multiple replicas of files across the nodes in the cluster. Writing is slow because Sector does it exclusively, meaning that when a file is being written no other operations are permitted on this file. Similarly to other distributed file systems, Sector's architecture comprises of a master node that handles files metadata and coordinates the other slave nodes that store the files and process requests. There is one additional security node that manages permissions and passwords.

Sphere is built on top of Sector storage facilities, allowing it efficiently process data [Yunhong, 2008]. The system uses "stream processing paradigm" where stream refers to large static dataset. Elements of the dataset are processed independently by a processing function or a group of processing functions. This could be the Map/Reduce functions or other user-defined functions. Thus large amounts of data could be processed in parallel in a distributed environment. Sphere takes care internally of locating and moving of data, of load-balancing and fault-tolerance allowing developers that use the API to focus on implementing data-intensive parallel applications.

2.3.3 High Level Middleware

2.3.3.1 Beowulf

According to [Buyya vol.1, 1999] no concrete definition of a Beowulf cluster can be given as no two clusters of this kind share the same architecture. Nevertheless, Beowulf defines a class of distributed cluster computing that strives for achieving highest performance on lowest price. This design model results in machines less expensive than proprietary supercomputers or MPPs but of comparable performance.

The most notable characteristic of Beowulf clusters is that they rely on low-cost commodity hardware. With broad selection of models and manufacturers available for any specific component, Beowulf clusters show great flexibility. They provide the possibility to configure, optimize and restructure the system to optimally run a particular application whenever it might be advantageous. The nodes of such a cluster are dedicated, meaning that their only purpose is to work on application together. Hence, a node can only consist of the most basic components, such as one or many processors, memory and means of network connectivity. Even hard drives can be omitted. Only the head node necessitates a keyboard and a monitor. In most cases a Beowulf cluster consists of a few old desktop computers interconnected via Ethernet. The term a Pile-of-PCs is very suitable for a Beowulf cluster. The software or the middleware is what brings out the power of these computers when working together. In contrast to MPPs which use mainly proprietary software components, Beowulf uses no-cost open source software as foundation of the system. All of the nodes usually run some distribution of Linux operating system. On top of it lies middleware which brings the system together. Basically all Beowulf clusters use MPI and PVM libraries. In addition other software components may be added according to the application that the cluster is being used to run. For example schedulers like MAUI and openPBS can be used together with resource managers like Condor. The core development environment for Beowulf machines is typically a GNU compiler, of which C, C++ and FORTRAN are most commonly used.

These characteristics of Beowulf clusters make them very suitable for utilizing an environment of heterogeneous computer components for achieving high performance. Beowulf.org gives a short definition of Beowulf: “Beowulf clusters are scalable high-performance clusters based on commodity hardware. Some Linux clusters are built for reliability instead of speed. These are not Beowulf.”

2.3.3.2 OSCAR

OSCAR (Open Source Cluster Application Resources) is a software package that simplifies the process of setting up a cluster [Sloan, 2004], [OSCAR, 2009]. A collection of open source cluster software, OSCAR includes everything that one might need for a Beowulf-type, high-performance cluster. Installing OSCAR builds a completely functioning cluster out of a network of computers. Thus it is suitable for novices in the area of cluster computing, allowing them to gain experience after they build a cluster.

OSCAR is designed with the idea to bring high performance to cluster computing but in practice it can be used for any cluster application. Basically, its design suggests that the computer nodes are dedicated to the cluster. Some of them can remain in standby mode waiting to take over if a failure occurs implementing in this way asymmetric cluster architecture [Sloan, 2004]. Usual OSCAR architecture consists of one head/server node and many other client nodes. Installation is done on the cluster's head node first. Then the OSCAR installs the remaining machines, from the server using the System Installation Suite (SIS). Since the head node is used to build the client image, it is also the home for most user services, and is used to administer the cluster.

OSCAR was born with the idea of moving cluster installation towards a unified standard. That is why OSCAR is a complete system that installs “best-of-class” software in one stroke eliminating the need of downloading, installation, and configuration of individual components. Still, most of the components exist as standalone versions and undergo further development and improvement. Installation is very flexible allowing the user to exclude some packages and include others depending on the overall purpose of the cluster and thus, for example, turning a high-performance cluster into a high-availability one. The package contains MPI, OpenMPI, PVM, MPICH and LAM (Local Area Multicomputer is an MPI programming environment and development system for heterogeneous computers on a network) [OSCAR, 2009]. For scheduling OSCAR relies on the Torque Resource manager and the MAUI Scheduler. The Maui scheduler handles task scheduling using some more sophisticated algorithms. These algorithms show to be very flexible allowing also to be configured by the cluster administrator. Torque has a first-in-first-out scheduler, but by default OSCAR uses the Maui Scheduler as it is more flexible and powerful. The Cluster Command Control (C3) tools comprise a set of cluster tools that take care of, for example, global command execution, remote shutdown and restart, file retrieval and distribution, and process termination.

In addition the OSCAR package can be also installed on PlayStation3 running YellowDogLinux (YDL) 5.0. This functionality gives the possibility to include graphical processors in a cluster like the 8-core Cell Processor which PlayStation3 comes equipped with.

2.3.3.3 OpenMosix

OpenMosix (Multicomputer Operating System for unIX) is a Linux kernel extension that turns a collection of ordinary computers into a supercomputer [Sloan, 2004]. The software package facilitates setting up a high-performance cluster with putting aside worries of installation of extra libraries and doing extra configurations. Applications often need little or no change to run on such an environment. It also supports a graphical management interface – openMosixView. Additionally it integrates very well within a Beowulf environment improving the performance of an MPI or PVM [Moche, 2002]. However, the openMosix Project has officially closed as of March 1, 2008. Nevertheless, it is currently still available for use and download.

A big advantage of OpenMosix, in contrast to other cluster environments, is when running an application on such cluster it requires no recompilation or integrations of additional libraries. MPI applications greatly benefit from this. OpenMosix also supports automatic resource-use optimizations techniques that control distribution of application's processes over the cluster. It implements advanced algorithms based on market economics. Although a process starts one node, automatically it is determined whether it would be better to run it on another, less loaded node. This process can be controlled by the system administrator, too. He can affect the load at runtime by manual configuration beforehand, specifying where applications have to run and directing the load distribution to certain nodes. There are some limitations to this process, however [Buytaert, 2004]. For example, applications that rely on pthreads will not migrate, but this is considered to be a Linux problem instead of an OpenMosix limitation. Furthermore, OpenMosix features a tool for auto-discovery which makes configuration of an OpenMosix cluster very easy. The tool detects when new nodes are added or removed from the network and modifies configurations on all nodes to reflect the changes.

2.3.3.4 CAOS NSA/ Perceus

CAOS-NSA is an open-source Linux distribution that is entirely community-managed and maintained [CAOSHome, 2009], [CAOSWiki, 2009]. Initially, it was developed together with the operating system CentOS as a Community Assembled Operating System (CAOS-Linux). Both of them are descendents of Red-Hat. The latest version CAOS-NSA 1.0, however, is combining various features of GNU/Linux in order to make the distribution simple, lightweight and fast. The goal of the developers is to implement a stable core operating system that can serve to be the basis for building different kind of clusters, servers, custom appliances.

CAOS-NSA (Node, Server, Appliance) 1.0, supports all x86_64 and i386 hardware varying from desktop machines to servers and clusters. It is an "all-in-one" suite that fully integrates all specific tools needed to turn a computer into a production server or a network of computers into a high performance cluster. CAOS-NSA simplifies cluster installation by combining the operating system and the Cluster Management System (CMS) in one distribution [Layton, 2009]. CMS is a tool or set of tools that help achieving a basic single system image (SSI) from separate computing nodes. It creates an image of an operating system, transfers it to the nodes, installs it and then starts monitoring them. CMS is not cluster middleware in the sense that it does not do scheduling, mapping or solve problems in parallel [Layton, 2008]. The CAOS system manager Sidekick takes care of installing components and tools for cluster deployment. Perceus is the main component that CAOS-NSA integrates. It installs all nodes together with cluster middleware and prepares the environment for running parallel jobs. It includes OpenMPI for message passing support, Warewolf for monitoring, Slurm and Torque for scheduling [CAOSHome, 2009]. Additionally, it can install a parallel file system like PVFSv2 or Hadoop and support of fast communication links like InfiniBand, which are commonly used in modern HPC clusters [CAOSWiki, 2009].

2.3.3.5 ROCKS

ROCKS is a cluster deployment tool designed and implemented by the Rocks Cluster Group at the San Diego Supercomputer Center at the University of California [ROCKS, 2009]. It is a complete software bundle that installs everything that one might need to turn a network of computers into a production parallel environment. It can be referred to as a "cluster out of a DVD".

The installation package of ROCKS includes even the operating system – CentOS. ROCKS is tightly integrated into the operating system and not only installs it automatically but also configures it together with all necessary low-level tools in order to achieve a single-system image from the computers in a cluster.

Additional tools for parallel computations, scheduling and mapping, monitoring, virtualization, etc. are also included in the "bundle". They can be installed and configured at initial set up or later, when there is need for them. ROCKS implements a separation strategy for its components – different packages are available in the form of rolls. The rolls are defined by their purpose and can include a whole set of different tools in them. Thus, a single installation adds to the cluster a new feature instead of a new tool. An example is the HPC roll that includes OpenMPI, MPICH, MPICH2, PVM and additional benchmarks for testing their functionality. Other rolls that come with the installation of ROCKS 5.1 are Area51, Bio, Ganglia, Java, SGE and Xen. The Area 51 roll takes care of system security, the Bio roll installs bioinformatics utilities, Ganglia is a cluster monitoring tool, Java installs Sun Java SDK and JVM, SGE is the Sun Grid Engine job queuing system, and Xen installs tools for virtualization. In addition, there are also available for installation the Condor roll which adds to the system the high-throughput computing tool Condor, the pvfs2 roll, which installs the parallel virtual file system v2, and, finally, the Torque/Maui roll which includes the job queuing system Torque and the scheduler Maui (packaged by HPC Group at University of Tromso, Norway). All these tools are installed and configured automatically, so that users have a fully-functioning cluster environment at the end of the installation process. Furthermore, third-party rolls can add different functionality to the cluster like support of high-speed cluster networks like Myrinet and Infiniband, or for parallel programming on graphical processors with CUDA.

2.3.4 Grid Middleware

2.3.4.1 Condor

The following section is based on the description of Condor on its home page

Condor is the product of the Condor Research Project at the University of Wisconsin-Madison [Condor, 2009]. It is a scalable software system that creates a High-Throughput Computing (HTC) environment. It usually utilizes large collections computing resources that are of distributed ownership. In contrast to High Performance Computing (HPC), which delivers a tremendous amount of compute power over a short period of time, HTC focuses on the need of large amounts of computational power over a long period of time. Problems computed are of a much larger scale. Interest is on how many jobs can be completed over a long period of time instead of how fast an individual job can complete.

Condor is a full-featured batch system that distributes the workload of compute-intensive jobs [Condor, 2009]. It was the scheduler software used to distribute jobs for the first draft assembly of the Human Genome. Condor implements different scheduling techniques that include job queuing, different priority schemes and scheduling policies as well as mechanisms for resource monitoring and resource management. Jobs are submitted to the system, which places them in a queue and then according to certain policies decides when and where to run them. Distribution of jobs works on the basis of issuing resource requests and resource offers by the individual nodes. The ClassAd mechanism provides a flexible framework for matchmaking resource requests with resource offers. In addition, users can influence the process of mapping by describing and prioritizing jobs.

While providing functionality similar to that of a more traditional batch queuing system, Condor can be used to manage a cluster of dedicated compute nodes (such as a "Beowulf" cluster) [Condor, 2009]. It is suitable for cluster resource management as well as for efficient job distribution. The idea is to install it on every machine that is part of the cluster. A Condor cluster is referred to as pool. Jobs can be launched from any machine, which gives certain flexibility to architecture design. After a job is submitted, Condor searches for a currently idle machine with resources that match the requirements of the job. When such a machine is found, Condor transfers the job, executes it and gathers the results back on the initial machine. In addition Condor has been ported to most primary flavors of Unix as well as Windows. A single pool can contain multiple platforms which gives possibility to utilize a heterogeneous environment.

One of the features of Condor is that it does not require programs to be modified to run on the cluster [Condor, 2009]. But code can be associated with the Condor libraries gaining the ability to produce job checkpoints and perform remote system calls. A checkpoint contains the thorough information about the state of a job allowing it to be resumed on every other machine at any time. This is both a failover mechanism and a mechanism that returns the resources of a machine to its owner in the case of a non-dedicated cluster environment. For long-running computations, the ability to produce and use checkpoints can save days, or even weeks of accumulated computation time. Condor uses remote system calls to preserve the local execution environment and hide that jobs are executed on remote machines. To users a program feels like being executed on the local machine. Condor determines the remote node or set of nodes to execute the program's tasks and also takes care of logging-in and transferring the data needed for the computations.

Condor can be used to build Grid-style computing environments that cross administrative boundaries. A "flocking" technology allows multiple Condor compute installations to work together. Additionally, Condor incorporates many of the emerging Grid-based computing methodologies and protocols. For instance, Condor-G is fully interoperable with resources managed by Globus. *Condor-G* allows Condor jobs to be forwarded to foreign job schedulers. Currently, Torque/PBS and LSF are supported. Support for Sun Grid Engine is also under development.

2.3.4.2 Globus ToolKit

Globus ToolKit (GT) is an open-source software "toolkit" used to bring together computing resources, databases and other tools across geographically distributed networks [Foster, 2005]. People can share resources securely without sacrificing local autonomy. The toolkit supports resource monitoring, discovery, and management, as well as file management, all carried out upon secure channels.

Globus relies mostly on Web Services to define its interfaces and structure its components [Foster, 2005]. For example, web services use XML-based mechanisms to describe, discover or invoke network services. What is more, these document-oriented protocols are very well suited for loosely coupled computations, which are preferred in distributed systems. GT uses Web services for most of its major components. The Grid Resource Allocation and Management service (GRAM) implements interfaces for management of computational elements, Reliable File Transfer service (RFT) manages data transfers. The GridFTP provides libraries and tools for secure, reliable, high-performance data movement but it still does not implement web services.

Mechanisms for monitoring and discovery of resources are very important in a distributed environment [Foster, 2005]. Globus implements them in its MDS system. Monitoring of resources allows administrators to find and diagnose problems early. Discovery mechanisms identify resources and find services which meet desired properties. Both, collect information from multiple and perhaps distributed sources. GT implements data exchange using XML-based *resource properties* and accesses them via either pull mode (query) or push mode (subscription). These mechanisms are built into every GT service and container, and can also be incorporated easily into any user-developed service. Globus also provides three *aggregator services* that collect recent state information from registered information sources.

Security in distributed environments is another important issue considering that multiple users from different locations can access a grid network [Foster, 2005]. At the lowest level GT implements protocols that support message protection, authentication, delegation, and authorization. GT relies on X.509 public key credentials. When communication takes place entities can validate each other’s credentials, or use them to create a secure channel for message exchange. Furthermore, delegated credentials can be created, transported and used in a way that allows a remote component to act on a user’s behalf for a limited period of time.

Chapter 3
Installation Report

This chapter focuses on high level cluster middleware and its installation process. It shows the process of installing and configuring a cluster using three different tools for cluster deployment – OSCAR v6.0.2, CAOS-NSA v1.0, and ROCKS v5.1. For this, eight machines were used in total for building two separate testing clusters. All used computers have commodity hardware installed on them. What is more, all machines are completely different, meaning that they use different processors, have different physical memory capacity and storage capacity. Table 1 shows the utilized hardware configuration. Computers 1 to 4 are used for creating the first testing cluster and computers 4 to 8 were used for the other. Additionally, the testing environments use entirely commodity means of networking - Ethernet. Machines in the both clusters are connected into two separate 100Mbit switched networks using small (5 ports), 100Mbit switches Netgear.

	Comp. 1	Comp. 2	Comp. 3	Comp. 4	Comp. 5	Comp. 6	Comp. 7	Comp. 8
CPU	Intel Pentium4 @ 2,40Ghz	Intel Pentium4 @ 2,40Ghz	Intel Celeron @ 2.60Ghz	Intel Core 2 6300 @ 1.86Ghz	AMD Athlon 64 3000+ @ 2Ghz	AMD Athlon 64 3000+ @ 2Ghz	Intel Pentium3 @ 866Mhz	AMD Athlon 64 4200+ Core2 @2.2Ghz
RAM	0.99 GB	1.98 GB	1.48 GB	2 GB	1 GB	1 GB	1 GB	2 GB

Table 1 Testing hardware environment

In particular the chapter focuses on three different tools for cluster deployment – OSCAR, CAOS-NSA, and ROCKS. Section 3.1 describes OSCAR cluster suite. Section 3.2 gives an overview of CAOS NSA and Perceus together with a detailed description of installation of PVFS v2. Then, Section 3.3 describes ROCKS and installation of PVFS v2.

3.1 OSCAR 6.0.2

3.1.1 Why OSCAR?

OSCAR is a tool designed to ease cluster installation. What is more, according to its home page [OSCAR, 2009], OSCAR suite includes “everything needed to install, build, maintain, and use a Linux cluster”. Installation uses a graphical interface that guides users through a process that is usually considered to be very difficult. Building a Beowulf-type of cluster consists of a lot more work than just connecting computing nodes into a network. One should focus on details that might be starting from hardware and choosing a proper operating system that not only supports that hardware but also utilizes it in the best way. Then comes the installation of all different nodes together with the configuration of services like secure remote login (ssh), time synchronization using NTP (Network Time Protocol), network addressing and name resolution, the maintenance of a database with node information, the management of local repositories. This is just a small example of tools and services that might be necessary to bring a network of computers to work as one computer. In addition, low level middleware like a parallel file system or an MPI implementation needs to be installed and configured to make the cluster able to perform parallel computations. If all this is taken to the scale of several hundred computing nodes, one could imagine the amount of work necessary for building a cluster and the chance for error that is introduced in such process. The Beowulf type of clusters only suggests that commodity low-cost hardware is used together with open-source operating system and tools in order to reduce the price to power ratio [Buyya vol.1, 1999]. Achieving a production environment of this kind could be a demanding task even for experts in the field.

OSCAR is an effort to automate the whole building process creating an all-in-one suite that takes care of installing and configuring all necessary components [OSCAR, 2009]. The cluster installation process comes down to installing OSCAR on one computer or head-node and then all other nodes connected to it get installed automatically. In this way, one builds a fully-functioning and fully-configured cluster with one installation which is of the primary reasons to start our cluster set-up with OSCAR. The suite itself contains some of the most widely used low level middleware in the area of high performance computing. It includes OpenMPI, PVM, MPICH and LAM (Local Area Multicomputer is an MPI programming environment and development system for heterogeneous computers on a network) [OSCAR, 2009]. Compared to other high level middleware like ROCKS or CAOS NSA, OSCAR builds a complex environment with a wide range of tools and thus provides a broader opportunity for experimenting and testing. For scheduling OSCAR relies on the Torque Resource manager and the MAUI Scheduler. The Cluster Command Control (C3) tools comprise a set of cluster tools that take care of global command execution, remote shutdown and restart, file retrieval and distribution, and process termination. All these are stand-alone components and OSCAR mechanism allows using their newest releases, thus building a computing environment that is up-to-date with latest achievements in the area of HPC.

Furthermore, OSCAR is an out-of-the box cluster installation. It is a package that installs on top of an existing Linux distribution. Similar to the way Windows users are installing software, OSCAR uses an installation manager that installs it on the existing operating system. In comparison, ROCKS first formats the hard-drive, installs CentOS and then integrates itself on it. CAOS NSA is a stand-alone Linux distribution that is optimized for cluster computing. OpenMosix is a Linux patch. This independence of the underlying operating system makes OSCAR very flexible. Users have the possibility to utilize any hardware in hand as long as the operating system supports it. This gives the opportunity to take full advantage of heterogeneous environments and use them for high performance cluster computing. OSCAR was designed with the idea of being very portable. However, achieving full platform and hardware independence is still far from possible. Currently OSCAR supports different Linux distributions by developing packages especially for each of them. What is more, the installation differs between distributions for different hardware architectures like x86_64, i386 or ppc (power PC).

Among other reasons for testing OSCAR before other middleware lies the fact that it really can be installed on low-cost commodity hardware. System requirements specify a CPU no older than i586 and storage space of at least 8GB (4GB for / and 4GB for /var) for the head-node. In comparison, ROCKS requires at least 1GB of memory and at least 30 GB of storage space per node. Even though such hardware is not that hard to find today, the possibility to use older computers is essential for our project and hence this could be considered as a big advantage of OSCAR.

3.1.2 Two Versions of OSCAR

Currently OSCAR exists in two versions [OSCAR, 2009]. Developers maintain version 5.1 which was released in 2008 and is still considered to be the latest production version. At the time of writing version 6.0.2 is the latest release which dates from April 2009. The reason for the maintaining two separate versions lies in the effort of reaching the state of independence from the operating system. While version 5.1 comes as a complete software bundle with everything included in it, version 6.0.x implements a completely new installation strategy using binary packages.

To understand this better one should become familiar with the architecture of OSCAR. At the lowest level OSCAR is a set of tools aimed at deploying and configuring sets of machines. These components for basic functionality form the OSCAR *core*. The core depends on other software components. Together with 3rd party software they form OSCAR *base*. What is remarkable about this approach is the separation of elements into independent binary packages. In this case the version of OSCAR is actually the version of the core. According to the design, the core should be independent of all 3rd party software and other "external" tools that are installed through RPM/deb dependencies. That way packages can change and update without affecting the core or an already working cluster set-up. The OSCAR infrastructure can use regular RPM/deb packages to install tools for parallel computation. The services and interfaces that connect these "external" tools to the OSCAR core are called OSCAR packages (*Opkg*). This separation of roles allows changing only opkg packages to tune a system for using a new implementation of a tool. An installation approach like this one provides more flexibility and scalability. In contrast to the all-in-one version 5.1, where particular versions of tools are tightly integrated into the package, the new version 6.0.x implements dependencies of the type "greater than" and "less than". This allows changing one tool with a newer version without breaking any functionality. For example when Oscar-core is installed and it depends on system-imager 4.1.3. If for a particular Linux distribution system-imager 4.1.6 is available, then it will be installed. Current work, however, still shows OSCAR to be rather limited. Undergoing work has managed to port OSCAR core and base to the newly supported Linux distributions. Third party binaries for parallel computation are not yet fully supported.

Introducing this new approach to installation is actually a solution to supporting a broader range of Linux distributions. While an all-in-one installation package has to be ported for each platform and Linux distribution, tuning the new installation requires changing only certain packages. That way, developers of OSCAR can tune the suite faster and with less chance of error. Initially OSCAR was developed especially for Red Hat Linux. Today the suite aims to support the most popular distributions both of the Red-Hat and Debian family. *Table 2* shows the distributions that version 5.0 supports [OSCAR, 2009]. People familiar with evolution of the Linux operating system can easily see that the listed distributions are with rather limited variety and are also rather outdated. Although developers surely do their best to keep up with the changes in the operating systems, OSCAR is still far from being independent from them and thus provides rather limited choice for parallel research. Work is underway for version 5.2 beta that provides broader support to new distributions including YellowDog Linux that can be installed on Sony Play Station 3. On the contrary, the new approach to the installation of OSCAR shows that this goal can be achieved. *Table 3* shows the distributions supported already in version 6.0.2. These distributions are both relatively new and update packages could still be found for them. According to the OSCAR home page [OSCAR, 2009] the new approach introduces fast improvement as already Debian 4 Etch is supported together with partial support of the new Debian 5 Lenny and their relatives Ubuntu 8.4

and Ubuntu 8.10. These operating systems are already “today’s” technology and are broadly supported with update packages and security patches. An installation of a cluster could now be done without worries of having security vulnerabilities with the nodes connected to internet. The cluster can be secured using the built-in functionality of the operating system like better firewalls or relying on the fact that “known” security “holes” are fixed.

Distribution and Release	Architecture	Tarball Name part	/tftpboot/distro/ path
Fedora Core 4	i386	fc-4-i386	fedora-4-i386
Fedora Core 4	x86_64	fc-4-x86_64	fedora-4-x86_64
Fedora Core 5	i386	fc-5-i386	fedora-5-i386
Fedora Core 5	x86_64	fc-5-x86_64	fedora-5-x86_64
Mandriva 2006	i386	mdv-2006-i386	mandriva-20060i386
SuSE Linux 10.0 (openSUSE)	i386	suse-10.0-i386	suse-10.0-i386
Redhat Enterprise Linux 4 AS	i386	rhel-4-i386	redhat-el-as-4-i386
Redhat Enterprise Linux 4 WS	i386	rhel-4-i386	redhat-el-ws-4-i386
Redhat Enterprise Linux 4 AS	x86_64	rhel-4-x86_64	redhat-el-as-4-x86_64
Redhat Enterprise Linux 4 WS	x86_64	rhel-4-x86_64	redhat-el-ws-4-x86_64
Scientific Linux 4	i386	rhel-4-i386	scientificlinux-4-i386
Scientific Linux 4	x86_64	rhel-4-x86_64	scientificlinux-4-x86_64
CentOS 4	i386	rhel-4-i386	centos-4-i386
CentOS 4	x86_64	rhel-4-x86_64	centos-4-x86_64

Table 2 – Supported distributions by OSCAR 5.0.
Source: OSCAR home page [OSCAR, 2009]

Distribution and Release	Architecture	Status	Known Issues
Red Hat Enterprise Linux 5 / CentOS 5	x86	Fully supported	None
Red Hat Enterprise Linux 5 / CentOS 5	x86_64	Fully supported	None
Debian 4	x86	Fully supported	Not all OSCAR packages are supported
Debian 4	x86_64	Fully supported	Not all OSCAR packages are supported
Ubuntu 8.04	x86	Fully supported	Not all OSCAR packages are supported
Ubuntu 8.04	x86_64	Fully supported	Not all OSCAR packages are supported
Debian 5	x86_64	Experimental	Testing still needed
Fedora Core 9	x86	Experimental	Testing still needed
Open Suse 10	x86	Experimental	Testing still needed

Table 3 – Supported distributions by OSCAR 6.0.2.
Source: OSCAR home page [OSCAR, 2009]

Another goal of the new installation approach is to provide support for the Debian family of Linux distributions. Currently, the Linux world is divided between the Red-Hat’s and Debian’s installation packages and systems that manage them. Binary packages of the two systems are incompatible with each other and thus a software installation must be ported separately for both of them. Red-Hat defines its software package format as RPM (Red-Hat Package Manager) while Debian packages just use the file format “.deb”. The RPM family includes Fedora Core, CentOS, Scientific Linux, Mandriva, Suse, etc. While OSCAR was initially developed to support only RPM based systems, the latest version focuses on supporting Debian too. Currently only OSCAR core is ported for Debian and none of the tools for parallel computation could be installed yet.

OSCAR stores all packages in online repositories. The differences between the two Linux families suggest different organization of the repositories for them [OSCAR, 2009]. The Debian world assumes that the repository is driven by the version of the distribution and thus OSCAR version cannot be included in the name of the repository. What is more, packages for different architecture and distribution can be stored together (e.g. Debian 4 and Ubuntu 8.04). In this case, an installation tool like *apt-get* relies on proper metadata description to determine which the needed packages to install are. In contrast, in the RPM world the repository name specifies distribution, version and architecture. A common solution that OSCAR is using is a name including only distribution and architecture. For example the online repository for OSCAR 6.0.2 is *http://bear.csm.ornl.gov/repos/debian-4-x86_64/ etch /.*

The installation of the suite itself is straightforward. The user specifies a repository address like the one already mentioned. Depending on which family the hosting operating system belongs to, the address is specified either in the */etc/apt/sources.list* for Debian systems, or by placing a file with the address in it in */etc/yum.repos.d* for RPM-based ones (See Table 4). Then the OSCAR packages are automatically downloaded and installed using a package management tool like *yum* for RPM systems or *apt-get* for Debian. To complete the installation one should follow the described steps in *Table 3* logged in as *root* user.

		Debian Based Systems	**RPM Based Systems**
		Installation on the head node	
1	Define online repository (*e.g. for a x86_64 architecture*)	Copy ***http://bear.csm.ornl.gov/repos/debian-4-x86_64/ etch /*** *in* ***/etc/apt/sources.list***	Add the following file: ***CentOS-x86_64-OSCAR.repo*** to ***/etc/yum.repos.d***
2	Update the system (**Do not upgrade**)	*apt-get update*	*yum update*
3	Installation of OSCAR	*apt-get install oscar*	*yum install oscar yume packman orm perl-AppConfig*
4	Specifying the current distribution	*oscar-config –setup-distro <distro>-<version>-<arch>*	*oscar-config –setup-distro <distro>-<version>-<arch>*
5	Install prerequisite packages and OSCAR server packages. Start services	*oscar-config --bootstrap*	*oscar-config --bootstrap*
6	Check if system is properly configured	system-sanity	system-sanity
		Cluster Installation	
7	Start OSCAR cluster installation GUI	oscar_wizard install	oscar_wizard install

Table 4 – Installation process of OSCAR

3.1.3 Installation

The whole installation process is well described in the documentation of OSCAR provided online [OSCAR, 2009]. That is why this section focuses on some points that determine the installation process as well as it describes experience gained from installing OSCAR 6.0.2 on different Linux distributions.

3.1.3.1 Environment Considerations

OSCAR is designed to implement a typical Beowulf cluster architecture. The installation process first installs OSCAR on the head-node (server, frontend, master node) and then it uses this installation to distribute itself and install on the client nodes (computing nodes). The head-node is usually a computer more powerful than the rest of the nodes in terms of memory capacity and processing power. It serves the requests of the clients, maintains a database with client information, distributes jobs among the clients, etc. In the case of OSCAR the only actual requirement for the head-node is to have two network interfaces. One is required to connect the head-node to the Internet and the other to connect it to the cluster network. OSCAR builds a cluster having only the head node connected to the "outside" world. The rest of the computing nodes utilize a local switched network. The head-node is the only one accessible on the internet and thus serves as a firewall for the rest of the cluster. That way installation does not have to take care of configuring security on the computing nodes. The head-node can be configured to use a firewall or port forwarding. It even can implement routing policies to provide access to the internet for the computing nodes. This, however, has to be configured manually as it is not part of the OSCAR installation. OSCAR only gives addresses to the nodes from a specified range and takes care of name resolution by modifying the */etc/hosts* file.

According to the OSCAR installation guide provided on its home page [OSCAR, 2009], OSCAR can be installed on a pre-existing server nodes but it is highly recommendable to use a fresh installation for building a new cluster. What is more, one should be very careful when choosing a distribution and should first consult with the documentation whether a distribution is supported and to what extent (see to Table 2 and Table 3). Although work is underway the current version is reported to work thoroughly only on Red Hat Linux. However, obtaining update packages for this distribution requires a license and it is not free. Thus, free distributions like Debian are preferred. However, at the time of testing installation supports only OSCAR core and base packages without support of any additional tools for parallel computing. They have to be installed separately after building the cluster.

Having a fresh installation one should not hurry and upgrade it because OSCAR still might depend on older versions of some tools. One should consider upgrading only security packages and installing security patches. Upgrades can install new versions of tools like the scripting language python, visualization libraries that the OSCAR GUI depends on, and many others. OSCAR makes heavy use of Perl and Python script for its installation and configuration and changing their versions might result in loss of functionality. Furthermore, often, even the kernel itself gets renewed and this means that the current version of OSCAR will no longer work because it is not supported. In this case the node has to be reinstalled and configured again.

What should be initially considered when configuring the head node is networking. The head-node needs to have Internet connectivity especially if a version 6.0.x is being installed as it uses online repositories to download the packages from. The configuration can be done using either the Command line interface (CLI) or a graphical network management tool. One might stumble upon problems of any sort. For instance, Ubuntu 8.10 has a bug with the Gnome Network Manager included with the release. It resets any static IP address settings when system is rebooted and sets the system to use dynamically obtained addressing via DHCP. This could cause troubles with an OSCAR cluster because it usually uses static addressing for the local network interconnecting the nodes and the head-node. The solution is to remove the Gnome Network Manager and manually configuring the interfaces by modifying the */etc/network/interfaces* file. Tutorials for configuring it can be easily found on the Web. Another thing that could cause trouble is the naming which the operating system uses for the network interfaces. By default OSCAR is configured to use *eth1* as the interface connected to the public network and *eth0* as the one that connects the frontend to the cluster network. In a case when the head-node operates in a corporate network it might get its addressing configuration via DHCP. Then, in order to have internet connectivity an interface's hardware address may be registered in the network. If the case is such that *eth0* is registered and *eth1* is not, one might want to use them the other way around. OSCAR defines which interface to use for the internal network in its configuration file. It can be changed in the first line of */etc/oscar.conf.* Furthermore, configuration of a firewall has to be carefully done for the local network in order not to restrict access to and from the computing nodes.

3.1.3.2 Installing OSCAR on the Head-node

Installing an OSCAR cluster starts with the installation of an appropriate operating system in accordance with the list of supported Linux distributions (see Table 2 and Table 3). Then one should think about configuration especially of networking. Only then, after the head node is prepared OSCAR can be installed on it (see Table 4). If the operating system is supported installation goes straightforward without any errors. Our experience showed that this is rarely the case with OSCAR 6.0.2. Bootstraping the system on step 5 (see Table 4) prepares the head-node for being a server for OSCAR services. At this stage it downloads and installs prerequisite 3rd-party packages needed for OSCAR installation and operation. It also starts and restarts services. Often, during this step errors are generated about missing packages and dependencies. OSCAR was implemented to try finding the missing packages and installing them on its own. However, one might need to install some dependencies manually. Some distributions, like Ubuntu 8.x, are rather light in terms of included tools and packages. Installation on Ubuntu 8.10 required manual installation of 16 packages including openssh, apache2, mysql-server, nfs-kernel-server. At the time of testing only Ubuntu 8.04 was reported to be supported. It managed to find these packages on its own and install them. In this case one could observe how important it is to install a supported distribution. However, OSCAR did not install on Ubuntu 8.04, too. The bootstrapping stage did not find a package *system-imager-initrd-template-<arch>*, where <arch> specifies some different architectures like ppc, ppc64, ppc64-ps3, etc. A search showed that such package exists only for i386 and x86_64 architectures, and not for the ones reported missing. These two packages could not be installed as they depend on opkg-sis-server which turned out not to support them.

When the bootstrapping stage succeeds, the software environment on the head-node is prepared for OSCAR installation and deployment of the cluster. One should check this by executing *system-sanity* check. It runs a series of scripts that check configuration. Usually it returns a warning or error message about network configuration. This can be fixed by configuring the */etc/hosts* file because the scripts check configuration in this file.

At this point the installation can continue by using the graphical user interface of OSCAR (see Table 4, step 7). Bringing up this interface of course requires the Linux distribution to support either of the two Linux graphical environments – KDE and GNOME. The *oscar-wizard install* command launches the interface. Oscar documentation gives a detailed description of the interface and how one can use it [OSCAR, 2009]. Here it is worth mentioning that the interface uses perl-Qt libraries for drawing on the screen and thus might not run if it does not find some components. At the time of testing OSCAR 6.0.2 on Fedora Core 9 had this problem. Although, according to OSCAR documentation this problem is only with OpenSUSE, it turned out that FC 9 also needs to manually install the *Qt* packages (qt, qt3-devel, libsmokeqt, libqt3). Proceeding with the installation, one should be aware that step 3 of the graphical interface installs OSCAR core, servers and components onto the head-node. The installation can also fail on this step. Fedora Core 8 is missing a package *python-elementtree*. This package is reported to be obsolete with Fedora Core and has been dropped out as being too old. OSCAR 6.0.3 is reported to have fixed problems like this and that now it supports FC 8 and 9.

The installation of OSCAR 6.0.2 was fully successful only on Debian 4 Etch. Even though the head-node in hand was of 64-bit CPU architecture a decision was made to install Debian 4 for 32-bit architecture. One should pay attention to what hardware is available for the cluster. The operating system, installed on the head-node, gets included in the client image and thus is distributed among the computing nodes during cluster set-up. If there are nodes of 32-bit architecture in the heterogeneous environment a possibility arises of having serious difficulties when trying to install the OSCAR image on those nodes. What is more, most 64bit commodity computers can operate simulating 32bit mode exactly because of compatibility issues with the operating systems. On the other hand, OSCAR provides a possibility to tune the image and build different images for different sets of nodes according to their architecture. This, however, increases the chance of having a serious problem with cluster deployment and was not a preferred option at the time of testing.

The installation process is without any errors only when relying on some tricks that take advantage of the development work on OSCAR 6.0.3. Starting with a "clean" installation one should keep in mind to remove the word *testing* from all mirrors listed in */etc/apt/sources.list.* Because this file defines the http and ftp addresses the system uses for downloading updates, it is crucial not to allow downloading of new testing versions of tools. Then the system should be updated but not upgraded. This means that only lists of available updates are downloaded without the files themselves. The difference is more obvious when considering the two separate commands *apt-get update* and *apt-get upgrade.* One should follow the steps of the installation process described on OSCAR's home page (see Table 4). According to the architecture of the Linux distribution a choice has to be made whether to use the online repository for x86_64 and i386 architecture. On step 7, when trying to run the installation GUI, OSCAR issues an error message for not finding a *Selector.pl* file. For fixing this, one should configure OSCAR to use the *unstable* online repository by changing the */etc/apt/sources.list* file and/or */tftpboot/ distros/debian-4-i386.url* file. Initially the /tftpboot/distros folder and the file do not exist. They are created only when distribution is defined on step 4 of the installation process (see Table 4). After this adjustment OSCAR needs to be bootstrapped once again. When dependencies are resolved one should change back to the original repository. It is important to change back the file before installing the server packages at step 3 of the GUI interface. It is not recommended to specify the unstable repository at initial set-up because that will lead to downloading and installing the next testing version of OSCAR and related packages. Specifying the stable repository downloads and installs packages from the current release. In this matter one should be very careful when changing the */etc/apt/sources.list* file and then issuing an update (or upgrade) of the system with *apt-get update (upgrade).* Still, fixing some problems regarding dependencies and missing packages might require switching to the address of the unstable repository. Currently there are two online repositories that can be used. One starts with: "*http://bear.csm.ornl.gov/ repos/*" and the other starts with "*http://bison.csm.ornl.gov /repos/*". Both can be configures in */tftpboot/distros/debian-4-i386.url* file. Changing between the unstable repositories on these two servers showed to be helpful and eliminated some errors at step 3 of the GUI installation. When the dependencies are resolved one should remember to change back to the original repository address.

3.1.3.3 Installing the Cluster

OSCAR gets installed and configured by following the process described in Table 4 and the steps of the GUI installation manager. The installation process guides users through deploying the cluster as well. The cluster deployment starts at step 4 ("Build OSCAR Client Image…") of the GUI installer. At this step OSCAR builds an image of the existing operating system and its configuration. The image is approximately 2GB of size and is stored in */var/lib/systemimager/images* directory. Interface prompts users to specify CPU architecture and Linux distribution for the image as well as hard drive type. These options provide opportunity for creating several different images in case of utilizing heterogeneous hardware. Even though, the purpose of this thesis is to study the behavior of heterogeneous environments, a decision was made to use a unified image for all computing nodes. The image is of Debian 4 Etch, i386, which is actually an image of the head-node.

OSCAR installs the nodes of the cluster using the System Installation Suite (SIS) [OSCAR, 2009]. SIS is a set of tools for automated massive Linux installations for both Red-Hat and Debian systems [Dague, 2002]. It has three major components: SystemImager, SystemInstaller, and SystremConfigurator. All of them are stand-alone components but are designed to integrate well with each other. SystemInstaller is the tool used to build a Linux image and place it on a server. SystemImager actually propagates, installs and manages the images on the clients. Finally the SystemConfigurator provides a single API for Linux configuration like tuning the network setup, boot loader setup, and ramdisk creation. It takes care of adjusting the images after they were installed by the SystemImager in order to tune them according to the underlying hardware.

Central to the whole process is the notion of an image. An image is a "clone", a replica of the operating system and its configuration on a certain node. This node is referred to as "golden client". Images are captured from running machines and are full live file systems. Then they are stored on an image server where the other nodes (clients) can download them from. In the case of OSCAR the golden client and the image server are one and the same – the head-node. The clients are the computing nodes to be installed.

According to the guideline of the OSCAR GUI installer, the installation process continues with step 5 ("Define OSCAR Clients…") which allows users to define a number of hosts that will be installed, name pattern for them (for example: oscarnode-0-x) and most importantly a network address range. The address range determines the IP addresses the nodes will use for the local cluster network. At the next step ("Setup Networking…") client installation is initiated. The process starts with registering clients MAC addresses and mapping them to IP addresses in the order they get discovered. During this stage OSCAR is already running an image server and listens for client requests on the internal network. Client nodes have to be configured to use network boot from their BIOS (Basic Input/ Output System) configuration. OSCAR counts that the client nodes use PXE (Preboot eXecution Environment) to establish initial connection to the server. PXE is a popular method for booting a system using the network. Clients obtain a temporary network addressing and search for Preboot server to download a boot kernel via TFTP (Trivial File Transfer Protocol). In the case of OSCAR, and SystemImager in particular, after booting, the clients download to their RAM memory a small kernel of Embedded Linux called BOEL (Brian's Own Embedded Linux). It helps the systems to initiate a proper BOOTP/DHCP request for obtaining an IP addressing configuration. When BOEL has brought the client machine to the network, it starts looking for an auto-install shell script. The script is named <nodename>.sh and is usually located in the */var/lib/systemimager/scripts/* directory on the server. The auto-install script determines the rest of the installation process. It defines how the hard-drive will be partitioned, mounts the newly created partitions on /a, invokes the System Configurator to tune the image to the particularities of the client's hardware [OSCAR, 2009]. Following this configuration the actual image starts downloading and installing. The file transfer uses rsync, which is a mechanism for remote file synchronization that provides a possibility to use a secured ssh connection. OSCAR can also use bittorrent or multicast (flamethrower) mechanisms for file transfer. Step 6 ("Stup Networking…") also gives users the possibility to burn a SystemImager boot CD that can be used in the case nodes do not support PXE-boot mode. The installation process remains the same with the only difference that the nodes need to be configured to use the CDROM for initial boot device instead of the network controller.

The installation of the cluster should be as easy as booting the computing nodes and wait for them to install. Our experience showed that problems may occur here, as well. When images were being installed an error occurred regarding the component *grub-install*. GRUB (Grand Unified Bootloader) is a small piece of software responsible for loading and initial loading configuration of the operating system. GRUB is a boot loader similar LILO (Linux Loader) but only more flexible and tunable. As it turned out, the image of the system did not include GRUB's home directory to the directory tree. Thus nodes were missing this directory and that is what caused error messages to be generated during the installation process. Adding a directory ~/boot/grub to the image hierarchy tree on the head-node fixed the problem.

3.1.3.4 Local Repository Setup

At this stage OSCAR 6.0.2 has built a working cluster environment on top of Debian Etch Linux. OSCAR core and base packages together with OSCAR database are configured and running in a way that makes the testing installation on four old desktop computers work as a cluster. However, at this stage the OSCAR cluster is far from being a working Beowulf-type parallel environment. Manual installation of numerous tools is the only way to get this set-up to the level of a parallel environment. OSCAR 6.0.2 does not install any of the stand-alone tools for parallel computation like OpenMPI, LAM, MPICH, Torque/Maui, Ganglia. What is more, it does not even install a time synchronization mechanism between the compute nodes and the head-node. This is something that tools for parallel computation may rely on. Also, most tools need to be able to access the computing nodes without using a password. Remote login is supported to the extent that openssh is installed but configuration is still required to remove the need for password on remote command execution. This situation demands for a solution to how to provide installation packages to the nodes of the cluster. One has to consider the fact that the nodes usually have no internet connectivity and are "hidden" behind the head-node from the rest of the world. Several approaches were tried in this situation. One is to define a proxy server on the head node and thus provide each node with internet connectivity. Another is to define a local repository on the head-node, fill it with all necessary packages and provide access to them either via HTTP or FTP. That way initial cluster architecture is preserved and again, all nodes can have unified images of the software installed on them.

A single package installation is all needed to install a proxy server on the head-node. Then it has to be properly configured to provide access only to certain sites. In the configuration file (e.g. */etc/apt-proxy/apt-proxy-v2.conf)* on the head-node one can specify the addresses of the allowed sites. Under the tag [debian] one can add

```
backends= http://ftp.us.debian.org/debian
          http://ftp.de.debian.org/debian
          http://ftp2.de.debian.org/debian
          ftp://ftp.uk.debian.org/debian
          http://bear.csm.ornl.gov/repos/debian-4-i386
```

Then, on each node the file */etc/apt/sources.list* has to be modified with the address and the port of the proxy server. An example is

```
    deb http://oscar_server:9999/ftp.de.debian.org/debian etch main
    deb  http://oscar_server:9999/bear.csm.ornl.gov/repos/debian-4-i386
etch /
```

where oscar_server defines the address of the head-node in the local cluster network and 9999 is the port defined by the proxy server. With this configuration the compute nodes should have access to the sites, where they can download update packages and installation packages using the native installation tool of the operating system. In the case of Debian this is apt-get. Furthermore, OSCAR helps in this case as it installs a tool for parallel command execution, file retrieval and distribution. The Cluster Command Control (C3) tool provides a way of issuing commands on all nodes at the same time. Using *cexec* (cluster execute) command, one can easily issue an update of the nodes from the head-node and even install packages. For example NTP packages can be installed by issuing

```
# cexec apt-get update
# cexec apt-get install --force-yes ntp -allow-unauthenticated
```

cexec does not support interactive mode, meaning that if a command requires a user response at some point, *cexec* will crash. It cannot return the question to the head-node and distribute the answer again. One has to make sure that any possible questions that might appear during the command execution are answered in advance (for instance, specifying *–force-yes* in apt-get options).

On the other hand, this approach to providing installation packages for the clients may introduce a number of difficulties and was not used on the testing environment. To start with, it introduces a conflict with the initial architecture design of the cluster. The head-node provides a single point of access to the outside world and thus it protects the cluster. Giving internet access to all nodes in the cluster introduces numerous security breaches and exposes the cluster to danger of external attacks. Nodes need to be configured additionally in order to improve security. Security patches have to be installed and even network configuration has to be changed. Even more, giving the nodes possibility to download their own packages makes administration of the whole cluster more difficult. There is no longer a single point of administration as, most certainly, nodes have to be configured separately. In general, all nodes should have the same software with the same versions installed on all of them in order to simplify control over the cluster. Any differences may result in having services that are not working. A slight overlook in installation of new packages could lead in having different versions of a tool running on the cluster. For example, if two versions of OpenMPI are running on the cluster, a parallel program might not utilize the nodes running the newer version. The result is loss of performance.

There is a second solution to providing installation packages to the computing nodes – a local repository on the head-node. It aims at eliminating both the problem of administration and the problem of security. This follows the initial design of the cluster, where the head-node is a central point of administration and access to the cluster. It can host necessary installation and update packages and provide access to them ether via HTTP or FTP. The computing nodes will need minimum configuration after they are installed. Additional packages can be installed on all of them using *cexec* and Debian's native package management tool. The advantage in this case is that all nodes will surely install the same packages. Monitoring and control of the updates and upgrades in this case become easier as cluster administrators only need to make sure that the local repository is properly managed. What is more, security is no longer an issue because the internal cluster network does not need to implement security policies except secure remote login (ssh). Security of the cluster comes down to defining security on the head-node and its connection to the outside world.

A local package repository for the cluster is not more than a file server running on the head-node. Probably, the only thing that distinguishes this file server from others is that clients will only download from it but will not store files on it. This can be achieved in two ways according to the way files are transmitted. One solution is to rely on the File Transmit Protocol (FTP). This means launching an FTP server on the head-node. With operating system like Debian, an ftp server is installed by installing a single package which in our case is *vsftpd.* Configuration (if needed) is also straightforward. The installation that we tested required placing all files in a single directory (e.g. /home/ftp/). Then the server uses this directory to share the files in it. Similar is the approach using an HTTP server for sharing the files over the cluster network. An advantage to using an ftp server is that an instance of an HTTP server is already installed and configured with OSCAR. OSCAR needs such server running because it maintains an oscar-database with node information like MAC addresses and names. Version 6.0.2 requires installing *Apache2* server on the head-node. At this point administrators of the cluster can take advantage of the already running apache server and create the repository by placing binary packages in the right directory. By default the directory apache uses is */var/www/,* meaning that when a client connects to the server requesting a WEB page or a file the server will start looking for them in this directory.

Let us assume that the directory of the local repository is */home/cluster/distro/debian-4-i386/*. Then the address that the nodes have to use to connect to the server will look like *http://gateway/repo/home/cluster/distro/debian-4-i386 etch /*. This is rather long address and that is why one can use a simple Linux trick to make it shorter and more intuitive. What we did is adding a symbolic link file to the default directory of the HTTP server (*/var/www/*). The file */var/www/repo* points to the actual directory of the repository */home/cluster/distro*. This way nodes can use only an address like *http://gateway/repo/debian-4-i386 etch /* to connect to the server.

In order to download installation packages, the compute nodes can rely only on Debian's native package management tool *apt-get.* When an update is issued it tries to connect to FTP and/or HTTP servers defined in the file */etc/atp/sources.list*. Hence, each node in the cluster has to have this file modified with an address of the kind *deb http://gateway/repo/debian-4-i386 etch /*. However, an update downloads only a list of all available binary packages in the repository (e.g. Packages.gz). After that, when installation of a package is requested, the manager checks the list whether there is an appropriate package of this kind on the server. This specific functionality of the manager requires the repository to have this "description" file. What is more, it has to be a compressed file (with gzip or bzip2) and also has to be renewed each time a package is changed or removed. At this point, it is worth mentioning that creation of a repository is often related to building and maintaining a special file hierarchy. Meaning, that in general binary packages and their description file can share the same directory but this is often not the case. The package management tool apt requires the description file to be separated from the rest of the package files. In our test environment three different tools were tried with different success for creating this file and the file hierarchy around it.

The directory of the repository can contain a controlled set of packages or just all packages that the head-node has already downloaded. Debian stores all ".deb" packages in */var/cache/apt/archives/*. One can place files in the repository just by copying them from this directory. After all needed packages are present a description file has to be created. *Dpkg* (Debian package) is a low level tool for managing .deb packages. It can install, remove and manage packages just like *apt* and *aptitude* but in a more basic manner. The tool dpkg-scanpackages can be used to create a compressed list of all packages in the repository. According to its man page "dpkg-scanpackages sorts through a tree of Debian binary packages and creates a Packages file". In the command line one can specify

```
# dpkg-scanpackages debian-4-i386 . /dev/null | gzip -9c debian-4-
i386/Packages.gz
```

where *debian-4-i386* is the directory that contains all binary packages. This command has to be issued from the directory "above" it. Then, a description file (Packages.gz) is created in the same directory.

To manage repository hierarchy one should rely on more complicated tools that actually make use of dpkg-scanpackages in a controlled way. One such tool is *debarchiver* [Liedert, 2005]. It is a tool to sort files into the file structure used by the Debian package management tools like apt-get, dselect, etc. All the user has to do is place files in a predefined input directory and debarchiver produces a sorted hierarchy into a predefined destination directory. Input and output directories can be changed in the configuration file */etc/debarchiver.conf*. Debarchiver is implemented to be called repeatedly over a certain time period. By default this period is 5 minutes but this can be configured by tuning the file */etc/cron.d/debarchiver*.

The last tool tried on the OSCAR head-node in order to make a repository out of a set of files, is *rapt*. Rapt is a tool that comes with OSCAR installation. It is a wrapper for apt-get that aims at managing cluster repositories. More specifically, according to its man page “Rapt is a tool for setting up, exporting apt repositories and executing apt-get commands for only those repositories”. [OSCAR, 2009] explains that the tool is designed to support repositories for different distributions and architectures. That is why it creates the file hierarchy specific for OSCAR repositories. As it is described in the section “Two versions of OSCAR” the repository hierarchy of OSCAR includes a distribution name and architecture (e.g. debian-4-i386). Then, according to this hierarchy the Packages.gz file is placed in the directory ~ */debian-4-i386/dists/etch/binary-i386/.* The following command builds the metadata cache (Packages file) for all binary packages issuing a call to dpkg-scanpackages. It takes care of creating the file hierarchy on its own

```
# rapt --repo /home/cluster/distro/debian-4-i386/ --prepare
```

Then a similar command is used to export the local repository via http utilizing the installed Apache server for this.

```
# rapt --repo /home/cluster/distro/debian-4-i386/ --export
```

At this point all nodes should be able to download the packages list file and install files from the local repository on the head-node. However, building this local repository is only an effort to fix the functionality of OSCAR 6.0.2. This production version is still far from being complete and introduces more difficulties in the process of building a Beowulf-type cluster than it eases it. OSCAR implements a strong feature set that will surely prove to be productive only when problems with support of the underlying distributions are resolved. Our testing environment showed that a working cluster can be achieved with Debian Etch as a foundation. However, it also showed that the cluster is not suitable for production. Even with a local repository running, installation and integration of tools for parallel computation turned out to require a lot more effort and time. An attempt to install OpenMPI returned many dependency errors and finally led to irresolvable situation as core libraries for C/C++ required upgrading of the core. And changing the core of the operating system to a newer one results in OSCAR not functioning anymore because compatibility issues.

3.2 CAOS-NSA

CAOS-NSA (Node, Sever, Appliance) is open-source RPM-based Linux distribution. Originally it was developed as a freely distributed descendant of Red-Hat Linux known as only CAOS (Community Assembled Operating System). However, the lifecycle of CAOS ends with version 2 giving birth to the project CAOS-NSA. It aims at providing production and scientific computing environments with an operating system that is both stable and lightweight. In contrast to other Linux distributions installation is rather small and consists of a single CD image of 608MB. After the system is installed it automatically updates itself with the latest packages. Only after that, the clean and simple operating system can be tuned to carry out a specific task. Additional tools and packages are downloaded, installed and integrated into it in order to make the operating system most suitable for performing this particular task. CAOS-NSA can be tuned to optimize system resources for creating a dedicated server environment, a cluster, or a development environment. For this CAOS-NSA uses Sidekick. Sidekick is a text-based tool for post-installation administration and configuration [CAOSWiki, 2009]. It is automatically launched during initial system installation process and guides users through configuration starting from language and keyboard-layout selection. Sidekick makes the installation a controlled process by letting the operating system automatically take care of configuration. It is also used to pre-configure the system for carrying out a particular role such as Graphical environment, Email server, Web server (LAMP), File-Server, Database administration, Support of virtual machines, Clustering. Once a profile is selected, the system starts downloading prerequisite packages and configuring itself. After all packages are installed some additional configuring may be necessary in order to tune the tools to the particular application in hand. Nevertheless, CAOS-NSA takes care on its own to integrate the new tools into the system and configure them. This way it minimizes the chance of error by minimizing user interaction. If another system profile is needed at some later point, it can be added to the running system again by using Sidekick. It installs all new tools without uninstalling the old ones. However, it deletes all meta-packages for the old profile preventing it from being able to update furthermore. This way the system focuses only on one particular role and dedicates to it.

The installation process of CAOS-NSA is described in detail on the wiki page [CAOSwiki, 2009]. It is straightforward, simple and fast (takes around 10 minutes). The clustering profile of CAOS-NSA prepares the initial system to be a master node for a HPC (High Performance Computing) cluster. According to the CAOS home page [CAOSHome, 2009] the operating system is designed and implemented to eliminate performance regressions at the lowest level of the single system so that when this system is brought to a large scale it does not create an aggregating performance drop. Installation of the clustering profile starts with installing the Perceus management tool, which is actually responsible for deploying the cluster. There is detailed information about installation and configuration techniques in the user guide of Perceus [Perceus, 2009]. CAOS-NSA, however, imports it and takes care of its complete configuration. Our experience shows that installation is simple and straightforward. Perceus then installs a number of other tools. In contrast to other high level middleware like ROCKS and OSCAR, it installs only OpenMPI as a tool for parallel computations. This provides rather limited range of MPI implementations to choose from considering the fact that both OSCAR and ROCKS include also MPICH, MPICH2 and PVM. On the other hand, Perceus adds the mechanism of environment modules that allows users to add different libraries and compilers for parallel computing. Environment modules allow users to change the execution environment by maintaining a set of module files, which hold the necessary information to configure the shell for an application. Furthermore, CAOS-NSA also downloads the Slurm job scheduler and the parallel file system Gluster2 and makes them available to be instantly included in the cluster configuration.

After Perceus is installed on CAOS-NSA it has to be configured with number of nodes to be added to the cluster and IP address range of the internal network. Like OSCAR and ROCKS, Perceus assumes that the master node has at least two network interfaces - one to be used for the internal cluster network and the other to be used for connecting the master node to the Internet. In contrast to ROCKS, Perceus can use dynamic address configuration for the interface connected to the Internet. Like in the case of OSCAR and ROCKS, Perceus also implements a cluster architecture where there is only one master node and all other nodes are connected to it via switched network. This way the master node acts as a central point for service provision to the rest of the nodes. According to the Perceus documentation [Perceus, 2009], such implementation is only applicable to small to medium sizes clusters. Clusters with more than 500 nodes may need another server in the network. Our test environment includes only four nodes, which requires installation of a single master node. All four nodes are desktop computers and have commodity hardware installed on them. It is crucial to note that Perceus runs on systems with CPUs of x86 architecture (e.g. ia32, x86_64). Hardware, like ia64 and PPC64 are still not supported. Although the supported architectures are rather popular and widely used, Perceus puts certain limits on the hardware that can be used for building a production cluster.

Nevertheless, CAOS-NSA together with Perceus manage to implement a fast and robust way to cluster deployment. According to [Layton, 2009] installation of the master node takes not more than 10 minutes and deploying a cluster with two nodes (master and slave) takes around 23 minutes. What is more, the documentation of Perceus [Perceus, 2009] states that a single master node can install simultaneously 32 computing nodes. So, one can have a cluster of 32 nodes running in 25 minutes, which according to [Layton, 2009] is faster than any other tool for cluster deployment.

To understand how this is possible one needs to become familiar with the installation technique Perceus employs. Similarly to OSCAR, nodes are installed with an image of the operating system provided by an image server that resides on the master node. A major difference is that Perceus installs all nodes in a cluster with a stateless operating system - nodes do not install the image on their local media but they load it into their random access memory (RAM). This is a fast and efficient way of installing as all operations takes place in the RAM and they do not involve any read/write operations of the hard drive. What is more, nodes can have no hard drives at all. Often in a high performance environment where nodes are dedicated only to performing fast calculations there is no need for have hard drives. In this way one can build an HPC environment that minimizes costs. Furthermore, a stateless operating system is easy to upgrade and modify. All that is necessary is to change the image on the master node and reboot all other nodes. Then all of them will boot with the new image. This solution provides great flexibility for the cluster and saves the time of administrators as nodes' hard drives need not be reinstalled. In comparison, ROCKS requires all nodes to be reinstalled after a new roll is installed and the golden image changes. On the other hand this approach of installing the operating system in the RAM has a major drawback – every reboot of a node requires an image to be provisioned. An image server has to be running in the cluster all the time. Usually this is the master node. Considering the fact that it also takes care of several other services like job distribution and scheduling and remote access to the cluster, it can happen that the master node crashes. Having a single point of failure is always a bad idea especially in a production environment when calculations are carried out upon sensitive data. The cluster administrators have to back up the image server and provide redundancy. Another point is that the whole cluster is rather susceptible to loss of power. Additional power supply is needed for a production cluster where nodes keep all the data they operate on in the RAM memory. Ram memory is erased when power is lost. Perceus supports a stateful installation, too. If nodes have a local hard drive installed and it is configured to use an active swap partition then the operating system can be swapped to the disk [Perceus, 2009].

Another feature of stateless systems that Perceus also implement is file system hybridization. This is a technique that aims at conserving memory space. Images distributed amongst the nodes in the cluster need to be as small as possible because of the limitations of both the network bandwidth and the local memory capacity. Also, images have to provide the functionality of a fully installed system. A solution is to install only those parts of the operating system that are actually used and all the other ones that are not that frequently accessed can reside on a remote location. Programs, libraries and data can be installed as non-local parts of the system image and can be hosted by another node using the NFS (Network File System) to achieve this.

Perceus installs all nodes in a cluster using a VNFS (Virtual Network File System) image. The image distributed amongst the nodes of the cluster is created by utilizing a VNFS capsule. It is a software bundle that contains all files needed to create a diskless boot image starting from a Linux distribution, hardware configuration and applications stacks. Perceus has a capsule that installs with it but one can always download a new one. An example of a capsule name is *~/CaosNSA-node-0.9-301.stateless.x86_64.vnfs.* Images are provisioned in two stages [Perceus, 2009]. Our experience shows that the whole process is straight forward and very quick. (It takes not more than a minute). Nodes must be configured to use the PXE (Preboot Execution Environment) to be able to boot from the local network. The master node runs a pxe server that waits for initial requests. Once a node boots and sends one, the server replies with proper addressing configuration and then it transfers the pxelinux and Perceus Operating System via TFTP. The Perceus OS is then booted and it starts a Perceus client daemon which issues a DHCP request to the Perceus image server (also running on the master node). At this point the master node registers that a new node is found and notes down its network hardware address (MAC) address. Perceus uses the MAC address to identify the nodes in the cluster. The master node assembles command sequence and sends it to the node. It helps the node receive the VNFS image. Once the image is transferred and prepared Perceus will execute the runtime kernel contained in the VNFS and load it into the RAM. Only then the initial Perceus OS is purged out of the memory.

3.2.1 Installing PVFSv2

The wiki page of CAOS-NSA [CAOSwiki, 2009] describes how one can install and configure PVFSv2. According to it, installation is rather simple with only a single make install command issued from the directory */usr/src/cports/packages/pvfs2/2.6.1/.* PVFSv2 is prepackaged with the configuration of the parallel environment. Source files are provided and installation comes down to compiling them. Our experience showed that this installation process also requires some tuning. The configuration on our test environment showed that three different source versions were present. Currently, the latest version of PVFSv2 is 2.8.1 [PVFS2, 2009]. This section describes its installation.

Installation process starts with making a new directory under the directory where sources are stored (*/usr/src/cports/packages/pvfs2/).* Then all files from one of the other versions have to be copied to that same directory in order to be updated to the new version.

```
# cd /usr/src/cports/packages/pvfs2/

# mkdir 2.8.1

# cp -R <directory-with-older-source> 2.8.1/
```

At this point, the *Makefile* has to be adjusted by editing the following fields:

```
    MATER_SITES=http://localhost

    VERSION=2.8.1
```

The homepage of PVFSv2 [PVFS2, 2009] contains links for downloading a compressed release of the latest version. Nevertheless, an update of the already existing files in the 2.8.1-directory is required. For this, we use the online repository for source code (via cvs – Code Versioning System).

The following commands create a folder *pvfs2* under the directory they are issued from and copy the source files of the latest development version there. It is crucial to note that these source files are still under development and might not produce a correctly working environment. Logging in the cvs server asks for a password. Users can use any password as the server grants public access.

```
#cvs -d :pserver:anonymous@cvs.parl.clemson.edu:/anoncvs login
#cvs -d :pserver:anonymous@cvs.parl.clemson.edu:/anoncvs co pvfs2
#cvs -d :pserver:anonymous@cvs.parl.clemson.edu:/anoncvs logout
```

After all files are downloaded (around 5MB) one has to rename the source directory.

```
# mv pvfs2 pvfs2-2.8.1
```

After that a compressed package has to be created from the source files of the new version.

```
# tar czfv pvfs-2.8.1.tar.gz pvfs-2.8.1
```

This package is going to be used to install the new version. Because the Makefile of the old one already states that the main site for downloading is *localhost*, all that one needs to do to make the installation available is to copy this package into the home directory of *http://localhost.* This directory can be changed by modifying the file */etc/http/conf/http.conf*. A line DocumentRoot can be configured to, for instance, DocumentRoot=*/svr/www/html.*

```
# cp pvfs-2.8.1.tar.gz /svr/www/html
```

Finally, before installing PVFSv2, the system needs to be upgraded with an additional package that contains development libraries and tools for database management.

```
# smart query db*
# smart install db4-devel
```

To install PVFSv2 it is necessary to execute make install from the directory of the source files (~/2.8.1). With the modified Makefile, installation has to download the compressed package from http://localhost, uncompress it and install it.

```
# export PATH=/sbin:$PATH
# cd /usr/src/cports/packages/pvfs2/2.8.1
# make install
```

After installation one can check whether the pvfs2 module is loaded by issuing:

```
# module avail
```

If the module does not appear to be loaded a small trick solves the problem:

```
#cp -R /usr/cports/modulefiles/nsa-1.i386/pvfs2 /etc/modulefiles
# module load pvfs2
```

At this point the module pvfs2 should be loaded and one can continue with configuration of PVFSv2 according to the documentation on the home page [PVFS2, 2009].

3.3 ROCKS

ROCKS is tool designed to simplify cluster installation and cluster deployment. It is a complete software bundle that installs everything that one might need to turn a network of computers into a production parallel environment. It can be referred to as a “cluster out of a DVD”. In contrast to OSCAR, it does not install on top of an existing operating system and what is more, it does not assume that any software configuration took place before installation. The installation process starts with formatting the hard drive and installing the operating system on it. ROCKS is tightly integrated into the distribution. This means that it is installed and also configured together with the operating system. Thus, a state of interaction is achieved that strives to maximize performance and eliminate chance of errors. The installation creates a controlled environment where the main source of errors - the user, is left out. User input is required only at certain points and consists of choosing packages to be installed and configuring the network addressing. This way, ROCKS takes complete control over of the process of building a cluster. Users do not need to start or stop services, configure tools or install prerequisite packages as in the case of OSCAR.

Additional tools for parallel computations, scheduling and mapping, monitoring, virtualization, etc. are also included in the “bundle”. They can be installed and configured at initial set up or later, when there is need for them. Like OSCAR, ROCKS also implements a separation strategy for its installation components. It uses rolls. They are not that fine grained as the binary package in the case of OSCAR - a single roll is a collection of several packages. This solution eliminates problems with dependencies and prerequisite packages as it includes all packages needed to make a certain tool or service work. Adding tools to the cluster in this way focuses on reducing the chance of errors regarding dependencies or incompatibility between installed components. Again, users are left out as they do not need to install and configure additional components. What is more, rolls are defined by their purpose and include a whole set of different tools that have the same purpose. Thus, a single installation adds to the cluster a new feature instead of a new tool. An example is the HPC roll that includes OpenMPI, MPICH, MPICH2, PVM and additional benchmarks for testing their functionality. The Torque roll includes the Torque and Maui job-scheduling systems. These two rolls together with Area51, Bio, Ganglia, Java, SGE and Xen are optional but come together with the installation DVD. The basic installation of ROCKS requires only the Kernel/Boot Roll, the Core Roll, OS Roll Disk 1 and OS Roll Disk 2 [ROCKS, 2009]. Additionally, third party rolls are developed that add support for utilization of high-speed cluster networks like Myrinet (1G, 2G) and Infiniband, or for parallel programming on graphical processors with CUDA. ROCKS has a management system that takes care of adding new components and integrating them within the configuration so no previous functionality is lost or broken.

3.3.1 Installation

At the time of writing the latest version of ROCKS is 5.1. This is the version used on our testbed to build a heterogeneous parallel environment. Installation process is well described by the user guide on the home page of ROCKS [ROCKS, 2009]. That is why in this section the description of the installation will focus on certain points that define it or may cause problems if being neglected as well as it will describe installation of PVFS on the cluster.

3.3.1.1 Environment Considerations

ROCKS is a tool that aims at building a Beowulf-type high performance cluster. Such type of cluster is defined as one which utilizes low-cost hardware together with open-source software. The goal is to achieve maximum performance at lowest price. Our experience shows that the newer the version of ROCKS is the more hardware resources it requires to work. According to the user guide [ROCKS, 2009] of the latest version 5.1 both the frontend and the computing nodes need at minimum 1GB of RAM memory and a storage space of at least 30 GB. In comparison to other high level middleware this requirements are rather costly. Today such hardware requirements cannot be considered expensive but for the purpose and our project they are rather limiting. Our experience showed that ROCKS can be installed on nodes that have less memory - in our case it was a node with 764 MB. However, after installation was complete, and when the node tried to boot up, it displayed an error message for not having enough memory. That is why one should consider which version of ROCKS to install according to the hardware in hand. In comparison ROCKS 4.3 (released in 2007) requires only 640MB of memory and 20GB of storage space, while ROCKS 4.1 (released in 2006) requires 512MB of memory and 16GB of storage space. On the other hand, one should bear in mind that ROCKS is a complete software bundle that includes the operating system, too. Initially, ROCKS was integrated into Red-Hat. Starting from version 4.0 (released in 2005) ROCKS installs and integrates into CenOS, which the open-source descendent of Red-Hat. Currently, the latest version of ROCKS uses CentOS 5.0. This distribution is relatively new (the latest at the time of writing is 5.3) and is still well supported with update and installation packages. In case, an older version of ROCKS is installed it will come with CentOS 4 and different set of updates. And installing an older distribution is not always a good idea as tools become obsolete with it and also updates are not that easy to find. On the other hand, new versions do not always provide proper support for new hardware. CentOS 5 is still being developed to support a wider range of processor architectures, while the older version 4 supports s390/s390x (IBM zSeries and IBM S/390) together with ppc/ppc64 (IBM Power, Mac), SPARC (Sun SPARC processors) and Alpha (DEC Alpha processors) [CentOS, 2009].

Another requirement for the frontend is to have two network interfaces. ROCKS implements a traditional cluster architecture according to which all computing nodes together with the frontend are connected to an internal switched network. The frontend is the only one that has a connection to the Internet and to the outside world. Hence, one network interface connects the frontend to the cluster network and the other connects it to the public one. This way, the frontend “hides” the cluster and protects it from external security threats. What is more, this setup minimizes configuration and maintenance efforts as security needs to be enhanced only on the frontend.

One should be aware that during the installation process the ROCKS asks users to fill in the initial configuration for networking. Addresses for the external and internal networks are required. It is crucial to note that the initial configuration is saved and then it is populated amongst the nodes. All host-name resolution and routing tables are configured in accordance with it. ROCKS is implemented to use only fixed static addresses for both of its network interfaces. Cluster-wide services require the frontend to always have a constant address and fully qualified domain name (FQDN). ROCKS does not work well when dynamically obtained addressing is used for any of its network interfaces. A dynamic addressing server runs on the frontend and it is configured to give always the same IP addresses to the same nodes. Functionally may be lost if users change the dhcpd configuration on the frontend and it starts giving different addresses to the nodes. What is more, the frontend itself looses internet connectivity if it receives its network configuration dynamically via DHCP. In this case networking configuration has to be manually adjusted every time a new address is obtained. The frontend will not be affected only if it receives the same configuration every time. This is why, one should make sure that if the frontend operates in a managed network it has one of its network interfaces registered with it so it can always receive the same address. When ROCKS starts installing it determines which interface is connected to the public network and suggests its settings as default settings so users need not remember them.

3.3.1.2 Installation on the Frontend

Installation of ROCKS 5.1 is straightforward and simple. The home page [ROCKS, 2009] provides a detailed user guide that describes the process thoroughly. Users can choose to download an installation "jumbo" DVD that includes the Kernel/Boot Roll, the Core Roll, OS Rolls and the rolls that install tools for parallelism Area51, Bio, Ganglia, Java, SGE, HPC, Web-Server and Xen. The installation process first detects the network interfaces so that later when the graphical user interface of the installer starts the user can choose whether to use the DVD or the network for source of rolls to install. Network installation takes much longer as all rolls have to be downloaded and this means downloading around 4GB. On the other hand, choosing the web as a source provides a broader range of rolls - with the current release it adds to the list also Condor and PVFSv2. At the time of writing PVFSv2 roll was recently completed and released for version 5.1. If the purpose of the cluster is determined at this point and one is certain about which rolls it will use, it is recommended that installation of additional tools takes place at this time. It is better to let the system configure itself and integrate the tools on its own. Adding them later, onto a working environment, can break working functionality as described in the section "Installing PVFSv2", where installation of PVFSv2 breaks the functionality of adding new nodes to the cluster.

Installation process on the frontend takes around 20 minutes when the installation DVD is used. After it completes the system reboots and CentOS 5 loads. The system has only a single user – the root user. After first login, when a terminal window is opened, the user is asked to create a security key (rsa key) to be used for ssh. It is recommended that one accepts all default values suggested and does not define a passphrase. This is important because ROCKS distributes the public key of the frontend to the nodes in order to enable secure remote login via ssh. If the key is encrypted with a passphrase, the user will be prompted to enter it every time a remote login is required. This can cause some tools for parallel computations to stop working as they do not have direct access to the nodes. What is more, one should also create a new user account different from root at this point. Tools for parallel computation require that executables are run from a non-root user account. For example OpenMPI has this requirement.

3.3.1.3 Installing the Cluster

Installation of the cluster nodes is also straightforward and follows the user guide [ROCKS, 2009]. Our testbed includes four machines of heterogeneous commodity hardware. All of them are desktop computers that have different processors, memory capacity and storage capacity. The frontend uses Intel dual core processor, while the others have processors of older architecture. In total, there are 5 processors, 6GB of dynamic memory and 470GB of storage space. All nodes are connected via a switch into a 100Mb Ethernet network. Thus, the testing environment aims at building a heterogeneous Beowulf-type cluster. ROCKS supports all the different hardware as long as the operating system, it is integrated into, supports it. In the case CentOS 5 is developed to support most of the available on the market commodity hardware (32-bit and 64-bit Pentium, AMD).

Nodes are added to the cluster in a similar fashion as with OSCAR. On the frontend one has to start the application that discovers and registers the nodes by typing in the command line: *insert-ethers*. The difference is that nodes are installed according to their functional role. Initially nodes can be "Compute" nodes but if PVFSv2 is installed, for example, nodes can be also "Compute + PVFS I/O Node", "PVFS I/O Node", and "PVFS Metadata Node". Tuning the installation according to the specifics of the individual nodes is done in a completely different manner from OSCAR. There is no "golden image" of the operating system of the frontend. The installation process uses Kickstart, which is a method for automated Red-Hat installations [Papadopoulos, 2002]. It enables administrators to specify in advance the exact software package and software configuration of a system. The kickstart method suggests that a single textual file is created containing the answers of all questions that are asked during normal interactive installation. In the common case, when the same nodes are installed over the network, a single static kickstart file can be distributed to all of them. But if nodes differ in some way they require an own specific copy of that file in order to be able to build their own image. A downside is that there is no scripting language for kickstart files and thus a single file has to be created for every node that is to be installed. ROCKS solves the situation by first distributing a script (CGI) amongst the nodes, which creates the kickstart file locally according to their specifics. Only then installation of the operating system can start and it is able to determine which packages are needed for the installation.

To download the script and start installation the nodes need to boot using the network (PXE boot) or using local media (CD, DVD). One has to configure the BIOS boot order in advance. Our testing environment showed that none of the nodes is able to utilize the network to boot. All of them had to be booted from the installation DVD. In this case, there is a difference to the installation process compared to the frontend. One should not type anything at initial prompt mode or should just press "Enter". A *vmlinux.img* and *initrd.img* are loaded together with drivers for the hardware. Only then, the system installer sends a DHCP request to the frontend. The server marks their request and replies with a static address, which is usually the last available address in the specified at installation range (for example the first given address from the range 192.168.1.0/255.255.255.0 is 192.168.1.254). At this point the screen of *insert-ethers* on the frontend must change with information about the node. The */etc/dhcpd.conf* file also registers the new node by noting its MAC address. A star at the end of the line on the screen of *insert-ethers* on the frontend marks when the kickstart script was transferred successfully. Then installation begins by downloading the operating system from the frontend.

Installation is different according to the functional role of the nodes. Using the DVD to install the frontend gives the possibility to install only Compute nodes after that. If PVFSv2 is installed nodes can be of different type. Then their installation and configuration differs from the rest. In this way one could make a cluster especially for high speed computations or make one for fast parallel data storage and management. A combination of the two is also possible by making the nodes of type "Compute + PVFS I/O Node". This, however, is discouraged according to the user manual of ROCKS [ROCKS, 2009]. Compute nodes that have scheduled jobs on them often crash and if a node crashes it brings down the whole PVFS system. What is more, the PVFSv2 roll installed on our testbed showed to crash the node-insertion process by making *insert-ethers* hang with nonsense displayed on the screen. Nevertheless, one should bear in mind, when installing different type of nodes that restarting the insert-ethers application might not be enough to start over. In the case when PVFS was tested, restarting *insert-ethers* showed that even installation of compute nodes fails after that. The solution is to reboot the frontend. That way compute nodes could be installed again.

After installation of the cluster is complete, one can inspect the overall state of the environment by launching the Web interface of Ganglia. It shows the hardware configuration of all nodes together with statistics for the load of each node. The information is presented in the form of various graphs. Ganglia gets configured and starts running from the beginning without further adjustments. In case the cluster installation process requires some manual network configuration one should be aware that Ganglia can lose connectivity to the nodes in the cluster. It reports that the nodes are down even though they are not. This is often the case when a *ping* commands are issued from the nodes. To fix this, one has to restart the ganglia daemon on all of the nodes by issuing on all of them `/etc/init.d/gmond restart`. Furthermore, network analysis done in our test environment show that the frontend generates constant UDP traffic on the cluster network. It acts as heartbeat signal that checks whether there is connectivity to the nodes and also collects information about their current load. This extra traffic might introduce certain level of network latency in an intense production environment. Nevertheless, the Ganglia monitoring tool is useful for debugging and troubleshooting on a large-scale cluster.

ROCKS 5.1 introduces another problem with Ganglia. A configuration conflict causes the monitoring tool to lose connectivity to all nodes in the cluster. What is more, after the cluster is restarted it no longer has records for any of the nodes. The frontend is the only node listed by the system and it is reported to be in a down state. In this case the first thing that one can do is to restart the main components of Ganglia – the daemons *gmond* and *gmetad*, on both the frontend and the compute nodes. However, this leads to an error being generated when the Ganglia Web interface is launched. Instead of displaying the proper page, the browser alredy shows an error message: "*Cannot find any metrics for selected cluster <cluster_name>*". According to the ROCKS support mail list this is a known issue of version 5.1 and it will be fixed in the next release. Philip Papadopoulos, who is a member of the development team of ROCKS, believes that "there is a bug in the way Xen is building their bridges in that not all routes are properly maintained". Xen is a tool that is included in the Jumbo installation DVD of ROCKS. It supports creation and maintaining of virtual machines over the cluster nodes. For this Xen installs several virtual network interfaces on the frontend causing at the same time a configuration problem - no multicast route is set on the fronted. The frontend transmits constant UDP traffic to the nodes of the cluster in order to determine their state. This heartbeat traffic is exchanged using a multicast address in the range 224.0.0.0 and port 8946. Xen bridge scripts break this functionality when they build the virtual network devices and rename them. Device specific routes (like the multicast route) turn out not to be automatically handled causing all data that is sent to *gmond* to be lost. To restore the correct functionality of Ganglia one has to add manually the multicast route every time the cluster starts. Then the Ganglia daemons have to be restarted on both the frontend and the compute nodes. As root user one has to issue from the command line

```
# route add -net 224.0.0.0/4 dev eth0
# service network restart
# service gmond restart
# tentakel "service gmond restart"
```

3.3.1.4 Installing PVFSv2

Installation of the PVFSv2 roll can take place during the process of initial installation and configuration of ROCKS or it can be added after that. Both ways of installation were tried on the test environment but none of them showed any satisfactory results. Nevertheless, it is recommended to include the roll during the initial installation process. That way ROCKS will take care of configuring and integrating it into the system. Adding it onto a working cluster requires either the whole cluster to be reinstalled or a new set of nodes to be added and installed with the new configuration. Existing nodes cannot be tuned to use the new functionality in a dynamic way. They have to be reinstalled with a new set of packages and configuration. ROCKS defines the type of a node at the time when it is added to the cluster. According to its functional role it can be either a "Compute" node or one of the following: "Compute + PVFS I/O Node", "PVFS I/O Node", and "PVFS Metadata Node". In order to preserve the capability to perform parallel computations, the nodes in the test environment had to be turned into "Compute + PVFS I/O Node".

The installation process starts with downloading the PVFSv2 roll. It is an ISO image file of size 5MB. Then in order to install it on the frontend one should execute a series of commands as root user. First, from the directory where the ISO file is located, one should issue the following command that unpacks the files of the image and places them in a proper directory hierarchy

```
# rocks add roll pvfs2*iso
```

After that the roll has to be enabled by issuing:

```
# rocks enable roll pvfs2
```

At this point one can check whether the roll is enabled by issuing `# rocks list rolls`. Only then the ROCKS distribution can be rebuilt. The distribution includes all installation packages of the operating system and the additional tools that are transferred to the nodes during their installation. The following command should be executed from the directory */export/rocks/install*. It is worth not note that this could take a while to complete

```
# cd /export/rocks/install

# rocks create distro
```

PVFS is now ready to be installed on the frontend. It is a 2 step process as the cluster database needs to be completed with an additional table. This step has to be executed manually as it requires a root password.

```
# kroll pvfs2 | bash

# mysql –u root –p cluster < /tmp/pvfs2.sql

# kroll pvfs2 | bash
```

Installation process on the frontend completes with rebooting the system. At this stage the system enables swap space on the hard drive and thus a substantial delay takes place.

```
# init 6
```

While the system is rebooting and after the swap space is enabled one can observe an error message indicating that the PVFS server cannot start. The message is "Configuration file error. No host ID specified for alias gateway". It is generated because the configuration file does not get created. Normally, the PVFS configuration file is located in */opt/pvfs2/etc/pvfs2-fs.conf*. When PVFS is installed during the initial ROCKS installation it creates this file but a message for unknown configuration options is displayed the first time the system reboots. One can try to generate a configuration file manually by issuing `#/opt/pvfs2/bin/pvfs-genconfig`, which starts an interactive-mode console application that requests the user to define a number of parameters. Generating the configuration file in this way is difficult for a non-experienced user. This is why a conclusion was reached that the configuration file must get generated when "I/O" nodes are installed and configured. At this point issuing *insert-ethers* with "Compute + PVFS I/O Node" or "PVFS I/O Node" option selected leads the application to crash with nonsense displayed on the screen. Network monitoring showed that the application crashes exactly at the moment it receives a DHCP request from a node. The node then is unable to download the kickstart script file and goes into manual configuration mode. It starts searching for a location to download the image file from. The HTTP server on the frontend hosts this file and a node can be tricked to start downloading it by specifying the location: */install/rocks-dist/i386/*. The ROCKS system installer Anaconda starts and downloads the image file. However, the node hangs just before bringing the graphical interface to the screen. After a while it reports "Cannot allocate requested partitions", "Not enough space left to create partition /boot". As a result PVFSv2 cannot be installed used on the testing environment running ROCKS.

3.3.1.5 Running an MPI Test

In order to test OpenMPI functionality one can use at first the tests that come with the implementation. ROCKS installs also MPICH and MPICH2, too. There are two separate tests present both as binaries and source codes in the directory */opt/mpitests/*. One test makes each node report the process running on it and the other makes processes send 1KB of data and then each process replies upon receipt of that data. These two tests are useful as they determine whether basic MPI functionality is running correctly. To be able to execute them, OpenMPI requires the frontend to have an extra user defined besides the root user. Adding a new user account to the frontend can be done by issuing from the command line `adduser <newuser>` and then `passwd <newuser>` to set a password for the new account. If the account is added after the nodes are installed the new account configuration needs to be populated by issuing in the command line `#rocks sync users`. Then, one should login as the new user - for example, by issuing the command `# su - <newuser>`. At this point the new user will be asked to create a security key for secure remote login via ssh. Again, as with initial configuration, it is recommended to accept all default values. One should make sure that the new user is able to login in the nodes without password being required for that. If this is not the case the new-user's public key has to be copied to the authorized_keys file on the remote machines. This can be done by issuing from the command-line

```
# scp /home/<newuser>/.ssh/id_rsa.pub <newuser>@compute-0 \
-2:.ssh/authorized_keys
```

where scp is an application for secure copying of files over the network and <newuser>@compute-0-2 specifies that the file should be copied in the directory /home/<newuser> on the node with address compute-0-2. Before issuing the command one should make sure that ROCKS synchronized correctly with the nodes and created the new user and its home directory. Otherwise, the above command will return a message that such directory does not exist.

Finally, ssh has to be configured and then a test can be run using the new user account.

```
# ssh-agent $SHELL
# ssh-add
# /opt/openmpi/bin/mpirun -np 4 -machinefile /home/<newuser>/ \
machines /opt/mpitests/bin/mpiring
```

where –np defines the number of processes that will be started on the machines. The *machines* file is a text file in the home directory of <newuser> that contains a list of the names of the nodes included in the computation process. For example, it can contain `gateway compute-0-2 compute-0-4 compute-0-5` where each of the names has to be in a new line. The `/opt/mpitests/bin/mpiring` file is the executable that makes all processes exchange 1 MB of data.

3.4 Summary

This chapter describes how one can turn a collection of desktop computers into a parallel environment for high-performance computing. It shows the process of installing and configuring a cluster using three different tools for cluster deployment – OSCAR v6.0.2, CAOS-NSA v1.0, and ROCKS v5.1. A detailed description of the installation process on our testing environment aims at revealing both the strong and the weak sides of these tools. Without going into detail, the installation process follows a similar way for all of them. First, a single computer is installed, which is intended to be the head node of the cluster. Only after being fully configured, the head node can install the nodes of the cluster by spawning an image of the operating system through the local network. All of the three tools use different techniques for creating the images. OSCAR dynamically creates an image of the current operating system and its configuration, while CAOS-NSA and ROCKS use predefined images that come with the installation. Upon installation the image is tuned in accordance to the hardware by means of configurations scripts. What is more, all three tools are designed to implement a simple cluster architecture, where there is only one Master node controlling the whole cluster and accepting jobs from users. All other nodes are dedicated to computing and, independently of the underlying hardware, have the same software installation and configuration. Thus, the head node becomes a single point of administration and, what is more, is the single point of access to the cluster – usually it is the only one that has connectivity to the Internet and the outside world. In our testing environment all computing nodes are connected with the head-node using a switched network.

Installation tests start with OSCAR v6.0.2, which at the time of testing was going through a process of renovation and migration to a completely new approach to the installation process. The software suite installs on top of an existing Linux distribution and uses its current configuration to create images for installing the nodes. Previous versions of OSCAR consist of one single installation bundle that needs to be ported and tuned for each Linux distribution while the new approach proves to be more flexible by dividing the installation into small pieces. This way, developers try to achieve broader and faster support of both RPM-based and Debian-based distributions. OSCAR allows users to choose a Linux distribution in accordance to their preferences and the requirements of hardware in hand. However, together with giving liberty to users, OSCAR requires them to have some experience in the field of clustering. Users are responsible for choosing an appropriate operating system that supports both their hardware and the OSCAR software. Also, they have to manually configure the system before and during the installation process. What is more, the current version is a development version and it is still being tested. Our testing environment showed that a working cluster can be achieved with Debian Etch as a foundation but it does not provide an installation of any tools for clustering. An approach for installing additional tools on the cluster was tested - a local repository was build. However, even with it, installation and integration of tools for parallel computation turned out to require a lot of effort and time. The production version OSCAR 6.0.2 is still far from being complete and introduces more difficulties in the process of building a Beowulf-type cluster than it eases it. OSCAR implements a strong feature set that will surely prove to be productive only when problems with support of the underlying distributions are resolved.

On the other hand, CAOS-NSA proves to be useful and easy to use when deploying a high-performance cluster. Using it, a test cluster of four computers with commodity hardware was deployed (see Table 1, Comp. 5-8). CAOS-NSA is a Linux distribution that is modified in order to provide a production environment with an operating system that is stable, reliable, lightweight, and fast. Upon installation users choose a profile for the system and it installs and configures itself for performing the tasks specific for that profile. In our testing environment CAOS-NSA is installed with tools for clustering. A major drawback in comparison to OSCAR and ROCKS, is that the installation includes only a basic set of tools for clustering. For example, MPI is supported only through OpenMPI. Additional tools have to be manually installed. Furthermore, CAOS-NSA implements a different approach to installing the nodes of the cluster. They are installed with a stateless image, which does not reside on the hard drive but is loaded into the physical memory. This way, using a small image, a cluster deployment can be achieved in a matter of minutes. What is more, this approach allows new tools to be installed on the cluster very fast as nodes need only to be rebooted and to load the new upgraded image. However, this is also the major disadvantage of using CAOS-NSA. The whole cluster is highly dependent on the head-node as it provides remote access to files and services. Additionally, should loss of power occur, the cluster has to be deployed once again. These reasons make CAOS-NSA unsuitable for fulfilling the needs of a production environment.

The cluster deployment tool ROCKS showed to be most usable of all three. A cluster of four machines of heterogeneous type was deployed and tested for the purpose of this research (see Table 1, Comp. 1-4). ROCKS incorporates an installation process that handles system configuration automatically. It installs the operating system and all tools for clustering by itself achieving at the end a fully-functioning cluster environment. After the head node is installed, cluster deployment comes down to booting the rest of the nodes. Compared to the other two tools, ROCKS turns out to be rather demanding when it comes to hardware requirements as it demands at least 1Gb of RAM and 30 Gb of storage space per node. Nevertheless, it integrates in the software environment of the cluster numerous useful tools like Ganglia, Condor, the Sun Grid Engine, MPI, etc. Our experience showed that other tools like PVFS v2 are difficult to be added to the system and do not work with this version of ROCKS.

Chapter 4
Product Evaluation

While the previous chapter describes how a cluster can be deployed using the tools OSCAR, ROCKS, CAOS-NSA, this chapter focuses on some tools that make the cluster environment useful. For this, the cluster environment created by ROCKS is studied in detail. A basic overview of the integrated tools, which facilitate job handling, submission and monitoring, is presented. The chapter discusses how they can be used and what can be achieved with them. A description of features and examples aim at revealing how these tools can be used for increasing the performance of a heterogeneous cluster environment.

The chapter is divided into four main sections dedicated to each of the tools that facilitate the utilization of the cluster. Section 4.1 describes the cluster monitoring tool Ganglia. Section 4.2 focuses on parallelism by presenting an example of a MPI application. The test application is presented first in its sequential form and then it is compared to a parallel implementation. Section 4.3 describes the batch system Condor. Section 4.4 shows examples for job submission under the resource managers Sun Grid Engine and Torque/Maui.

4.1 Ganglia

The ROCKS suite installs and configures the real-time cluster monitoring tool Ganglia. It is implemented to provide vast amount of statistical data in a simple form using a Web-based interface. Data is gathered using a multicast listen/announce technique which allows the tool to be used for monitoring clusters of up to 2000 nodes [Massie, 2004]. Resource utilization data is stored in a database and can be used for long-term analysis. In ROCKS, Ganglia v3.0.7 is integrated into the cluster software environment upon cluster deployment. Section 3.1.3 of the Installation Report (see Chapter 3) describes the process of building a small cluster of 4 nodes using ROCKS 5.1. This small testing environment is used to show how Ganglia operates and how one can use it to monitor cluster resources. Section 3.1.3 also describes why Ganglia may not be working correctly immediately after installation and gives a simple solution to this problem.

A common approach to building a cluster environment is to have a centralized point of administration and control over the cluster. The frontend node is usually used for monitoring and reconfiguring all other nodes. This, however, turns into a tedious task for administrators even in the case when the cluster is not that big, like the one in our testing environment. What is more, one can consider a case when a hardware failure has to be pointed out in a cluster of 1000 nodes. A tool that provides an overview of the hardware resources of the whole cluster proves to be more than useful. Ganglia aims at providing a simple graphical representation of all machines in a cluster or cluster of clusters. In this way, troubleshooting in large-scale clusters becomes rather achievable. An example of finding a problem would be if jobs are queued for execution in a cluster but they are not started even though nodes are available [Hoffman-Ganglia, 2003]. Furthermore, Ganglia reports on the average load of each machine with respect to its CPU, memory, network, etc. This resource utilization information is more than valuable in a heterogeneous HPC environment where tasks have to be mapped to proper machines for execution. Ganglia provides users with a fast way of determining which machines are in use and to what extent. Jobs can be scheduled based on this information to achieve better performance.

Ganglia operates using mainly two daemon programs – the ganglia monitoring daemon (*gmond*) that runs on every node of a cluster and the ganglia meta daemon (*gmetad*) which runs only on the frontend [Massie, 2004]. Additionally, it incorporates a *gmetric* tool that defines new metrics for the monitoring tool to track. It can be used to add for monitoring some application specific metrics. Also, Ganglia includes a command-line tool for executing simple distributed jobs over the cluster, named *gexec*. The *gmond* daemon is a multi-threaded program that carries out monitoring of a single cluster. It runs on every machine and implements the multicast listen/announce protocol. *gmond* collects local metrics information, sends it to the other nodes in the form of XDR (eXternal Data Representation) messages and gathers messages from others. A *collect and publish* thread in *gmond* is responsible for collecting local node information and transferring it to all nodes using a multicast address (of the range 224.0.0.0) and port 8946. The *listening* threads take care of receiving data from the multicast channel and updating the gmond's local database. Each node in the cluster maintains in-memory storage in the form of a hierarchical hash table of monitoring metrics. That way, each node has an image of the state of the whole cluster and can distribute it in case of a crash. Entries in that storage are tagged with a time of reception. If certain period of time passes information is considered to be outdated and gets deleted. Finally, a thread pool of XML *report* threads are dedicated to handling client requests for monitoring data. Additionally, the *gmond* multicasts only the metrics that are defined for monitoring and only when they exceed a certain change threshold or the time since the last transmission passes a certain time threshold. All this configuration data can be modified in the configuration file of *gmond /etc/gmond.conf*. These policies are implemented to reduce network traffic. "For example, the number of CPUs is multicast only once per hour, but the 1-minute load average might be sent as often as every 15 seconds" [Hoffman-Ganglia, 2003]. Nevertheless, network is occupied by constant traffic of heartbeat message between the nodes. This way *gmond* implements a membership protocol that aims at determining whether new nodes are added or existing nodes have failed. Heartbeat messages are exchanged periodically over a random period of time, so no synchronization takes place between the nodes.

Ganglia implements a hierarchical design allowing users to monitor clusters of clusters (see Fig.1). On the frontend of each Ganglia cluster runs another program *gmetad* which reports the state of the local cluster in the form of XML and over unicast channel [Massie, 2004]. It replies to requests from other *gmetad* programs. *gmetad* is responsible, also, for storing all collected statistical data in a Round Robin database (using the RRDTool). It can be used later on for analysis of cluster load and performance. What is more, the web front-end uses this information to present data and graphics to the web browser.

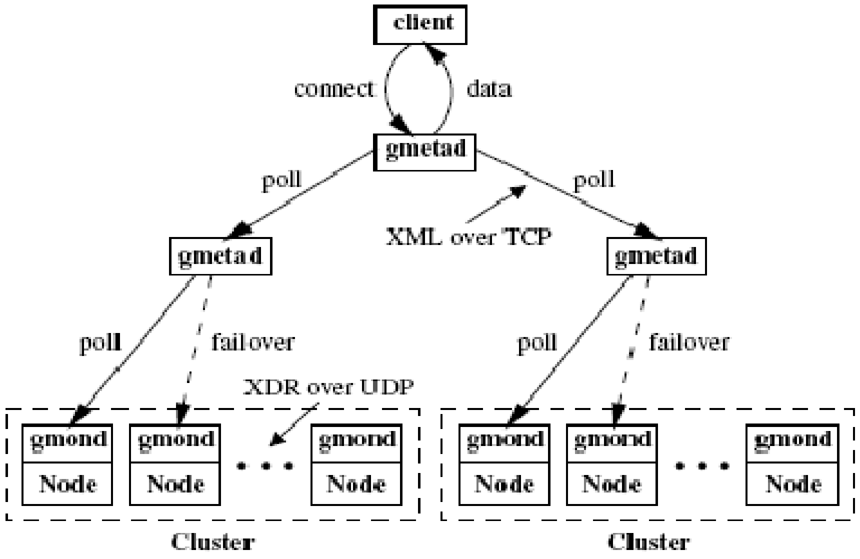
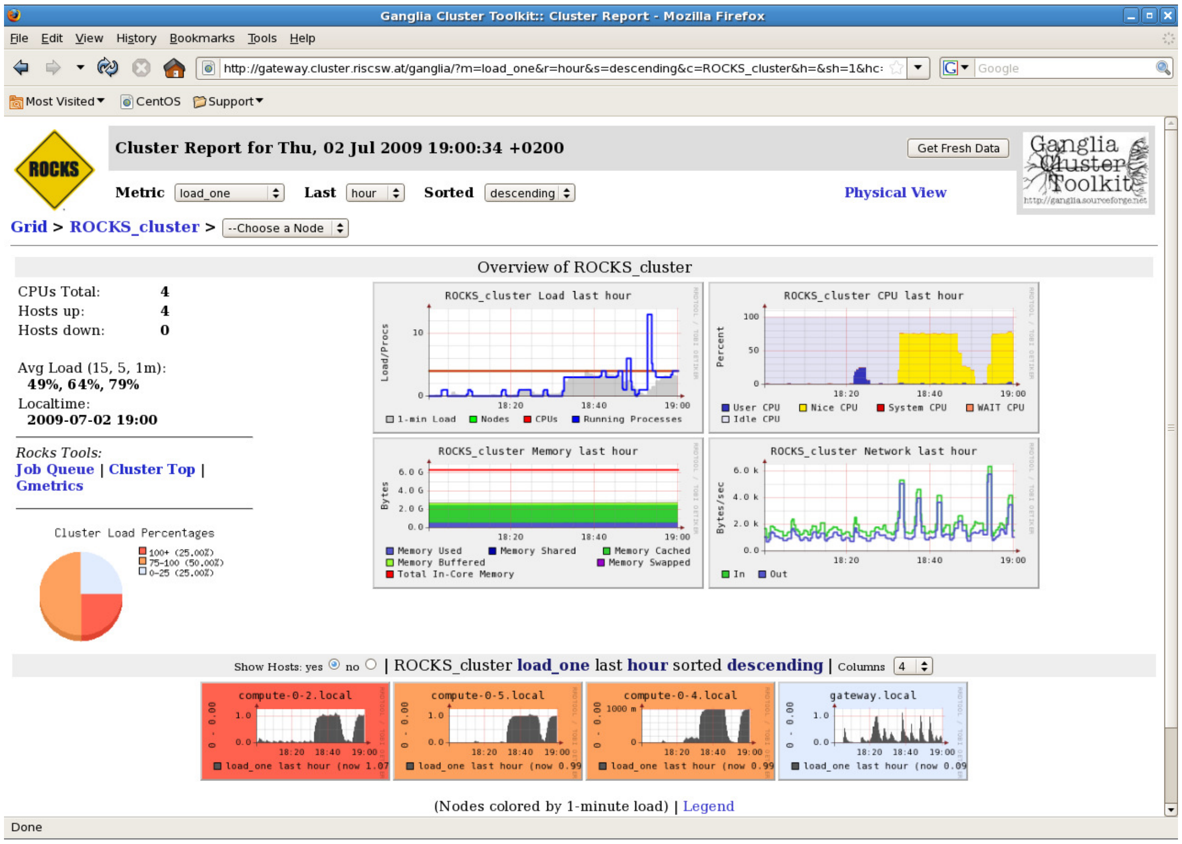


Fig.1 Ganglia architecture
Source: [Massie, 2004]

In a cluster environment created using ROCKS 5.1 the web front-end of Ganglia loads as soon as a Web browser is started. The web front-end of Ganglia provides real-time visualization of all collected data. The RRDTool is used for both storing the monitoring data in a Round Robin database and visualizing it. It generates various graphs which present how metrics change over time. They are posted on the web front-end which is implemented in PHP scripting language. Pages are dynamically generated by parsing the complete Ganglia XML tree which is obtained by contacting the local *gmetad* on port 8951 [Hoffman-Ganglia, 2003]. Fig.2 shows the contents of the starting monitoring page for our ROCKS cluster. In the center of the page four graphs always show the load for the last hour of the cluster as a whole together with the involved CPUs, the memory and the network. In the bottom of the page all nodes are depicted in a color according to their current load level (in our case there are four nodes). On the left users can find statistical data for the average load of the cluster for the last minutes (15, 5, 1) in percentage. At the upper part of the page, under the tag Metric users can choose a certain metric from a drop-down menu that causes the page to reload displaying statistical data for that metric per node. Data can be queried according to the past hour, day, week, month by choosing an option from the Last menu.


Ganglia Cluster Toolkit:: Cluster Report - Mozilla Firefox

http://gateway.cluster.riscsw.at/ganglia/?m=load_one&r=hour&s=descending&c=ROCKS_cluster&h=&sh=1&hc:

Cluster Report for Thu, 02 Jul 2009 19:00:34 +0200

Metric: load_one | Last: hour | Sorted: descending

Grid > ROCKS_cluster > --Choose a Node

Overview of ROCKS_cluster

CPUs Total:	4
Hosts up:	4
Hosts down:	0

Avg Load (15, 5, 1m):
49%, 64%, 79%

Localtime:
2009-07-02 19:00

Rocks Tools:
Job Queue | **Cluster Top** | **Gmetrics**

Show Hosts: yes [✓] no | ROCKS_cluster **load_one** last **hour** sorted **descending** | Columns 4

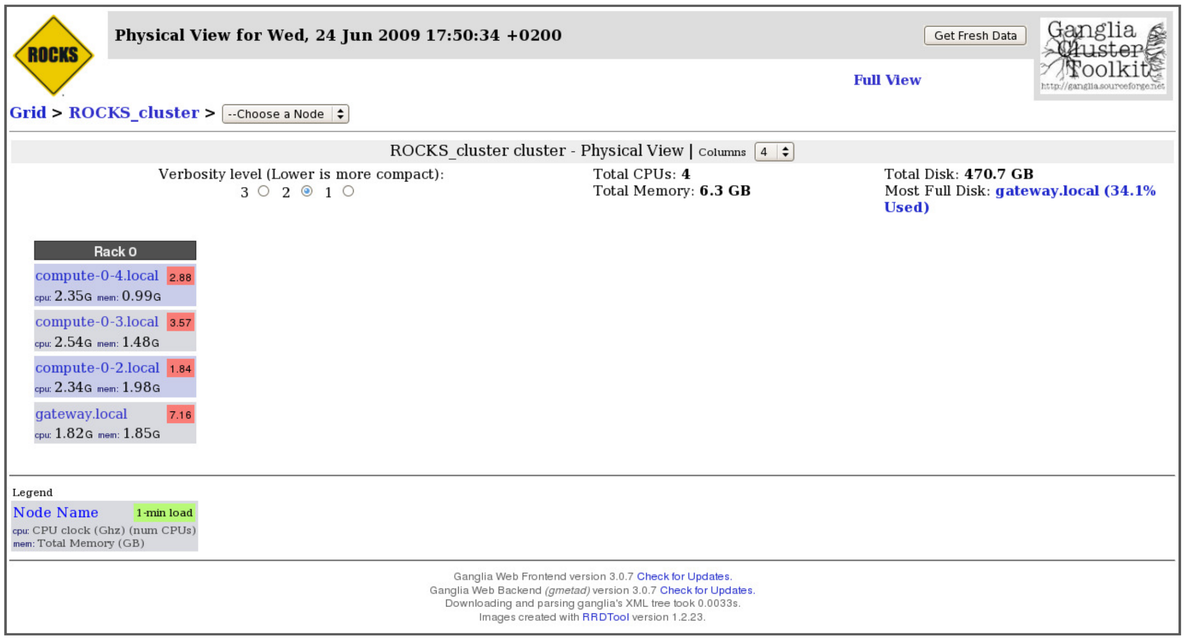
(Nodes colored by 1-minute load) | Legend

Done

Get Fresh Data
Physical View
Ganglia Cluster Toolkit
ROCKS_cluster Load last hour
ROCKS_cluster CPU last hour
ROCKS_cluster Memory last hour
ROCKS_cluster Network last hour
Cluster Load Percentages
100+ (25.00%)
75-100 (50.00%)
0-25 (25.00%)
compute-0-2.local
compute-0-5.local
compute-0-4.local
gateway.local
load_one last hour (now 1.07
load_one last hour (now 0.99
load_one last hour (now 0.99
load_one last hour (now 0.09

Fig.2 Screenshot of Ganglia web frontend on the testing ROCKS cluster

The Web front-end allows users to see data about the monitored environment on different levels that start from a general view of all monitored clusters – the multi-cluster view (Fig.2), and go through the physical view that shows only one cluster (Fig.3) until finally the node view shows detailed information about a certain node (Fig.4, Fig 5). The Physical view shows a list of all nodes with basic information about their CPU speed, total memory size and average load. It also contains summarized hardware information about the whole cluster like total memory (6.3GB in our case) and total disc space (470.7GB in our case) along with the name of the node which has most full disk.



Physical View for Wed, 24 Jun 2009 17:50:34 +0200

Grid > ROCKS_cluster >

ROCKS_cluster cluster - Physical View | Columns 4

Verbosity level (Lower is more compact):
3 ○ 2 ◉ 1 ○

Total CPUs: **4**
Total Memory: **6.3 GB**

Total Disk: **470.7 GB**
Most Full Disk: **gateway.local (34.1% Used)**

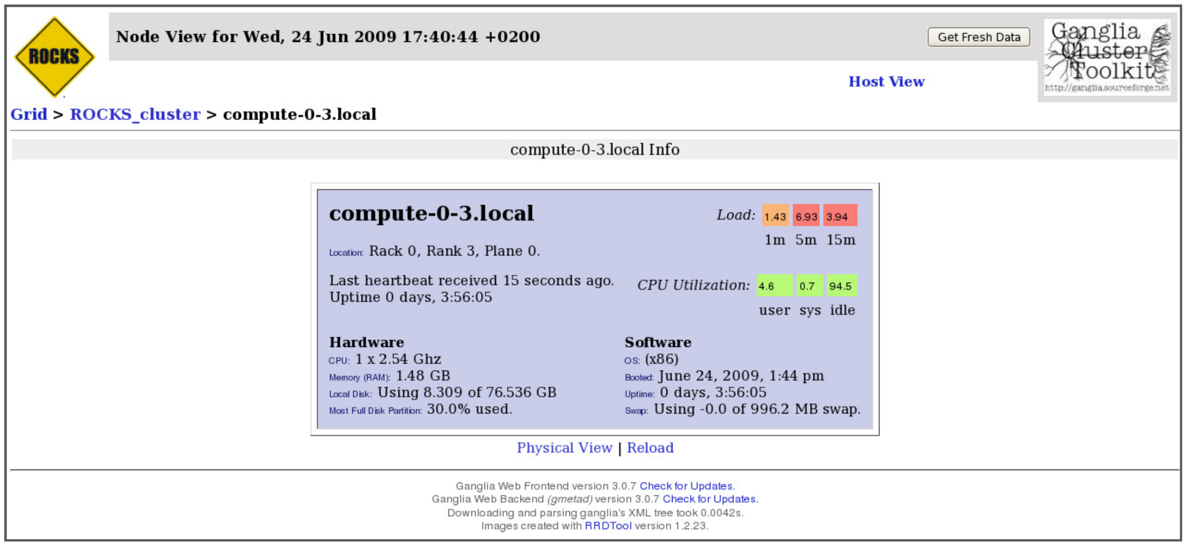
Rack 0	
compute-0-4.local	2.88
cpu: 2.35G mem: 0.99G	
compute-0-3.local	3.57
cpu: 2.54G mem: 1.48G	
compute-0-2.local	1.84
cpu: 2.34G mem: 1.98G	
gateway.local	7.16
cpu: 1.82G mem: 1.85G	

Legend

Node Name	1-min load
cpu: CPU clock (Ghz) (num CPUs)	
mem: Total Memory (GB)	

Ganglia Web Frontend version 3.0.7 Check for Updates.
Ganglia Web Backend *(gmetad)* version 3.0.7 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0033s.
Images created with RRDTool version 1.2.23.

Fig.3 Ganglia Physical view



Node View for Wed, 24 Jun 2009 17:40:44 +0200

Grid > ROCKS_cluster > compute-0-3.local

compute-0-3.local Info

compute-0-3.local

Location: Rack 0, Rank 3, Plane 0.

Last heartbeat received 15 seconds ago.
Uptime 0 days, 3:56:05

	1m	5m	15m
Load:	1.43	6.93	3.94

	user	sys	idle
CPU Utilization:	4.6	0.7	94.5

Hardware
CPU: 1 x 2.54 Ghz
Memory (RAM): 1.48 GB
Local Disk: Using 8.309 of 76.536 GB
Most Full Disk Partition: 30.0% used.

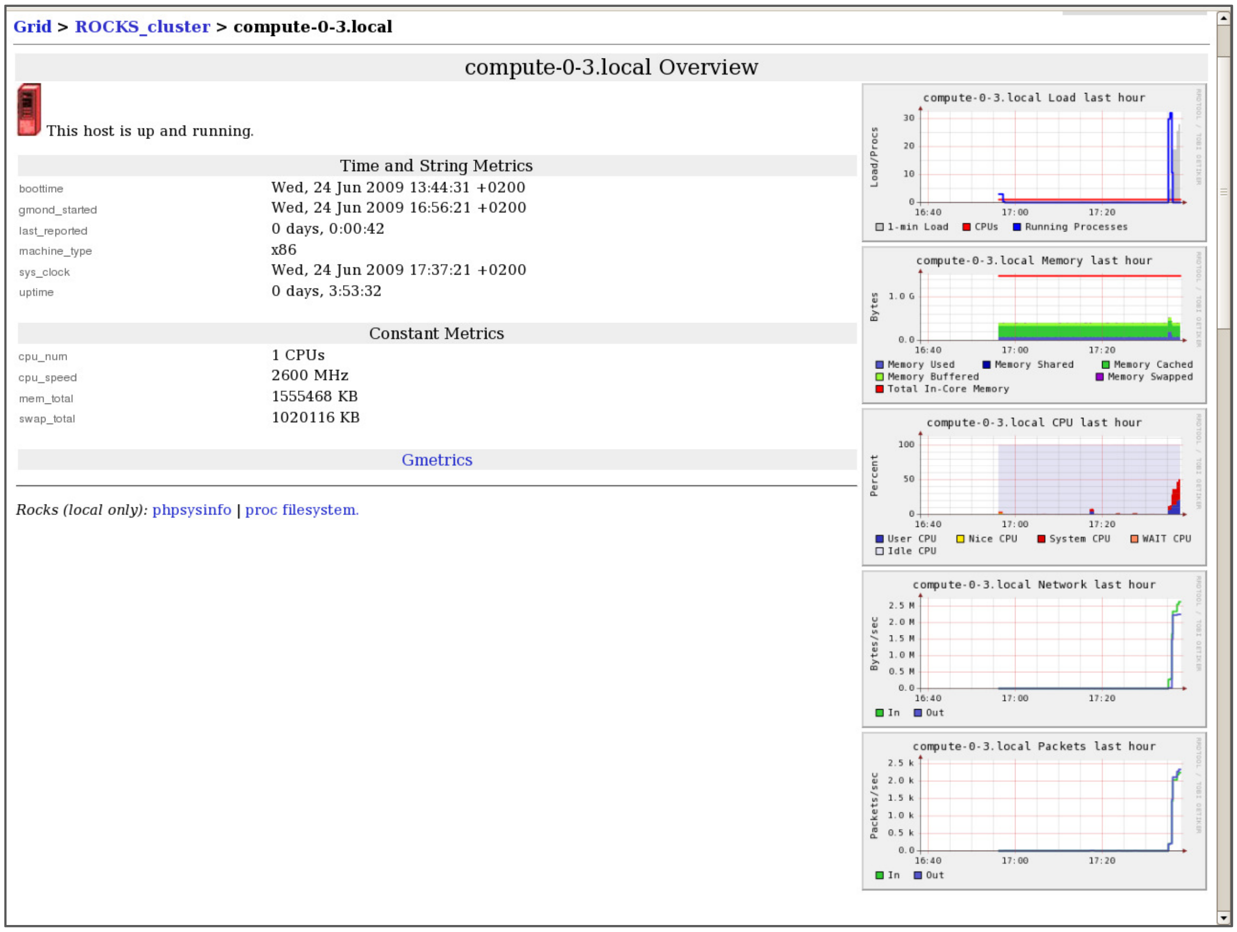
Software
OS: (x86)
Booted: June 24, 2009, 1:44 pm
Uptime: 0 days, 3:56:05
Swap: Using -0.0 of 996.2 MB swap.

Physical View | Reload

Ganglia Web Frontend version 3.0.7 Check for Updates.
Ganglia Web Backend *(gmetad)* version 3.0.7 Check for Updates.
Downloading and parsing ganglia's XML tree took 0.0042s.
Images created with RRDTool version 1.2.23.

Fig.4 Ganglia Node view

Finally, Fig. 4, Fig.5 show how Ganglia presents node information. The page on Fig.4 shows only an overview with most important information together with system average load and CPU average load while Fig. 5 shows similar information only expanded with numerous graphs that show how all monitored metrics change over time for this particular node.


Grid > ROCKS_cluster > compute-0-3.local

compute-0-3.local Overview

This host is up and running.

Time and String Metrics

boottime	Wed, 24 Jun 2009 13:44:31 +0200
gmond_started	Wed, 24 Jun 2009 16:56:21 +0200
last_reported	0 days, 0:00:42
machine_type	x86
sys_clock	Wed, 24 Jun 2009 17:37:21 +0200
uptime	0 days, 3:53:32

Constant Metrics

cpu_num	1 CPUs
cpu_speed	2600 MHz
mem_total	1555468 KB
swap_total	1020116 KB

Gmetrics

Rocks (local only): phpsysinfo | proc filesystem.

Fig.5 Ganglia Node View

4.2 MPI

The installation of ROCKS 5.1 creates a parallel environment out of a collection of interconnected machines. Upon cluster deployment OpenMPI, MPICH and MPICH2 are installed and configured to make the cluster ready for executing MPI applications. While many of the tools included into the software environment are necessary to make all machines work as a single computer, an installation of an MPI implementation is crucial for a High Performance production environment. Distributing the workload of an application among the computing nodes defines the purpose of creating a cluster. Section 3.3.1.5 of the Installation Report (see Chapter 3) describes how one can use the parallel environment. Namely, MPI code is compiled using the command *mpicc*/*mpiCC* and ran using *mpirun*. The example in Section 3.3.1.5 and all examples in this section utilize the installed OpenMPI 1.2.7 (*/opt/openmpi/*). This section, however, aims to describe a concrete example of a parallel application that shows how a parallel environment can be used to speed up an, otherwise, slow sequential application.

Our testing application solves the mathematical problem of calculating the value of a definite integral of a function. In the simplest form, a definite integral calculates the signed area enclosed between the graph of a function, the x-axis and the points within which the integral is defined. A solution can be found using Numerical Integration. Without going into detail, numerical integration is the approximate calculation of a definite integral and it incorporates several different techniques. For describing the one used, assume that the interval, over which the integral is defined, is partitioned into multiple subintervals of equal size. Then the method calculates the sum of the area of all trapezes defined by the subintervals and the values of the function in the end points of each subinterval (Fig.6)

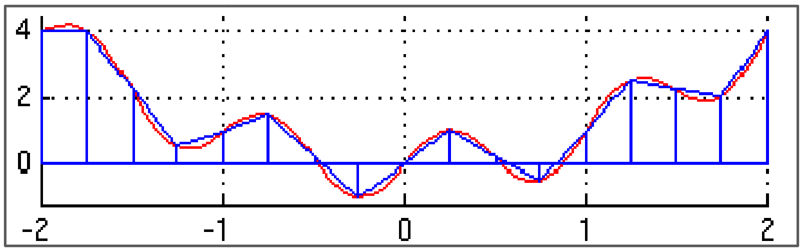


Fig.6 Trapezoidal rule for calculating the Numerical Integration of a definite integral. Source: Wikipedia

Our testing application calculates the value of the following function using the *Trapezoidal rule* for numeric integration.

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

The interval [0, 1] is divided into multiple subintervals of equal size. Then, for each interval the area of the corresponding trapeze is calculated and added to the total area. The accuracy of the calculation can be scaled arbitrary by increasing the number of subintervals. In fact, the sequential implementation relies on this to achieve as much computational time as possible. The result of the calculation is an approximation of Pi.

Paralyzing the solution to this problem is not a difficult task as the problem area provides an obvious way of dividing the work among the computing nodes. The solution comes down to calculating the area of each trapeze and then adding all areas into a single result. Hence, each computing node can be handled a portion of trapezes to compute. It is important to note here that all trapezes are defined by a subinterval of [0, 1], which are all of the same size, and the same function. This makes all of them equal for computing in the sense that the area of each trapeze can be computed independently of all others. Thus, there is no importance to the order that they are computed in. Having this in mind, parallelizing the solution can be done in two ways. The first solution suits best a parallel environment where all nodes (processors) are of the same type and have the same speed. It does not take into consideration that one processor might be faster than another but relies more on the fact that computations on all machines will take equal amount of time because of their equal performance capabilities. A second solution utilizes the possible heterogeneity of an environment as it distributes more workload to faster machines and less to slower ones. It leaves no machine idle until computations are finished.

The first solution suggests that the problem area is divided into equal-size portions that match the number of processors in hand. Each processor is handed a portion to work on. Then the results are collected onto one machine which adds them into a single, final result. This is the solution that our testing application uses as it is simple and fast to implement. What is more, it has a major advantage – it minimizes communication between the computing nodes. In an environment, where computations are carried out by separate machines connected together by means of networking, traffic between nodes can introduce significant latency to the computational time.

Considering the case when a cluster of computers utilizes a switched, 100Mbit Ethernet network, latency of constant inter-process communication can lead to great performance loss. This first solution proposes that messages are exchanged only twice during the whole computational process. What is more, traffic does not affect computations as it takes place just before actual work starts and just aster it is done. Messages need to be exchanged at the beginning to specify the portion which every node has to work on and at the end to collect the results. On the other hand, there is a second solution, which distributes work dynamically, allowing each process to fetch the next available piece of work (trapeze) as soon as it is done computing its current one.

The solution used to test the functionality of the installed OpenMPI relies on the first approach. Each processor is handed a portion of all trapezes to compute. In fact, each processor computes the area of the trapeze, whose sequential number equals the number of the last computed one plus the number of processors. Fig.7 shows a situation when 4 processors have to split the work between them. The area of the equally colored trapezes is computed by the same processor. This way of handling all pieces of work is much simpler than dividing the problem area into equal parts and then dealing with the remaining trapezes. Namely, if **n** processors are dedicated to computing there will always be **n-1** remaining trapezes after dividing the total number of trapezes to the number of processors. These **n-1** trapezes have to be assigned to processors once more. When each processor calculates the area of a trapeze after skipping **n** trapezes no remainder is introduced. Upon completion intermediate results are combined using the function `MPI_Reduce()`. It applies a reduction operation on all processors in a group and places the result in one of them. This function takes care of automatically collecting all intermediate results from the involved machines and adding them. Furthermore the application calculates its exact execution time using the function `MPI_Wtime()`. It returns on the calling processor the elapsed wall-clock time in seconds. The application calls this function two times – once at the beginning to mark the starting time and one more time, at the end of execution to calculate the difference in seconds.

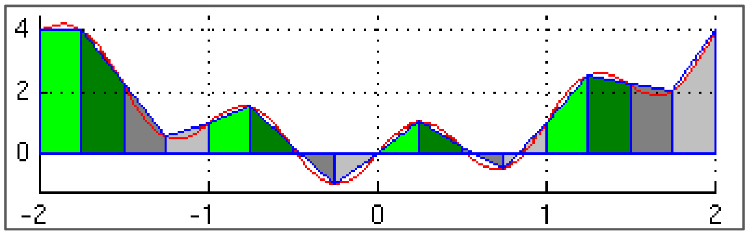


Fig.7 Dividing work among four processors

A key feature of the application is that it aims at dividing the problem area into as much pieces as possible. For this a variable NUM_PIECES is defined to hold the static value 2'147'483'647 (2^{31}). This is the maximum value that 32-bit machines have for the type `signed long int` as defined by the macros `LONG_MAX` in the header file `limits.h`. For the sake of testing, larger integer numbers can be used, too. However, in a heterogeneous environment where processors are of different architecture (e.g. both 32bit and 64bit) using the MAX values for numerical types can lead to erroneous results. The sequential implementation (see code below) shows to take rather long time to compute. It makes the computing nodes utilize all their processing power - Ganglia indicates CPU utilization of 100% on the computing nodes. Table 5 shows the measured wall-clock timings for the sequential implementation executed on the frontend and all the compute nodes. As result, the code below takes an average of **4:44 minutes** to be computed on the Intel Pentium4 2.6Ghz processors and **2:60 minutes** on Intel Core 2 1.86 Ghz.

```
for(i = 0; i < NUM_PIECES; ++i){
     x_new = (long double)(i+1) * a_step;
     x_old = i * a_step;
     area = ((((long double)4/(1+(x_new*x_new))) + ((long
double)4/(1+(x_old*x_old))))* a_step ) / 2;
     sum+=area;
}
```

	compute-0-2 Intel P4 2.4Ghz	**compute-0-4 Intel P4 2.4Ghz**	**compute-0-5 Intel Celeron 2.6Ghz**	**gateway Intel Core 2 6300 1.86Ghz**
1	284	284	252	170.76
2	279	283	256	170.17
3	274	281	258	170.52
4	288	285	260	170.62
5	289	285	264	170.83
6	288	285	264	170.54
AVG	**283**	**284**	**258**	**170.58**

Table 5 Execution wall-clock timings (in seconds) for the sequential implementation of the Numerical Integration Test application

Paralyzing the sequential code of the test application according to the described approach and techniques requires little change (see Appendix A for the full code). The code below shows the main part of the MPI implementation, which actually is responsible for handling the computations. MPI initialization and timing measurements are deliberately omitted.

```
step = 1.0 / NUM_PIECES;
for(i = myid; i < NUM_PIECES; i+=numprocs){
     x_new = (long double)(i+1) * step;
     x_old = i * step;
     area = ((((long double)4/(1+(x_new*x_new))) + ((long
double)4/(1+(x_old*x_old)))) * step ) / 2;
     mysum+=area;
     }
MPI_Reduce(&mysum,   &sum,   1,   MPI_LONG_DOUBLE,   MPI_SUM,   0,
MPI_COMM_WORLD);
```

Then the testing application has to be compiled using the OpenMpi compiler program *mpicc*. Running the program requires using the command *mpirun*. The example below is executed from the home directory of <newuser>, which can be essentially any user different from root.

```
# export PATH=/opt/openmpi/bin/:$PATH
# mpicc TestSources/NumIntegrationTest.c -o\
NumIntegrationTest.exe
# ssh-agent $SHELL
# ssh-add
# mpirun -np 4 -machinefile machines /export/home/<newuser>/
```

where –np defines the number of processes that will be started on the machines. The *machines* file is a text file in the home directory of <newuser> that contains a list of the names of the nodes included in the computation process. For example, it can contain `gateway compute-0-2 compute-0-4 compute-0-5` where each of the names has to be in a new line.

Table 6 shows the execution timings in seconds for five separate runs. The application is implemented in the way that it measures its execution time in each process separately and then, just before exiting, reports on the total execution time. The different runs, described in Table 6, prove that the parallel implementation of our testing application is faster than the original sequential one. The average execution time from five runs shows that the application executes for **73.84** seconds. Compared to the timings of sequential one this reaches a speed-up of **3.9** times which is almost the perfect speed-up an application can achieve when distributed over 4 processors. The perfect speed-up of a parallel application equals the number of processors it is distributed to.

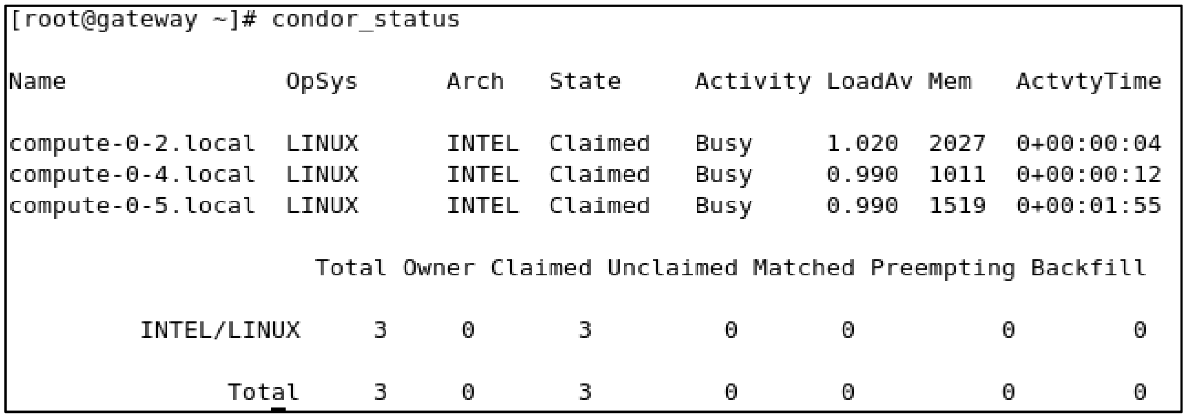
	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
gateway Intel Core 2 6300 1.86Ghz	42.94	42.94	42.93	42.93	42.93
compute-0-5 Intel Celeron 2.6Ghz	68.41	68.39	68.39	68.47	68.40
compute-0-2 Intel P4 2.4Ghz	73.07	73.08	73.06	73.07	73.08
compute-0-4 Intel P4 2.4Ghz	73.74	73.74	73.82	73.81	73.81
Total execution time	73.799	73.7979	73.8572	73.8826	73.85

Table 6 Execution wall-clock timings (in seconds) for the parallel implementation of the Numerical Integration Test application

4.3 Condor

The software bundle of ROCKS 5.1 can include an installation of the Condor roll. Users have the possibility to add it when they choose to install ROCKS using an online repository. Then, Condor v7.0.5 is installed, configured and fully integrated into the software environment of the cluster. The user manual of Condor [Condor-Manual, 2009] defines it as being "a specialized batch system for managing compute-intensive jobs". In other words, Condor is a tool that takes care of finding computing resources for executing a job in a distributed environment. Once a cluster starts having multiple users who submit jobs to it, a need arises for a tool that handles distribution of these jobs to the available machines. What is more, it is often the case that users have preferences regarding the machines that execute their jobs. For example, a certain job may require a fast processing unit while another one requires more available memory, and a third one requires a Linux operating system. In this case, the users of the cluster can start competing for available computing resources especially in a heterogeneous computing environment which incorporates computers with different characteristics. A solution is a common batch system that distributes jobs to the nodes of the cluster taking into consideration both, the requirements of the job and the characteristics of the single machines. Condor is such system that can be used in a Beowulf-type cluster environment where all computing nodes are dedicated to executing tasks and fall under centralized management. However, Condor was designed to suit the needs of large-scale distributed environments like Networks of Workstations (NoWs) or the Grid. These resources no longer fall into the characteristics of a cluster as they are usually of non-dedicated type, meaning that the single machines are normally used for other purposes (e.g. desktop computers in a company). What is more, the resources can be distributed geographically and thus belong to different networks with different management policies. Condor implements a mechanism of job migration if the current machine becomes unavailable. That way, when the owner of certain computing resources claims them back, Condor reschedules the job to another machine and continues executing it there. Before all, however, Condor is a queuing system. It maintains a common job queue for all nodes. Jobs can be submitted by any node and they are defined by their owner (the user who submitted them), their requirements and their priority. Based on those characteristics Condor decides when and where to place the jobs for execution. The order in which jobs get executed is based either on their priority or simply on the FIFO property of the queue.

The concept of ClassAds [Condor-Manual, 2009] is central to understanding how the Condor's scheduling mechanism works. The name comes from classified advertisements in a newspaper and aims to define the process of matching jobs to machines. Similarly to advertisements in a newspaper, where sellers advertise what they have to sell and buyers advertise specifics about what they want to purchase, the scheduling algorithm implemented by Condor works by matching resource advertisements issued by the machines (sellers) with job requirements issued by the users (buyers). This process involves constant exchanging of ClassAds between machines in the Condor pool of resources and its central point of administration – the Central Manager node. A pool of resources is formed by all machines that fall under the management of a Central Manager node. Each machine in the pool "advertises" its own ClassAds (machine ClassAds) by sending periodically a list of attributes to the Central Manager describing its own hardware and software profile. Attributes include CPU architecture and speed, available RAM memory, disk size, virtual memory size, current load average, name of the machine, operating system, etc. What is more, machine ClassAds are designed to facilitate scheduling of jobs on non-dedicated nodes by providing a possibility for the owners of the nodes to define under what conditions the resources can be used and what type of job can run on them. For example, machine ClassAds can be tuned to advertise that a machine is available only at night time or when there is no keyboard activity. Furthermore, it can specify the type of jobs a machine accepts like a user who submits them or a rank of a job. On the other hand, upon job submission, users can define a number of requirements and preferences for where the job should be executed. For example job ClassAds can include a requirement for a machine that has at least 1Gb of RAM memory or one that uses certain CPU architecture. The Central Manager collects all the machine ClassAds and according to the job ClassAds of the first non-running job in the job queue computes its best match. One can see a simple summary of the machine ClassAds by issuing *condor_status* from the command-line interface on the Central Manager node. Fig. 8 shows the output in our testing environment that has only 3 computing nodes. The command "*condor_status –l <node_name>"* gives a detailed list of the machine ClassAds advertised by a node. And the command "*condor_q –l <jobID>*" returns all ClassAds specific for a job.



```
[root@gateway ~]# condor_status

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

compute-0-2.local  LINUX      INTEL  Claimed   Busy     1.020  2027  0+00:00:04
compute-0-4.local  LINUX      INTEL  Claimed   Busy     0.990  1011  0+00:00:12
compute-0-5.local  LINUX      INTEL  Claimed   Busy     0.990  1519  0+00:01:55

                     Total Owner Claimed Unclaimed Matched Preempting Backfill

         INTEL/LINUX     3     0       3         0       0          0        0

               Total     3     0       3         0       0          0        0
```

Fig. 8 Output of condor_status command

Condor operates by maintaining a number of daemon programs running on the machines in a pool [Hoffman-Condor, 2003]. The *condor_schedd* is the daemon that allows jobs to be submitted from a machine. It sends job ClassAds to the Central Manager and usually runs on all computers in the pool. The *condor_startd* is the daemon that makes a node an execute machine as it allows jobs to be started on it. It sends periodically machine ClassAds to the Central Manager and communicates with the scheduler program running on it. Any node in the pool can be an execute machine, even the Central Manager. Furthermore, the Central Manager is responsible for collecting all job and machine ClassAds and for matching them in the proper way. For this several daemons run only on it. The *condor_collector* is part of the Central Manager and is responsible for collecting all advertisements from both submit and execute machines while the *condor_negotiator* performs all match-making between jobs and resources. The last communicates with both the *condor_schedd* and the *condor_startd* sending requests to them for ClassAds. For every job submitted from a machine a *condor_shadow* daemon is started locally. It watches over the job providing it with access to the local resources (e.g. files) and handling system calls. All daemons running in a Condor pool are controlled by a single top-level program – the *condor_master*. It runs constantly and ensures that all other daemons are running correctly. If they hang or crash, it restarts them.

4.3.1 Jobs and Condor

Condor is designed to ease job management in a distributed environment by giving users the freedom to submit jobs of various sorts. In most cases executables do not need to be changed as long as the local software environment is able to run them. Code written in C, C++, FORTRAN, JAVA, or using MPI does not need further recompilation and can be submitted directly to the system. For this Condor implements a mechanism of supporting numerous run-time environments referred to as universes. Each universe aims at tuning the environment for executing specific code. For instance, JAVA needs a virtual machine to be started and MPI might require some environment variables to be adjusted. Condor v7.0.5 supports several universes for user jobs: Standard, Vanilla, MPI, Grid, Java, Scheduler, Local, Parallel, VM. Of course, each universe is used according to the specifics of a job but the most commonly used ones are the Standard and Vanilla. In our testing environment the Parallel universe was also tested. Although Condor provides certain freedom to the kind of jobs that it can schedule, it also introduces some restrictions to how jobs should operate. Jobs must be able to run in background. After scheduling a job to a certain machine Condor leaves it unattended and running in the background. This way jobs that require interactive input and output cannot be executed correctly. Hence, it is recommended that jobs are implemented in a way that they read all their input from a file. In a case where only a few simple arguments are needed Condor provides a way of redirecting the standard input (stdin) to a file where all necessary keystrokes can be coded. It also takes care automatically of redirecting the console output (stdout and stderr) to files on the submission machine.

The Standard and Vanilla universes are the two most commonly used execution environments. However, they can be used for submitting sequential programs only. While the Standard universe provides mechanisms for job migration and remote system calls, the Vanilla universe is used for more simple applications that do not require much I/O operations [Condor-Manual, 2009]. It is particularly useful for execution of shell scripts. Test runs in our testing environment showed these universes to be equivalent when used in a cluster environment where all nodes are dedicated to execution. Furthermore, the Grid universe provides an interface for submitting jobs intended for remote management systems. The Java universe can run a job on any machine that has a JVM running, regardless of its location or owner. Additionally, the Scheduler and Local universes schedule jobs to be executed directly on the submitting node. And, the Parallel universe takes care of executing parallel code written in MPI. Finally, the VM universe facilitates the execution of jobs that are not a single executable but are a disk image and require a virtual machine to be executed (e.g. VMware and Xen).

In more detail the Standard universe is intended to provide support of execution of sequential programs in a non-dedicated environment where each machine has its owner and is used for performing a specific task. Compared to the Vanilla universe, it supports the mechanism of job migration which ensures that a job reaches its completion point even if the initial machine, which it started executing on, fails. Condor implements this by a technique called check-pointing. A checkpoint image saves the current state of a program by saving its Program Counter (PC) and the memory block it occupies. This way a job can be moved around the different available machines until it finally completes. Check-pointing takes place automatically at regular intervals but it can be also forced by the commands *condor_checkpoint* and *condor_vacate*. Checkpoint images get transferred back to the Central Manager. Depending on the particular application they can be rather big in size. That is why it is often the case that in a large-scale environment a checkpoint server stores all checkpoints for the pool. Additionally, the Standard universe also implements a mechanism of remote-system calls. This mechanism provides uniform access to the resources of the submitting machine from any other node in the pool. The job perceives that it is being executed on its home machine because remote system calls transfer each request for local resources to the *condor_shadow* running for that job on the submit machine. The daemon program executes the request and sends back the result to the execute machine. For instance a job requires a file stored on the submitting machine to be opened and read. Then the condor_shadow program will find this file, read it and will send the contents to the computer executing the job. In this way the Standard universe handles file I/O operations easily and without the need of additional file system like for example the NSF. In contrast, the Vanilla universe does not support remote system calls and thus needs to use either a shared file system or to transfer the input and the output files between the submitting and the executing machine. For this Condor implements a mechanism for transferring files on behalf of a user.

On the other hand the Standard universe has some major drawbacks [Condor-Manual, 2009]. First of all, an executable can be run under this environment only after it is re-linked to use the Condor libraries. This is the only way that support for check-pointing and remote system calls can be added to the application. Nevertheless, re-linking is rather simple using the *condor_compile* command. Furthermore, standard jobs are quite limited at certain system level as no multi-process jobs are allowed (system calls such as fork(), exec(), and system()) together with no inter-process communication, no sending and receiving of thr signals SIGUSR2 and SIGTSTP, no alarms, timers, and sleeping, etc. [Condor-Manual, 2009]. In comparison, the Vanilla universe has no restrictions and what is more, does not require any changes to an executable.

4.3.2 Submitting a Job

This section describes how one can submit jobs to Condor. The examples and all tests are run under our ROCKS cluster environment. ROCKS installs and configures Condor in the way that the frontend node is the Central Manager and also is the only entry point for job submission in the cluster. All computing nodes (in our case compute-0-2, compute-0-4, compute-0-5) form the pool of resources. In addition, ROCKS installs a user account named condor with a home directory */export/home/condor*. One can either use this account or any other to submit jobs.

Jobs are submitted to the system with the command *condor_submit* [Condor-Manual, 2009]. It is a complex command but in the most common case it takes only one parameter – a submit-description file. This file is a simple text file that describes the characteristics of a job. Usually it consists of several lines that control the details of job submission like what executable to run, the name of the files used for input and output, simple list of arguments, job requirements. It is crucial to be noted here that job ClassAds are specified in the submit-description file. The following examples show how our sequential test application NumIntegrationTest is submitted to the system starting with the Vanilla universe. A very simple submission file has the following contents:

```
#Example 1
Executable      =    NumIntegrationTest.exe
Universe        =    vanilla
Output          =    NumIntegrationTest.out
Error           =    NumIntegrationTest.err
Log             =    NumIntegrationTest.log
Queue
```

The above contents are saved in a file *~/NumIntegrationTest.submit*. They specify the name of the binary (NumIntegrationTest.exe), the type of the runtime environment (vanilla) and the names of the files that will hold the contents of the standard output. If the `universe` command is omitted then Condor will use the Standard universe by default. Condor automatically redirects the stdin, stdout, and stderr of any job to files. These files are usually stored in the home directory of the submitting user. If no names are defined for any of the commands `input`, `output`, or `error` (like in this example there is no name for the input) the stdin, stdout, and stderr will refer to */dev/null*. A log file is also generated that holds the information about what happened to the job during its lifecycle inside Condor. It is very useful for debugging and troubleshooting. Furthermore, this simple example does not define any requirements for the platform the job should be executed on. In this case, Condor assumes that the job has to be executed on a machine with the same architecture and operating system as the submitting machine.

The next example shows a slightly modified subscription file that is used for instantiating 10 different jobs that execute the same binary NumIntegrationTest.exe.

```
#Example 2
Executable      =    NumIntegrationTest.exe
Universe        =    vanilla
Output          =    out.$(Process).NumIntegrationTest
Error           =    err.$(Process).NumIntegrationTest
Log             =    NumIntegrationTest.log
Initialdir      =    tests
Queue 10
```

This example aims at demonstrating how flexible the syntax of the submit-description file can be. First of all, starting from the last line, it places 10 jobs in the queue, all of them executing the same task. To differentiate between output of the different jobs the files are named using the sequential number of each job (`$(Process)` is replaced with a number) resulting in creating 10 out files (out.0.NumIntegrationTest, out.1.NumIntegrationTest,…) and 10 err files. This time, however files are placed in a separate directory */export/home/<user>/test* defined by the command `Initialdir`. One can notice that the log output remains directed to a single file. This way monitoring of job status is eased because the information about all jobs is printed in one place. What is more, troubleshooting is also eased in this way. Finally, the job is submitted using the condor_submit command

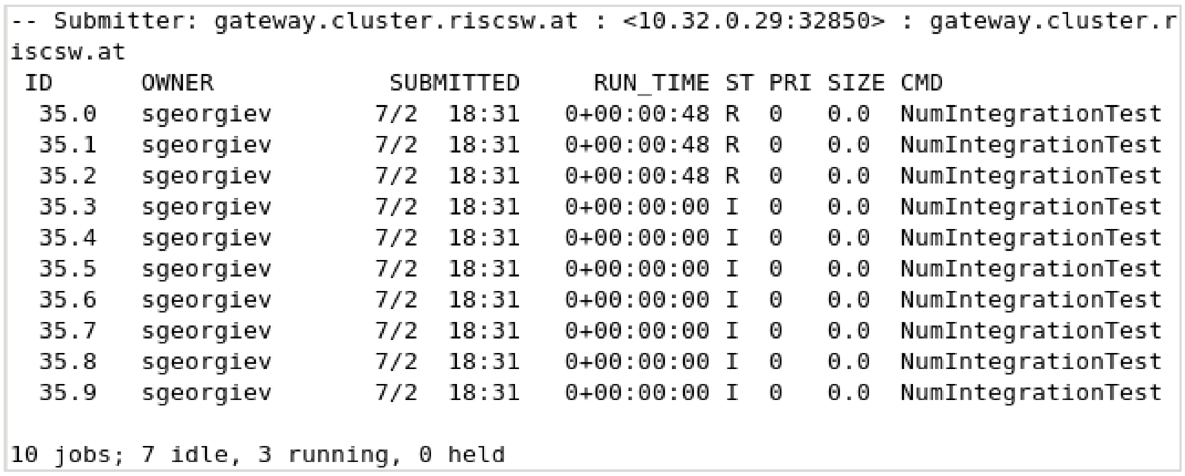
```
# condor_submit NumIntegrationTest.submit

Submitting job(s)...

Logging submit evet(s)...

10 job(s) submitted to cluster 35.
```

Once a job is submitted to the system, there are several ways to monitor its execution lifecycle. The command `condor_q` is the starting point of every monitoring process over a job [Condor-Manual, 2009]. It displays information about jobs in the Condor job queue. Issued without any options or arguments, it lists all jobs that are currently in the queue together with their status, current runtime, priority, owner, and ID. An “R” in the status column means the job is currently running. An “I” stands for idle - the job is not running right now, because it is waiting for a machine to become available. The status “H” is the hold state. In the hold state, the job will not be scheduled to run until it is released. `condor_q` is useful as a starting point when performing troubleshooting on a job. One can immediately notice when a job remains in the idle state even when there are available resources in the cluster. Fig.9 shows a screen shot taken on our testing environment after submitting the job in the above example.



```
-- Submitter: gateway.cluster.riscsw.at : <10.32.0.29:32850> : gateway.cluster.r
iscsw.at
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
  35.0   sgeorgiev       7/2  18:31   0+00:00:48 R  0   0.0  NumIntegrationTest
  35.1   sgeorgiev       7/2  18:31   0+00:00:48 R  0   0.0  NumIntegrationTest
  35.2   sgeorgiev       7/2  18:31   0+00:00:48 R  0   0.0  NumIntegrationTest
  35.3   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest
  35.4   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest
  35.5   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest
  35.6   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest
  35.7   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest
  35.8   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest
  35.9   sgeorgiev       7/2  18:31   0+00:00:00 I  0   0.0  NumIntegrationTest

10 jobs; 7 idle, 3 running, 0 held
```

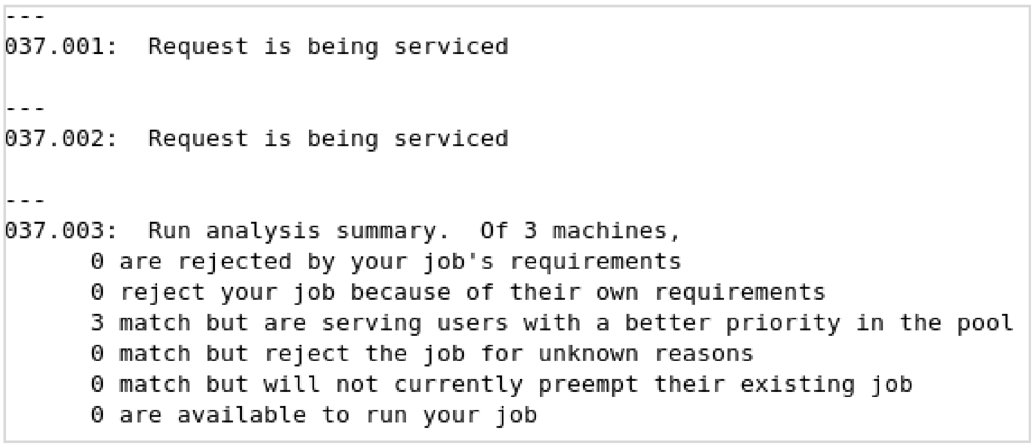
Fig. 9 Output of condor_q command

Another very useful approach to monitoring a job's status is through the user log file. If the submit description script of the job contains the command "log", a log file is generated for a job or a set of jobs. It contains a detailed list of all events that take place during the execution lifecycle of a job. What is more it can be used for real time monitoring through the command "tail -f" as the file is generated as soon as the job is submitted. The file contents follow a certain formatting model when describing the events [Condor-Manual, 2009]. Four fields are always present in the description. The first one is a 3-digit numeric value that describes the event type. Condor defines 28 values that describe different types of events. The most basic ones that are present in every log file are 000 indicating that a job was submitted; 001 - a job started executing; and 005 - job terminated. The second field identifies the job uniquely by specifying in parenthesis the ClassAd job attributes of ClusterId value, ProcId value, and the MPI-specific rank (if executed in the MPI universe, otherwise zeros). The third field marks the time the event took place and the fourth field contains a brief description of the event. The log file is particularly useful for determining immediately on which node a job gets executed. This can be achieved using the condor_status command as well. One can list the names of all machines that run a job submitted from a certain user by issuing

```
# condor_status –constraint 'RemoteUser="<name_of_user>"
```

What is more, upon job completion the description of the termination event contains exit code and status together with statistical data like total time of execution, bytes send and received. Furthermore, the contents of the log-file indicate when a job is suspended or resumed, when the check-point image is captured or updated and when errors occur because of a bad executable.

When a job is submitted to the system but it remains in the idle state even though resources are available for executing it, Condor provides a mechanism of defining automatically what the cause might be. The command condor_q issued with the options –analyze and –better_analyze tries to determine why a job is not running by performing analysis of all machines in the pool. The "-analyze" option tries to determine how many resources are available for executing the job. "The reasons may vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the PREEMPTION_REQUIREMENTS expression" [Condor-Manual, 2009]. The option "-better_analyze" performs a more time consuming analysis, which is also more extensive and finds how many resources are available for a job. When it comes to analyzing jobs in large-scale networks of machines, the Condor manual recommends issuing "-better_analyze" for only one job as it consumes significantly more resources. The screenshot in Fig. 10 shows the output of "condor_q -better_analyze" in a moment when the job queue has six jobs (37.000 – 37.005) and the first three are running while the rest are in idle state waiting for available resources (our testing environment consists of only three machines).



```
---
037.001:  Request is being serviced

---
037.002:  Request is being serviced

---
037.003:  Run analysis summary.  Of 3 machines,
      0 are rejected by your job's requirements
      0 reject your job because of their own requirements
      3 match but are serving users with a better priority in the pool
      0 match but reject the job for unknown reasons
      0 match but will not currently preempt their existing job
      0 are available to run your job
```

Fig.10 Output of conor_q –better_analyze

A final option of tracking the progress of a job is to use the built-in mechanism of sending notifications to users upon job completion. One can set notifications to be send to the submitting users in the form of an e-mail message. The command "notification = < Always | Complete | Error | Never >" can be added to the submit-description file. It sets the system to notify users when certain events take place. If the argument is "Always" e-mail messages will be send whenever the job produces a checkpoint, as well as when the job completes. If defined by "Error" users are notified only when a job exits abnormally. The default e-mail address used is `job-owner@submit-machine-name`. It can be configured using the command "`notify_user = <email-address>`" that has to be defined also in the description file. The message itself contains the job's exit status together with a lot of statistical data like timings of running, check-point status, I/O statistics.

A job can be submitted to the Standard universe using the same submit-description file as in the case of using the Vanilla universe. The only difference is that the key word "standard" has to be specified. However, the binary file used to initiate a job must be re-linked with the libraries of Condor. This can be done using the command `condor_compile` [Condor-Manual, 2009]. It is used in a similar way to compiling source code files using a conventional compiler. If the program is fully-installed on the system it allows users to enter any command or program, including `make` or shell-script programs. Otherwise, users are restricted to use only one of the following programs - `cc` (the system C compiler), `acc` (ANSI C compiler, on Sun systems), `c89` (POSIX compliant C compiler, on some systems), `CC` (the system C++ compiler), `f77` (the system FORTRAN compiler), `gcc` (the GNU C compiler), `g++` (the GNU C++ compiler), `g77` (the GNU FORTRAN compiler), `ld` (the system linker), `f90` (the system FORTRAN 90 compiler). Because our testing application uses the programming language C it is compiled like

```
# gcc -c NumIntegrationTest.c -o NumIntegrationTest.obj

# condor_compile gcc NumIntegrationTest.obj -o          \
NumIntegrationTest.exe
```

Condor facilitates utilization of heterogeneous parallel environments with the `condor_submit` commands `requirements` and `rank`. They can be specified in the submit-description file and are a powerful tool for defining job ClassAds. Users can set numerous preferences regarding the platform that is going to run their applications. The requirements command defines the job requirements. Using these, Condor filters the list of available resources and creates a set of machines that match these requirements. A job can be run on any one of them. The list can be sorted additionally using the ranking criteria defined by the `rank` command. That way, a perfect match can be found for a job. In the case when a heterogeneous environment is used to execute jobs, users can help the system to place their executable on the most appropriate machine. The `requirements` command can resolve expressions that evaluate to being true or false (boolean expressions), which are written in a similar way to how they would be written using the programming language C. These expressions can define the processor architecture, available physical memory, operating system, etc. An example is

```
Requirements   =    Memory>=1000 && ARCH=="INTEL"
```

which defines a requirement for machines to have at least 1Gb of physical memory and their processors to be of x86 architecture. The tags "Memory" and "ARCH" are classID tags. They are case insensitive, while the values are case sensitive. All available classID tags can be displayed using the "`condor_status -l`" command issued from the command-line interface. What is more, before constructing a requirement, one can check its validity by displaying all machines in the pool which fulfill it. This can be achieved by issuing "`condor_status - constraint <Boolean expression>`". Furthermore, when users specify ranking scheme in the submit-description file, Condor asserts each machine according to that rank value. The one with the highest value is matched to the job. For instance,

```
Rank =    memory
```

causes a job to be matched to the machine with the highest amount of physical memory available. Another example is

```
Rank =    kflops
```

which causes the machines with most powerful floating point processor to be chosen [Condor-Manual, 2009]. This example shows how ranking can lead to scheduling a job to the wrong machine. While all commodity machines today have a processing unit that is dedicated to managing floating-point operations, not all of them have the kflops attribute defined. In a case, when there are machines in the pool that do not have this attribute set, the ranking mechanism will only assert the ones that have it defined. That way a machine with the fastest floating-point capabilities could be left out. To prevent this from happening, users have to check the list of machines against the criteria. This can be done using the command "`condor_status -constraint <boolean expression>`".

4.3.3 Modifying a Job

Condor is a system which makes sure that heavy computational tasks get completed. Sometimes, however, a running job may start producing erroneous results or it may take too long time to complete. In these cases (and many others) administrators of the parallel environment may have to alter a job's behavior in order to free resources for other applications. Condor provides some commands that can be used to cancel a job, pause it, resume it or reschedule it.

A submitted job can be cancelled and removed from the queue using the command `condor_rm` [Condor-Manual, 2009]. It takes as an argument a job Id to stop a certain job; a username to stop all jobs started from this user. A very useful option of the command is the "-constraint", which removes all jobs that match the job ClassAd expression constraint. For example, the following line removes all jobs of the user named sgeorgiev that are not currently running.

```
# condor_rm sgeorgiev -constraint Activity!=\"Busy\"
```

Jobs can be made to free the occupied computing resources and return in the queue. Condor provides users with the command `condor_hold` that puts a job in the hold state [Condor-Manual, 2009]. Jobs remain in the queue waiting to be resumed by a `condor_relese` command. When one puts a job in the hold state, this causes a hard kill signal to be sent to the machine executing the job. For a Standard universe job, this means that the job is stopped without allowing it to update its check-point image. When resumed it continues from the last checkpoint image. In the case when the job was running under the Vanilla universe, it starts simply starts over. Both commands take as an argument a job Id or a name of a user. If no user name is specified both commands assume, by default, that they have to manipulate only the jobs of the user that issued the command. In the following example all jobs of the user sgeorgiev are put on hold together with the ones that are not currently running.

```
  # condor_hold sgeorgiev -constraint "JobStatus!=2"
```

4.4 Sun Grid Engine and Torque/Maui

This section provides a basic overview of how one can use resource managers in the production environment created with the installation of ROCKS. Series of examples aim at providing a basic understanding of how pieces of work (jobs) can be submitted to the cluster, how they can be monitored and how they can be modified.

Upon installation ROCKS 5.1 includes into the software environment of the cluster a resource management tool - the Sun Grid Engine (SGE) v6.1u5. It gets fully configured and after cluster deployment it is ready to be used without further configuration. ROCKS provides a possibility for installing a second popular tool for resource management. The roll with Torque/Maui is available for download on the main page of ROCKS [ROCKS, 2009]. It needs to be manually installed (instructions are available on [ROCKS, 2009]) and adds to the environment the resource manager Torque v2.3.6 together with the scheduler Maui v3.2.6. The two resource managers operate in a way similar to the way the batch system Condor operates as they both implement a mechanism of matching pending workload to the available resources of the cluster environment. It is worth mentioning here that a resource manager is different from a scheduler although they both manage and distribute jobs. Resource managers usually have a scheduler integrated with them. In the case of Torque it has a simple built-in scheduler *pbs_sched*, but its design aims at providing a common interface that facilitates utilization of different schedulers (like Maui in our case). The SGE also has a scheduler included in its implementation, which allows jobs to be scheduled over time (when to start and how long to run). A scheduler program implements a way of finding a proper order for executing jobs that corresponds to their priority and timing preferences and is in accordance with the availabe resources. On the other hand a resource manager focuses on providing a low-level functionality of managing job queues, starting and stopping jobs, monitoring their status. A scheduler alone cannot control jobs. Furthermore, another similarity between the two resource management tools is that they both inherit from the outdated resource manager and scheduler OpenPBS (Portable Batch system). Torque is a direct derivative of OpenPBS and still implements an architecture that consists of a Master Node and Compute Nodes via the daemon programs *pbs_server* and *pbs_mom*. The SGE, on the other hand, builds a different environment using a Qmaster daemon program that resides on one node and a Qmaster shadow daemon that handles overhead load and resides on some of the execution nodes. What is important, in this case is that both Torque and SGE use the same technique of job submission (using a shell script) and utilize the set of same commands for job handling (there are small differences).

The Torque/Maui roll needs to be manually added to the software environment created by ROCKS 5.1. Installation is simple and includes several commands [ROCKS, 2009]. It starts with installing the roll on the frontend and then the image of the other nodes in the cluster needs to be updated. This requires all nodes to be reinstalled. Our experience showed that all this takes around 30 min. After downloading the installation roll (10MB) on the frontend the following commands install it and then renew the image of all compute nodes. Finally, installation can be verified by the command "`pbsnodes -a`", which lists all known to the system nodes together with their attributes (state and hardware configuration).

```
# rocks add roll /path/to/torque/roll
# rocks enable roll torque
# cd /export/home/install
# rocks create distro
# kroll torque | bash
# reboot
# tentakel /boot/kickstart/cluster-kickstart
```

Both SGE and Torque manage submitted jobs by using a queue. A queue determines where and how jobs are executed. Different scheduling policies (e.g. job priority, user priority, job requirements, etc.) determine which of all submitted jobs will be matched to available resources first. Torque implements a mechanism of supporting different queues with different properties. Usually there is a single queue that spans on all hosts and all submitted jobs are placed in it. Using the command *qmgr* [Torque-Manual, 2009] users can create additional queues that for example allow only certain users to submit jobs to them, or match jobs only to a subset of the execution hosts. SGE, on the other hand, implements a different mechanism for achieving this [SGE-Guide, 2008]. A queue is as an abstraction that aggregates a set of job slots on one or more execution hosts. Job slots are defined as the capacity of a node for executing a job. Normally, the system assigns a slot to each available processor or core on an execution node. That way a queue determines the distribution of jobs to available slots. The SGE supports creation of different queues that can be configured to utilize, for instance, only certain job slots or to be used only by certain users (Fig. 11). The `dnetc.q` is configured to use only one slot per execution host. After installation of SGE, a default queue is automatically defined (called `all.q`) that spans on all hosts in the cluster and can use all available job slots. By default, all jobs, submitted by all users get assigned to it. Being an abstraction, a queue in SGE operates by managing the queue instances that it consists of. A queue instance is defined by the execution host it is associated with. It manages its free slots. The name of a queue instance is formed by the name of the queue it belongs to and the name of the host it describes. In Fig.11 there are seven queue instances: all.q@paikea, dnetc.q@paikea, beagle.q@paikea, all.q@exec1, dnetc.q@exec1, all.q@exec2, dnetc.q@exec2. Queue instances inherit the properties of the queue they belong to but also override them with the specifics of the node they belong to. For example, the default queue `all.q` can have the attribute slots set to 1, which offers one job slot for every node in the queue. The queue instance all.q@paikea overrides this value with the correct number of CPUs available in the host (4 in this case). When users submit jobs to a certain queue, the scheduling mechanism of SGE starts sorting all queue instances in the queue until a match is found for executing the job. If job is submitted without specification of a queue, then it is matched to the first available slot in the queue that the user has rights to submit to. The examples described in this section make use only of the default queues of both SGE and Torque.

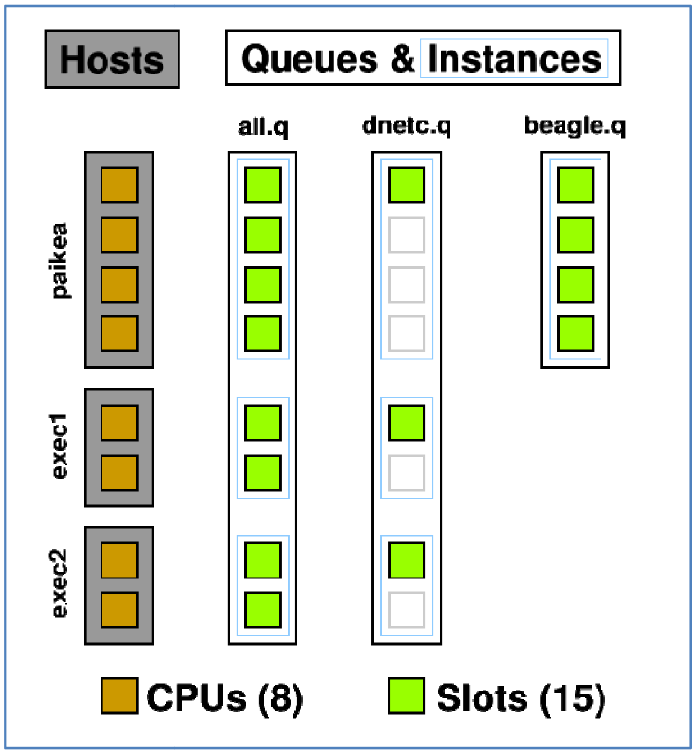


Fig.11 An example of different SGE queues

4.4.1 Submitting a Serial Job

Jobs are submitted to SGE and Torque using the command `qsub`. This complicated command has many options that, for instance, define the name of a job, its input and output files, execution environment, resource requests, etc. Some of them will be discussed in detail in the following examples. The argument `qsub` accepts is a shell-script file that includes a list of all necessary options for executing a job. It can take an executable as an argument but the option "-b y" needs to be explicitly specified. Similarly to the submit-description file of Condor, the shell script contains the name of the executable together with possible environment adjustments. Each line in the script that starts with special sequence of characters ("#$" for SGE and "#PBS" for Torque) is meant for `qsub` to interpret. These lines describe all options of the command that tune the job submission. Below is an example of a script file that submits to SGE the sequential implementation of our testing application NumIntegrationTest.exe (see the section on MPI).

```
#!/bin/sh
#$ -S /bin/sh
#$ -cwd
#$ -V
#$ -N NumInt
$HOME/NumIntegrationTest.exe
```

This script file contains only options for the `qsub` command together with the name of the executable that is going to initiate a new job. The SGE ignores the first line of the script and it uses the queue's default shell `csh` for executing jobs. That is why users can specify the preferred shell using the "-S" option. In the example it defines usage of the `sh` shell. Following the options list, the "`-cwd`" defines that the job is executed in the current working directory it was submitted from. If this option is not specified the job will be executed in the home directory of the submitting user. Specifying a separate working directory is important if several runs of the same job are instantiated and all of them require different input files. As with Condor, SGE and Torque automatically redirects input and output (stdin, stdout, stderr) to files. The names of these files can be specified additionally by using the options "`-i`" for an input file, "`-o`" for output file, and "`-e`" for a file that contains error messages. If these options are not defined, like in the example above, input is redirected to */dev/null* and output creates two files. In our example they will be placed in the submitting directory because "-cwd" is specified. The names of the output files are formed like <job_name>.o<jobID> and <job_name>.e<jobID> (e.g. NumInt.o3, NumInt.e.3). Additionally SGE provides another option "`-j y`" for manipulation of the stdout and stderr. It causes the output of the two streams to be merged into a single file for easier manipulation and readability. The next option "`-V`" passes all environment variables of the submitting shell to the executing shell ("-v", lower case, passes only a certain environment variable). Finally, "`-N`" defines the name of the job.

SGE provides the possibility to define job requirements upon job submission. The requirements about the platform of the execution host can determine the speed at which an application is executed in a heterogeneous environment. Users have a notion of how much processing power their application needs or how much memory it consumes. That is why it is recommended that hardware requirements are specified when a job is submitted as the built-in scheduler has no other way of finding which the best match for a job is. An extra option "-l" to the `qsub` command defines a subset of machines which can execute a particular job. It can be used to specify resource requirements for the remote host like free physical memory (mem_free), CPU architecture (arch), or a hostname. The example below shows two requests – one for nodes that have 1Gb of free physical memory and CPUs of x86 architecture and another specifying a concrete host. Users can check the available options for resource requirements using the `qhost` command. Issued no arguments, it shows the hardware configuration of all nodes in the cluster (Fig. 12). It lists all nodes together with their CPU architecture, number of CPUs, memory capacity, current load, and used memory. Issued with the option "-l attr=val" it generates a filleted list of all nodes that match the requirement.

```
#$ -l mem_free=1G, arch= lx26_x86
#$ -l hostname=compute-0-2
```

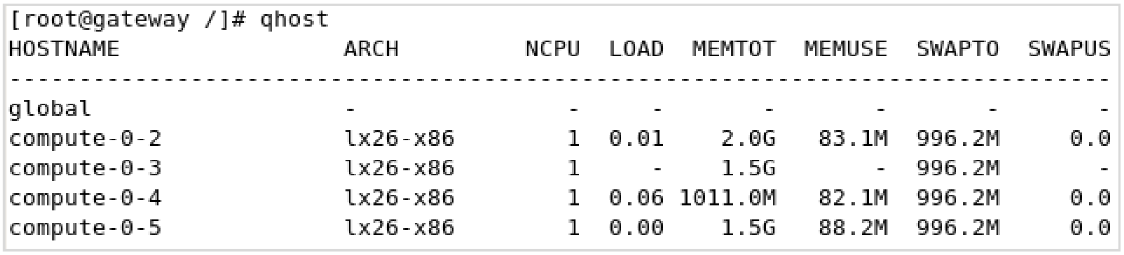

```
[root@gateway /]# qhost
HOSTNAME                ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
-------------------------------------------------------------------------------
global                  -               -     -       -       -       -       -
compute-0-2             lx26-x86        1  0.01    2.0G   83.1M  996.2M     0.0
compute-0-3             lx26-x86        1     -    1.5G       -  996.2M       -
compute-0-4             lx26-x86        1  0.06 1011.0M   82.1M  996.2M     0.0
compute-0-5             lx26-x86        1  0.00    1.5G   88.2M  996.2M     0.0
```

Fig.12 qhost output

After the script file is created the test job NumInt can be submitted using `qsub`.

```
# qsub SubmitScript.sh
```

SGE allows users to manipulate the system using an X-Windowing environment. It is an aggregation of the command-line tools, which make up the SGE, into one graphical environment. The interface can be started by issuing the command `qmon`. A small window appears on the screen at start-up (Fig.13) that contains a number of icon buttons, each of which initiates a dialogue window for fulfilling one of the major administrative and user tasks on the environment. These dialogues are Job Control, Queue Control, Job Submit, Complex Config, Host Config, Cluster Config, Scheduler Config, Calendar Config, User Config, PE Config, Checkpoint Config, Ticket Conf, Project Conf, Browser, and Exit. The first button (top, left) starts a Job Control dialog window which has three tabs each containing a table with a list of pending jobs, running jobs, and finished jobs. Here users can suspend and resume jobs, reschedule, delete them or change their priority. The second button starts a job submission dialog window (Fig.14). One can easily notice that all fields match the options that can be specified in a subscription file. Executables, still, have to be specified inside a script file. The name of the job can be specified together with a list of arguments, working directory, shell to be used and name of I/O files.

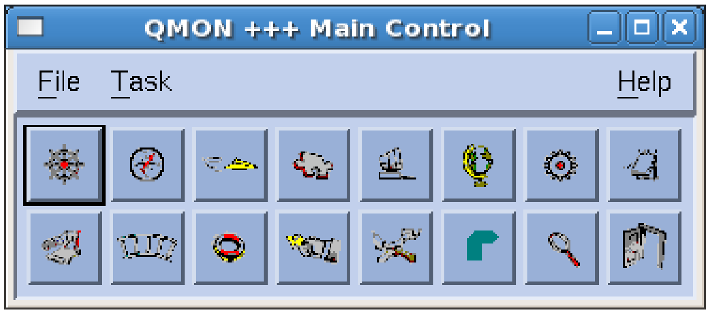
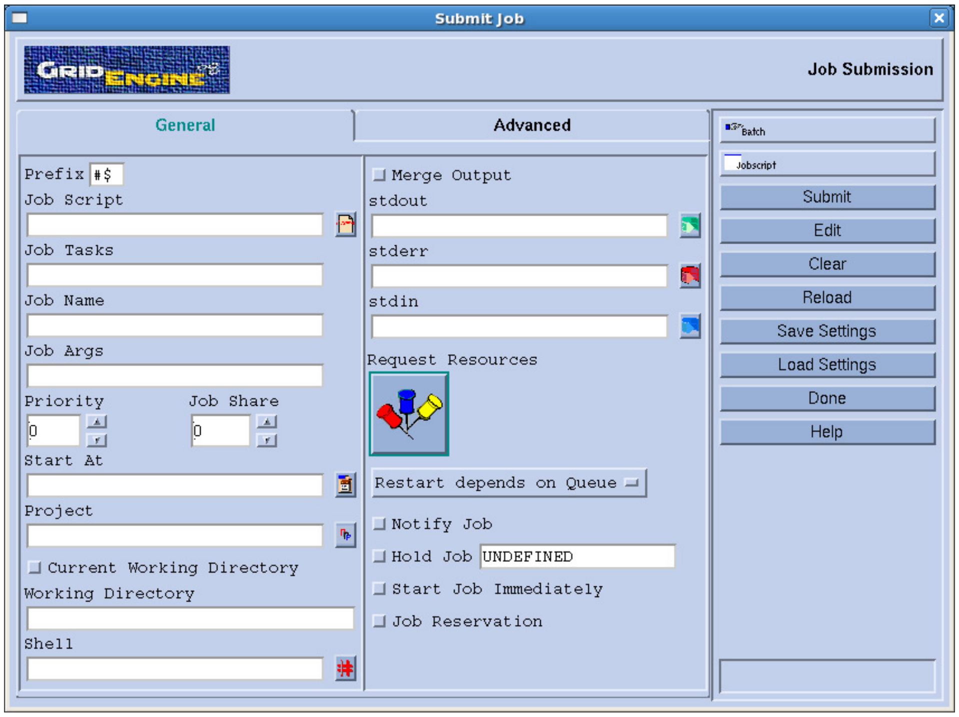


Fig. 13 SGE's graphical interface Main Control



```
Prefix #$
Priority 0          Job Share 0
Restart depends on Queue
Hold Job UNDEFINED
```

Fig. 14 Graphical interface for job submission

Once a job is submitted to the system, it is placed into a queue. In our testing environment the SGE uses its default configuration. Hence, no additional queues are defined in the system and it uses the global one (`all.q`). Users can check the status of the submitted jobs using the command `qstat`. Torque uses the same command. Fig.15 shows a screenshot taken from our testing environment after submitting the job in the above example. Without specifying any options the command returns a list of all jobs in the default queue. Jobs are described by their ID, priority, name, state, submission time, allocated queue instance and number of occupied slots. From the output one can determine what state the job is in (waiting, running, suspended, error). If it is in state Rr (running), like in Fig.4, one can also determine on which node it has been assigned to for execution (in this case node compute-0-5). If the job does not appear in the queue statistics immediately after submission one must to check the contents of the automatically generated error-report file.

```
[sgeorgiev@gateway ~]$ qstat
job-ID  prior   name       user         state submit/start at     queue
                 slots ja-task-ID
-----------------------------------------------------------------------------
---------------------------------
     63 0.55500 NumInt     sgeorgiev    Rr    06/30/2009 12:22:15 all.q@comput
0-5.local               1
```

Fig.15 Output of qstat for a serial job

Furthermore, the `qstat` command proves to be quite useful for monitoring the queues in the system and the jobs submitted to them.

- `qstat -j <jobname_or_ID>` prints various information about a job. If the job is in the error state, the error reason is displayed andif it is in the running state it shows information on resource utilization.

- `qstat -f` specifies a “full” format display of information. Summary information of all available queues in the system is displayed.
- `qstat -u <Username>` displays information about all jobs and queues the specified user has access to. It is useful for troubleshooting as it lists the contents of some environment variables, specific for SGE. They contain paths to binaries and libraries.

4.4.2 Modifying a Job

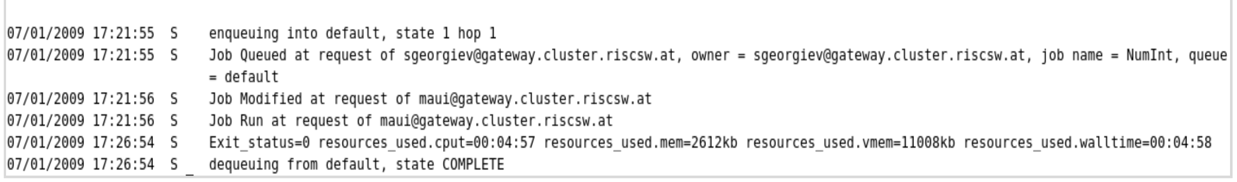
Jobs submitted to a cluster environment might be such that they require lots of time to complete or they do not schedule the way they are expected to. In cases like this, cluster administrators might have to suspend a slow job or even stop it permanently. A job can be canceled using the command `qdel (qdel <jobname_or_ID> | -u <username>).` If the job is already in the running the system will refuse to stop it with a warning message. To force stopping the job one can use the option “`-f`”. If a user is specified as an argument, then all jobs submitted by him will be cancelled. Torque uses `qdel` to cancel jobs, too.

Jobs can also be put into a suspended mode. In this state they free the occupied resources and return to the queue waiting to be resumed. Modifying a job is achieved using the `qmod` command. It can be used to suspend/resume jobs, to disable/enable them, reschedule them, or clear their error states. When a job restarts after being in suspended mode or is rescheduled its status is already “`Rr`”.

- `qmod -sj <jobname_or_ID>` suspends a job.
- `qmod -usj <jobname_or_ID>` resumes a job.
- `qmod -rj <jobname_or_ID>` reschedules a job.
- `qmod -f -rj <jobname_or_ID>` force reschedules a job.

4.4.3 Troubleshooting

After a job completes its execution, all output is contained in the automatically-generated output files (NumInt.e.63 and NumInt.o.63). If an error occurs at some point during execution it is listed in the error output file. That is why this should be the first thing to be checked when a job finishes executing. Additionally, users can see statistics about the execution process by issuing from the command-line `qacct -j <jobID>.` The command provides an access to statistical data regarding all executed jobs. It produces summary information for wall-clock time, CPU-time, and system time together with exit status, and node that handled execution. Its options allow users to search the data for entries that match a specific time period, user, job, hostname, etc. When a job is rescheduled to use another machine or it simply occupies several machines during execution time, summary is printed for each machine involved. In the case of Torque, the command `tracejob <jobID|name>` provides summary information of the execution process. Example output is shown in Fig. 16. The command pints a job’s exit status, execution time on the CPU, wall time, used physical memory.



```
Job: 0.gateway.cluster.riscsw.at

07/01/2009 17:21:55  S    enqueuing into default, state 1 hop 1
07/01/2009 17:21:55  S    Job Queued at request of sgeorgiev@gateway.cluster.riscsw.at, owner = sgeorgiev@gateway.cluster.riscsw.at, job name = NumInt, queue
                          = default
07/01/2009 17:21:56  S    Job Modified at request of maui@gateway.cluster.riscsw.at
07/01/2009 17:21:56  S    Job Run at request of maui@gateway.cluster.riscsw.at
07/01/2009 17:26:54  S    Exit_status=0 resources_used.cput=00:04:57 resources_used.mem=2612kb resources_used.vmem=11008kb resources_used.walltime=00:04:58
07/01/2009 17:26:54  S _  dequeuing from default, state COMPLETE
```

Fig.16 Output of tracejob

Finally, if a job does not run at all or generates errors, SGE provides a way of answering why this might be happening. The command `qstat -explain c -j <jobID>` provides a reasonable message. An example is “queue instance all.q@compute-0-4.local dropped because it is temporary unavailable”, which indicates that compute-0-4 might be offline.

4.4.4 Submitting an MPI Job

The SGE introduces a way of submitting parallel jobs to the cluster as well. What is more, OpenMPI also implements support for the grid engine starting from version 1.2.0. This means that when an `mpirun` command is executed in an SGE job, it will automatically use the SGE mechanisms to launch and kill processes [OpenMPI, 2009]. In addition, it uses the SGE configuration of the environment and its knowledge about the nodes in it. An `mpirun` command no longer needs specifying a machine file containig a list of nodes to which workload must be distributed. SGE allocation is done at a higher level and it generates its own machine-file which is usually placed in *$TMPDIR/machines*.

SGE provides two different ways of submitting parallel jobs to the cluster. The first way is more suitable for executing small tasks on all nodes in the cluster. It uses the SGE command `qrsh`, which operates in a similar way to the normal `rsh` command with the difference that no hostname is passed as an argument. Instead, an executable or a shell script is run on every node of the cluster at the same time. The results are directed back to the submitter’s terminal. `qrsh` operates differently from `qsub` as it does not place the job into the queue. If the job cannot be run immediately, it is dropped. Additionally, the input and output streams are not redirected automatically to files but users have to specify this explicitly using the shell redirect operators. Nevertheless, the following example runs our MPI test application on the cluster.

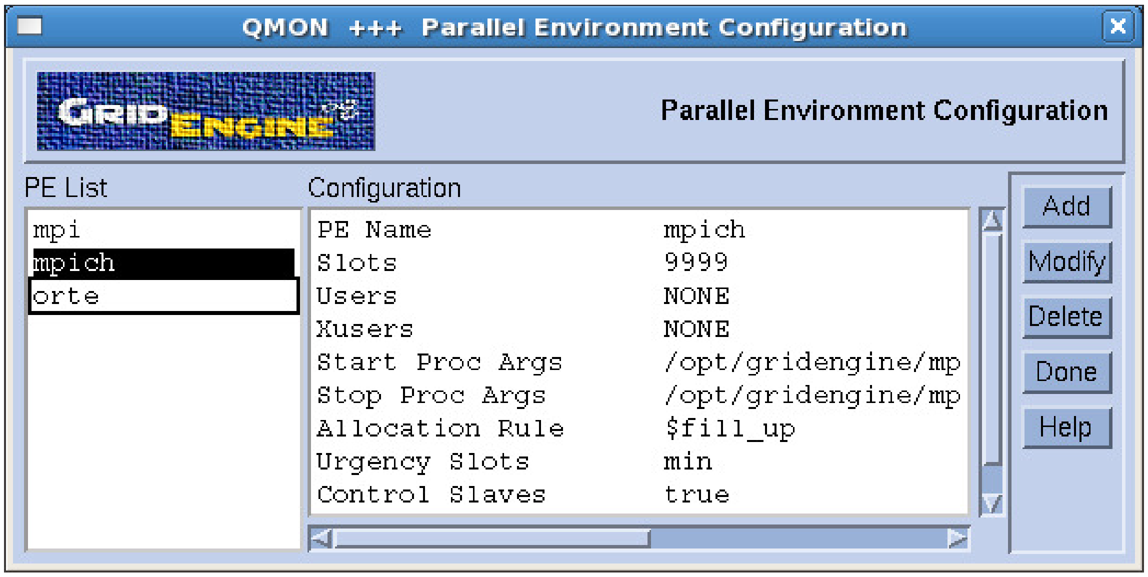
```
# qrsh -V -pe orte 3 mpirun -np 3  \
$HOME/MpiNumIntegrationTest.exe
```

The second option for running an MPI parallel application is using the command `qsub` and a shell script. The script used to run our test application is the following

```
#!/bin/sh
#$ -S /bin/sh
#$ -cwd
#$ -j y
#$ -V
#$ -N MpiTest
#$ -pe orte 3
MPI_DIR = /opt/openmpi/bin

$MPI_DIR/mpirun   -np   $NSLOTS   --mca   pls_gridengine_verbose  1
$HOME/MpiNumIntegrationTest.exe
```

where the difference to the example submitting a serial job, is in the options “-j y” and “-pe orte 3”. The first one causes the output of the error stream to be merged with the output of the standard output stream into one file. The second option holds, actually, the main difference to the first example. This option specifies a parallel programming environment that handles the execution of an MPI application with a number of processes in it. The current configuration has three predefined parallel environments (PEs): *mpi*, *mpich* and *orte*. A list of the available PEs can be displayed using the command “`qconf -spl`”. What is more, details about what parameters each parallel environment contains can be displayed by issuing "`qconf -sp <orte|mpi|mpich>`" (see Fig. 17). All of them have defined 9999 slots and differ in the configuration of the parameters `start_proc_args` and `stop_proc_args`. For *orte* they both contain */bin/true/* and for *mpi* they contain */opt/gridengine/mpi/startmpi.sh $pe_hostfile* and *~/stopmpi.sh*. One can add a new parallel environment using the command `qconf -ap <pe-name>`. However, for doing this and modifying the SGE environment in general one must be a manager of the environment. Upon initial configuration ROCKS sets the root user to be the manager. SGE usually does not grant administrator privileges over the system even to the root user – a different user is normally defined to be an SGE manager. Nevertheless, in the ROCKS environment the command works by opening a "Vi" text editor which is intended to create a help environment for configuring a new PE. All options are already defined and filled in with default values. Users have to save the file when they are done with the configuration. Finally, a newly created PE can be removed with the command `qconf -dp <pe-name>` and modified with the `qconf -mp <pe-name>.`


QMON +++ Parallel Environment Configuration
GRID ENGINE
Parallel Environment Configuration
PE List
Configuration
Add
Modify
Delete
Done
Help

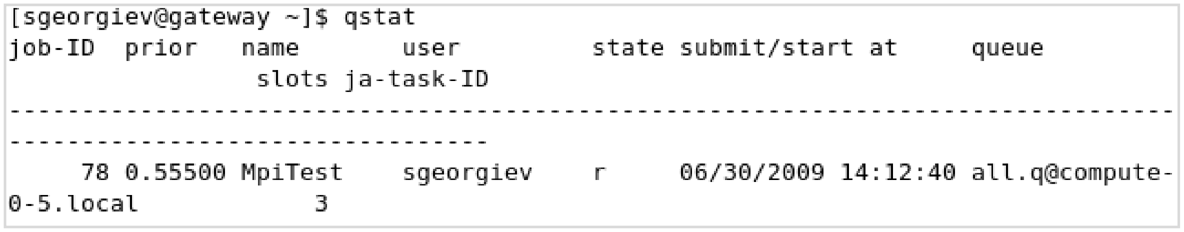
```
mpi
mpich
orte
```

```
PE Name             mpich
Slots               9999
Users               NONE
Xusers              NONE
Start Proc Args     /opt/gridengine/mp
Stop Proc Args      /opt/gridengine/mp
Allocation Rule     $fill_up
Urgency Slots       min
Control Slaves      true
```

Fig. 17 Parallel environment configuration using the SGE GUI

After the script file is created the test job MpiTest can be submitted using `qsub`. The job is immediately put into the queue and then it is executed occupying 3 slots (all available compute nodes). Fig. 18 shows the contents of the queue `all.q`.

```
# qsub SubmitScript.sh
```



```
[sgeorgiev@gateway ~]$ qstat
job-ID  prior   name       user         state submit/start at     queue
                 slots ja-task-ID
--------------------------------------------------------------------------------
---------------------------------
     78 0.55500 MpiTest    sgeorgiev    r     06/30/2009 14:12:40 all.q@compute-
0-5.local            3
```

Fig. 18 Output of qstat for a parallel job

Our experience showed that using the SGE's mechanisms for submitting parallel jobs puts certain constraints to the execution process compared to using the conventional OpenMPI interface. The grid engine operates in a way that on each execution host an execution daemon is installed, that takes care of handling the particularities around a local job execution process. It is also responsible for determining the capacity of the machine to run jobs by defining available slots for it. The SGE does not limit the number of job slots that the daemon can offer, but normally it associates a job slot to each processor or core available on the machine. This way, the resource pool consists of available slots that can be matched to a job. When an MPI job is submitted to the system the SGE tries to create an optimal execution environment by initiating a number of processes that matches the currently available slots in the system. If there are machines with quad core (8 cores) processors, the SGE will create 8 different processes on each of those machines. In our case, however, all machines in the cluster consist of single core processors. That is why, when an MPI job is submitted to the cluster it can use only a number of processes that matches the number of processors (3 in our case). When a larger number of processes are specified the SGE replies with an error message that slots are not available and refuses to run the job. The job remains in the queue in an idle state. Thus, in an environment of four machines, like the one used for testing, the SGE allows only 3 of them to be used effectively. The frontend is recognized as manager-submit node and does not contribute to the number of available slots with its processor(s). In contrast, the conventional OpenMPI interface has no such restriction. It allows users to create any number of processes as long as the hardware is able to support them. What is more, the machines used for executions are defined only by an installation the OpenMPI and the frontend can be also included. For instance, the same application can be submitted for execution using 200 processes through the OpenMPI interface. This, of course, does not lead to any improvement of performance as the hardware capabilities of the testing cluster are rather limited. When using it, however, one has to keep track manually of the available processors in the resource pool and only then to specify the optimal number of processes to be created.

4.5 Summary

ROCKS 5.1 provides a set of tools that help users to manage the hardware and software resources of a cluster and properly distribute the workload in a way that best matches them. These tools are reviewed in the form of examples that aim to describe what can be achieved with them and how they can be used. A basic overview of the main features and the ideas behind their functionality show that the tools Ganglia, OpenMPI, Condor, Sun Grid Engine, Torque/Maui are more than useful in a parallel production environment.

The chapter starts with a section dedicated on the real-time monitoring tool Ganglia. It is a simple but powerful tool for monitoring the resources of high-performance clusters or federation of clusters (clusters of clusters). The tool produces a vast amount of statistical data regarding resource utilization at any point in time. What is more, it saves this information in a database so that analysis can be performed on the performance of the cluster upon execution of a certain application. Cluster administrators can benefit greatly from using Ganglia as it provides a general overview of the whole environment in the form of easy-to-read images. This makes Ganglia usable for both troubleshooting and monitoring the available resources.

Furthermore, the chapter continues with presenting an example of a parallel application that aims to show how useful MPI still is for achieving parallelism in applications. MPI provides a strong programming interface for low-level control of processes spawned on different machines but working all together on a single application. It also provides an irreplaceable mechanism for implementing different parallel algorithms and techniques in order to increase the performance of an application. Utilizing it, however, requires users to manually analyze the available hardware resources and the interconnection method in order to create an implementation that best suits them.

The tools described in the following sections aim at solving this problem automatically. They implement mechanisms of keeping track of the available resources and matching the cluster workload to them. The main difference is that their design focuses mostly on finding the most appropriate machine to execute a single application (a job). While they also allow users to submit parallel applications (e.g. MPI binaries), these are executed under some restrictions compared to the conventional MPI mechanism.

The first tool, Condor, is a batch system that creates an environment for assigning jobs to available computing resources. In contrast to others, its main purpose is not only to find the best machine for executing a job but also to ensure that long-time-running jobs complete eventually. For this, Condor relies on a mechanism of job migration (using checkpoints) that allows a job to continue executing on a different machine in case the initial one fails. Thus, it is designed to serve large-scale networks of machines of distributed ownership like Networks of Workstations or the Grid. In this way the created ROCKS cluster can be easily included into a larger scale distributed environment.

Two other tools that work similarly to Condor are the resource managers Sun Grid Engine (SGE) and Torque/Maui. Compared to Condor both tools are more cluster oriented. SGE implements a wide variety of policies to facilitate matching of jobs to machines. What is more, it provides a convenient GUI for manipulating the system. Similarly to Condor it also supports the mechanism of check-pointing through its API called DRMAA. It also supports submission of parallel jobs to the system by providing a wide variety of parallel programming environments. However, jobs are executed with some differences to the conventional MPI mechanism because of the specifics around handling the resources in the SGE system. Nevertheless, users are relieved from manually analyzing the available resources as the SGE automatically occupies those machines that respond to the user-specified requirements of the application. SGE includes a scheduler in its implementation that decides which jobs are going to be run next. In comparison, Torque uses an external scheduling module – Maui. Still, SGE proves to be more than useful in a ROCKS cluster environment.

Chapter 5
Test Application

This chapter aims to describe the behavior of the achieved testing environments when confronted to the challenges of running a real-life production application. Developed by RISC Software GmbH, this application solves compute-intensive tasks by utilizing optimized algorithms, complex data structures and the message passing interface. It is designed and implemented to run on a distributed environment in order to achieve faster execution time. The two testing clusters built with ROCKS and CAOS-NSA, were used to run the application. Results, in the form of wall-clock timings, network load and memory load, are analyzed and compared.

Section 5.1 provides an overview of the application and the techniques it incorporates. Section 5.2 shows the results obtained during testing. Here, a comparison is made between the two clusters used. Section 5.3 summarizes the gained experience and provides conclusions.

5.1 Application Description

A real-life production application is used for testing the functionality of the ROCKS and CAOS-NSA clusters. This application is a product of the joint effort of the parallel-programming team at RISC Software GmbH and the University of Applied Science, both located in Hagenberg, Austria. It is developed by Andreas Scheibenpflug under the supervision of Michael Krieger.

The application itself is designed to serve the needs for other more complicated pieces of software that deal with problems in the area of route optimization. It provides a core component for industrial logistics applications which often need to solve the problem of how transportation and delivery of goods can be organized so that travelling expenses are minimized. This problem is known as the Travelling Salesman Problem (TSP). While there is no known algorithm that solves the TSP in the general case, many optimizations and limitations can lead to finding a satisfactory solution. The application used for testing provides a basic module for computing the TSP – it calculates the shortest paths between numerous of sites. It does not deal with computing the TSP itself but it creates a graph data structure based on map data that contains customer locations and the shortest paths between them. All values for a predefined set of customer sites are saved in a matrix (often referred to as a distance matrix) which is then used for creating the graph.

Unlike TSP, finding the shortest paths between vertices in a graph is a problem which there are numerous solutions to. However, it proves to be a compute intensive task when the problem area becomes too big. In our case, the test application works on a data structure that represents the street network of Austria. This data structure is a graph that consists of about 900’000 nodes and 1.1 million edges. It is generated based on real map data that contains information about streets (type, direction, length), intersections, addresses, etc. This data is simplified in order to remove all unnecessary details resulting in a graph representation where all intersections are introduced as nodes and all streets between these intersections as edges. The optimized data structure holds the information about the complete street network of Austria in a file that is of size 110 MB. During execution time, this file is loaded in the physical memory and, thus, makes the application free from interacting with the hard-drive. Another optimization is that the test application only computes the shortest paths between a small subset of nodes which represent the client sites. Furthermore, the core component of the application is the algorithm which, actually, is responsible for how optimal the distance matrix computation is. Dijkstra's algorithm was chosen for several reasons. First, it shows to be faster compared to others when computing the single-source shortest path problem even when executed on a conventional desktop computer. In fact, this is why the application does not parallelize the algorithm itself but runs multiple instances of it on different machines that work on different sets of input data. More importantly, the algorithm relies of the fact that the shortest path to a node in a graph structure consists of all shortest paths to all preceding nodes in that path. This means that shortest path between two nodes can be computed independently from the one between other two nodes and then results can be combined into a single path that is also the shortest. The parallel implementation benefits from this by being able to dividing the workload. Hence, when utilizing a parallel environment, input data can be easily divided between computing nodes so that computations are carried out independently and results are combined at the end. The application itself is implemented using MPI. It incorporates a client/server approach, where the master process is responsible for providing work to the slaves and then collecting the results which form the distance matrix. Communication between processes is kept to minimum but synchronization still takes place between the server and the clients.

5.2 Testing and Comparison

This section describes the results obtained from running the distance-matrix calculation on our testing cluster environments. Upon development, the application was tested using different technologies and approaches in order to compare them and determine which one results in achieving lowest execution time. In addition to MPI, an alternative implementation was also created using OpenMP. What is more, the application was tested also using the middleware BOINC and several desktop computers. However, previous to testing it on the two cluster environments it had not been executed on another HPC parallel computer. Hence, the obtained results only determine the behavior of the ROCKS cluster in comparison to the CAOS-NSA one.

There exists an alternative implementation of distance-matrix calculation using OpenMP. OpenMP is a programming approach that is used for paralyzing computations in shared-memory environments. Instead of creating different processes, like MPI, it creates multiple threads that work concurrently on a single machine. This way, the Dijkstra's algorithm could be tested using one and two threads on a multi-core desktop computer. Using an input of 100 nodes, the OpenMP implementation executes for an average of **86.6 sec** when utilizing a **single thread** on a dial core machine (see Table 1, comp. 4). Five different runs were performed for obtaining this value. Another five runs reveal that the application executes for an average of **93.67 sec**. when **2 threads** are used on the same dual-core machine. Comparison of these values clearly shows that the implemented algorithm is not parallelized effectively. Not only, it achieves no speed up when using more processors, but it shows performance degradation.

As described in Chapter 3, two cluster environments were built using the cluster deployment tools ROCKS and CAOS-NSA. Table 1 (see Chapter 3) provides a list of the machines included in each of them together with their hardware configuration. Both clusters consist of four machines, three of which have single core processors and one (the head node) has a dual core processor. Hence, there are 5 processors in total available on each cluster.

The application is tested first on the ROCKS cluster using different sets of input data against different number of processors. Testing data includes different files containing IDs of nodes from the graph representation of street network in Austria. Four different test files were used containing 10, 35, 50, and 100 IDs in them. The application computes the shortest path between each pair of nodes using the street graph. However, for the purpose of presenting the behavior of the parallel environment in terms of execution time, this section focuses mainly on the most compute-intensive example of calculating the shortest paths between 100 nodes.

Using input data of 100 nodes the MPI implementation of the distance-matrix calculation is executed on the ROCKS cluster. Table 7 shows a detailed overview of the execution behavior by describing the wall-clock runtime of each process together with the total execution time for five separate runs. The average total execution time on the five processors is **70.13 sec**. In order to pass a basic notion of how the speed-up improves when multiple processors are involved a comparison is made to the execution time of the OpenMP implementation. The current design of the MPI application does not allow it to be executed on a single process because it implements a server/client architecture and at least two processes need to be started. Thus, there is no other alternative for comparison to a sequential implementation that the OpenMP one. Of course, the obtained results from the two techniques cannot be compared fairly as the OpenMP implementation operates completely differently on both the hardware and software level. One has to consider that threads are faster than processes and they use shared memory to communicate. Nevertheless, a comparison reveals that *utilizing 5 processors instead of 2 in the computation process results in achieving a relative speed-up of 1,34*. This result is far from being satisfactory as little absolute speed-up is achieved but it still shows that the parallel environment introduces better speed-up than using a single dual-core machine.

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
Server Process (gateway)	10.66	13.13	11.27	13.31	13.29
compute-0-2	68.82	68.30	68.41	69.33	68.68
compute-0-5	63.58	62.57	63.20	63.19	63.33
compute-0-4	65.77	69.44	69.59	70.27	70.94
gateway	54.04	51.50	52.18	52.03	52.55
Total execution time	69.1883	69.8349	70.9970	70.6682	71.3777

Table 7 Execution timings (in seconds) for each process executing the distance-matrix calculation on the ROCKS cluster

When the distance-matrix calculation is executed numerous times using different number of processes (processors) it reveals an important characteristic of every parallel application – increasing the number of processors increases the execution speed. This shows how scalable parallel implementations are in means of achieving speed-up. Fig 19 shows the execution wall-clock timings measured on the ROCKS cluster with an input of 100 nodes. One can see how increasing the number of processors improves the execution timing drastically but only up to the point of reaching the optimal value of 70.13 sec. However, the chart also shows how the environment is limited by its hardware profile. The ROCKS cluster contains in total 5 processors (See Table 1, Comp1-4). Thus, speed-up is only achievable up to the point that the number of processes meets the number of available processors. Starting more processes introduces significant degradation in performance. Similar results can be observed on the CAOS-NSA cluster.

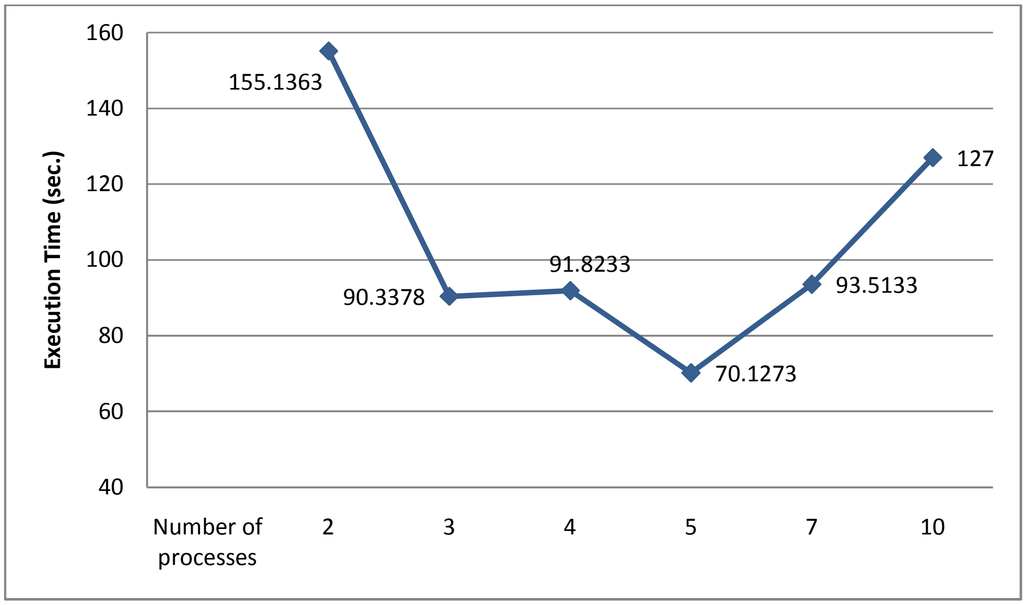


Fig. 19 Execution timings (in seconds)
on the ROCKS cluster for different number of processes

Introducing more processes to a system that cannot support them with its available hardware results in loss of performance for a parallel application. The reason is that more processes require additional memory and CPU clock cycles. If none are available, processes have to wait for others to finish or what is even worse they start executing, by stealing clock cycles from an already running process causing synchronization and context-change overhead. What is more, in the case of MPI applications, more processes results in more communication demands and synchronization for networking resources. Table 8 shows how network traffic increases with the number of processes started in the system. These results are obtained using the Ganglia monitoring tool. They clearly show that network traffic increases with the number of processes as, in the case of our application, processes receive work from the master process and have to synchronize with it. More processes result in more data traffic which indicates that a production HPC environment which contains even several hundred processors requires fast communication links. Otherwise network throughput will surely not be able to cope with the speed of the processors and network latency will result in processes being idle waiting for a message.

Number of processes	Data Transfer
2	4-6 MB
3	4-6 MB
4	7-8 MB
5	11-12 MB
7	16 MB
10	22-23 MB

Table 8 Network traffic for different number of processes measured on the ROCKS cluster

On the other hand, increasing the number of processes in a parallel system does not always result in achieving speed-up even in the case when the underlying hardware infrastructure supports multiple processors. Our experience shows that execution time of an MPI application is also tightly dependent on the problem size. For large input data sets, which obviously require lots of computational time, more processors compute faster. But, when input is small in size, then the application suffers from inter-process communication rather than it benefits from it. The larger the number of processes is, the bigger the communication overhead becomes. In cases like this, network latency takes over parallelism resulting in performance difference. Table 9 shows a comparison between execution wall-clock timings for 4 and 5 processes measured on the ROCKS cluster with input data 10. Table 10 shows a similar result comparing execution wall-clock timings for 3 and 4 processes measured on the CAOS-NSA cluster. In both cases, adding one more process introduces a delay of roughly 2 seconds.

Number of processes	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**AVG**
4	32.7486	32.8408	34.1129	32.9065	32.7829	**33.1781**
5	34.4617	35.1585	35.4698	34.8707	37.5924	**35.5106**

Table 9 Execution timings (in seconds) for 4 and 5 processes on the ROCKS cluster for input of 10 nodes

Number of processes	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**AVG**
3	34.1212	34.3120	34.0535	34.4155	34.3218	**34.2448**
4	36.3170	36.4842	36.0285	36.0721	36.0940	**36.1991**

Table 10 Execution timings (in seconds) for 4 and 5 processes on the CAOS-NSA cluster for input of 10 nodes

In this last example one can notice that for the CAOS-NSA cluster (see Table 1, comp.5-8) the comparison is made between 3 and 4 processes even though the environment supports 5. The reason is that the CAOS-NSA cluster revealed a major downside to utilizing heterogeneous environments. It showed that the whole system is as slow as its slowest part. In contrast to the ROCKS cluster, this one contains a node that has rather old architecture (PIII 866Mhz). When the distance-matrix calculation is executed this node shows significant slow-down making all the other nodes wait after they finish. This proves that a heterogeneous cluster environment can benefit in achieving faster computations only when it utilizes similar processors. Once a node introduces a processor that is significantly different from the others, it will cause synchronization problems. Even in the case the processor is much faster than the other ones it will not perform at its best because it will either remain idle till the others are done or it will be stuck in waiting states while trying to synchronize. These difficulties can be overcome only at programming level. Applications have to be designed to implement a different kind of parallelism that takes into account the heterogeneous nature of the execution environment. This way slower nodes can be given less workload and thus finish in time with the others.

In the following examples the slow node was excluded from the CAOS-NSA cluster. The tables below aim to show the behavior of both the ROCKS and CAOS-NSA clusters when executing the distance-matrix calculation with different input data and different number of processors. All described results are obtained as an average of five different runs. Table 11 describes average wall-clock timings using different input data on the ROCKS cluster. The example contains measurements of runtime using 3 and 4 processors so that comparison can be made to the CAOS-NSA environment. Table 12 shows the results for the CAOS-NSA cluster. It does not contain a third line with timings for 5 processes because the slowest node was eliminated previous to performing the tests.

Number of Processors	**Input Data 10**	**Input Data 50**	**Input Data 100**
3	35.1203	58.9195	90.3378
4	33.1781	53.74345	91.8233
5	35.5106	47.6663	70.1273

Table 11 Execution timings in seconds for different number of processors and different input measured on the ROCKS cluster

Number of Processors	**Input Data 10**	**Input Data 50**	**Input Data 100**
3	34.2448	58.1279	87.4807
4	36.1991	52.7227	75.2497

Table 12 Execution timings in seconds for different number of processors and different input measured on the CAOS-NSA cluster

While the average execution timings for using 3 processes show that the CAOS-NSA cluster performs, more or less, equally to the ROCKS one, the timings for 4 processes reveal a significant difference especially when input data reaches the maximum value of 100. Then the CAOS-NSA cluster performs rather faster than the ROCKS one beating its time with roughly 15 seconds. Nevertheless, this result cannot be explained with the software advantages of one environment over the other. Both utilize OpenMPI and in this case it operates independently from the underlying clustering software. Thus, at this point no reasonable explanation can be provided regarding the utilized tools for parallelism. The matter requires further analysis and investigation. One explanation can be that the CAOS-NSA set of machines performs better under these circumstances as the utilized hardware happens to be more suitable for solving this particular problem.

5.3 Summary

This chapter analyzes the behavior of the ROCKS and CAOS-NSA cluster environments when executing a real-life application. The application, developed by RISC Software GmbH, was chosen to be tested on the clusters as it was designed to operate in a distributed parallel environment. Both, the application and the performance of the cluster were assessed with the described test runs. This way the development team received valuable data about execution timings. The application itself computes the shortest paths between a set of points taken from the road-map of Austria. It is intended to be integrated into industrial logistics applications in order to help route optimization in the area of transportation and delivery of goods. The application creates a distance-matrix carrying out computations in parallel.

The application was tested against different sets of input data and different number of processes on both the ROCKS and CAOS-NSA clusters. Results do not reveal any advantage or disadvantage of using either software environment but prove that the used hardware plays significant role in how fast computations are performed. What is more, results undoubtedly show how speed-up of a parallel application increases with increasing the number of processors involved in the computations. Again, hardware puts certain limits and determines a maximum value for this speed-up. Results describe how communication and synchronization overhead take over after a certain point resulting in loss of performance. On the other hand, the test runs on different input data confirm that speed-up is also tightly dependent on the size of the problem area. When size is too small, the performance decreases as it suffers from all the process creation, handling, and communication. Finally, monitoring execution behavior upon different test cases proves that one can determine which machines in the heterogeneous environment are more suitable for performing calculations and adjust the environment every time a subset of computers is needed. In addition, certain machines can show significant slow-down when confronted to an application and thus degrade the performance of the whole system.

Chapter 6
Conclusions and Future Work

This thesis describes thoroughly the process of building a Beowulf-type cluster for high-performance computing. The presented evaluation focuses on techniques and approaches for creating a parallel production environment that uses heterogeneous commodity hardware platforms instead of specialized high-performance ones. What is more, the study reveals the possibilities of reaching maximum performance at the lowest price by using common desktop computers. This is an important result for companies that already use or create software that requires a parallel environment in order to run faster and more effectively.

Creation of a fully-functioning parallel environment for high-performance computing requires installation and configuration of cluster middleware. This is the gluing component that makes a collection of interconnected machines work like a single more powerful computer. That is why, the thesis starts with an overview of different middleware. Further, cluster middleware is systematically analyzed by following a process of creating a real heterogeneous cluster environment. For this high level middleware is assessed in detail by comparison of the cluster deployment tools OSCAR v6.0.2, ROCKSv5.1, CAOS-NSA v1.0. An elaborate description of the installation process of each of the tools aims to reveal both its strong and weak sides. The latest version of OSCAR (and the one used for testing) shows to be quite limited as it is still under development. In general OSCAR is a tool that brings a strong feature set and has proven its effectiveness towards building HPC parallel environments. However, at the time of testing a renovation process was taking place that aimed to improve the installation process and make the tool more flexible and independent from the underlying Linux distribution. Our testing environment showed that a working cluster can be achieved with Debian Etch as a foundation but it does not provide an installation of any tools for clustering. An approach for installing additional tools on the cluster was tested - a local repository was build. However, even with it, installation and integration of tools for parallel computation turned out to require a lot of effort and time. Thus, our experience showed that the current version of OSCAR does not provide the required functionality and, what is more, it introduces lots of difficulties to the installation process. CAOS-NSA, on the other hand, implements a quite fast way of cluster deployment. A cluster of four computers was deployed within 25 minutes. The reason why this is possible is that CAOS-NSA installs stateless images on all nodes in the cluster. These do not reside on the local hard drives but occupy only the physical memory of the machines. Because of this, CAOS-NSA proves to be quite effective for building a parallel cluster environment for testing MPI. However, compared to ROCKS and OSCAR, it incorporates a quite poor list of tools for clustering. Taking this into account together with the fact that the whole cluster is highly dependent on the head-node for remote access to files and services makes CAOS-NSA not that reliable to be used in a production environment. Further, using the cluster deployment tool ROCKS we created a fully-functional cluster with four heterogeneous computers. ROCKS showed to be easy to install on all machines. What is more, it has reached a state of full automation of the installation process where there is very little interaction with the user. This is a major advantage of the tool making it the preferred tool.

Because of this the assessment of the functionality of the built environments focuses basically on the tools provided by ROCKS. It installs a rich variety of tools for resource management, resource monitoring, and submission of sequential and parallel jobs. Series of examples, executed on the cluster, reveal both the strong and weak features of the tools. The batch system Condor is compared to the resource mangers Sun Grid Engine (SGE) and Torque/Maui.

Examples, of submitting serial and parallel jobs to the cluster provide guidelines for users how to utilize the cluster environment for professional purposes. While Condor implements features that make it more suitable to be used in large-scale distributed environments, the Sun Grid Engine together with Torque/Maui show to be more suitable to be used on a cluster. What is more, the SGE implements more advanced feature set than Condor and definitely beats its competitors by providing users with a GUI, which, on top of that, supports the full functionality of the system.

Finally, the functionality of MPI is tested using a simple test application which proves that absolute speed-up can be achieved on the cluster. On the other hand, test runs of a real production application show that achieving absolute speed-up depends mainly on the implementation and the utilized methods for parallelization. Examples reveal that in certain cases communication overhead can take over and result in decreasing the performance of the whole system. What is more, this application proved that a cluster is as fast as its weakest element. This result suggests that heterogeneous environments can demonstrate better performance only when hardware is combined in a proper way. Utilization of processors drastically different in speed leads to loss of performance.

Using heterogeneous environments for high-performance computing is a relatively new approach for achieving better performance with a Beowulf-type cluster. The examples and achievements described in this thesis prove that such an environment can be created and, more importantly, it can be used for implementing better parallel algorithms. The two small testing clusters created using ROCKS and CAOS-NSA can be further improved, so that, RISC Software GmbH can benefit from having a real production parallel environment instead of the testing set-ups. What is more, such a production environment can be built by combining the two test clusters into a single large cluster that consists of 8 machines, 10 processors, and 10 GB RAM. This cluster can already provide a broader range of opportunities for testing and research. As discussed in chapter 5, with larger number of processors incorporated in the cluster, network traffic increases. Hence, testing the capabilities of such an environment requires considering faster means of networking. Gigabit Ethernet or even faster technologies can be further studied and evaluated in order to determine their impact on the performance of the parallel computations. Once the larger cluster proves to be productive, it can be used for conducting further research in the area of high performance computing. Even more, existing parallel algorithms can be tuned to take advantage of the heterogeneity of the machines. Heterogeneous computing is still relatively new area and there is plenty of room for testing and improving. Thus, it requires research and development of new techniques. Job schedulers and resource managers can be further improved to make better use of the heterogeneous resources. Additionally, new scheduling techniques need to be studied so that job distribution is handled automatically depending on current machine load, machine speeds, and current network load. Resource managers can be tuned, for instance, to make processes migrate as soon as faster hardware becomes available in the cluster. This idea, together with many others can be implemented and tested on the larger cluster.

Another opportunity to improve performance in a heterogeneous environment is using graphical processors for performing computations. This idea is relatively new to the field of HPC and has become recently popular with the release of the 8-core Cell Processor on Play Station 3. The popularity of this gaming console makes the computing resources included in it available for testing in a heterogeneous computing environment. While ongoing research proves that this 8-core processor is quite powerful, research is still needed to determine whether it can be used in an environment that consists of conventional commodity CPUs (e.g. typical Beowulf cluster). Interaction between machines has to be controlled by advanced programming techniques that divide workload among the computers in a way that the 8-core processor receives more tasks. Also, a parallel implementation has to arrange computations so that they match the specifics around the way graphical processors operate. Furthermore, when using conventional desktop computers (like in our testing environment) one has to consider the fact that every desktop computer has a graphical processing unit installed on it, because each machine must support displaying of graphical user interfaces. Having this in mind, a reasonable idea is using these processors for performing computations as well. This is a challenging task due to the difference in the architecture of these processors and the fact that they are optimized for performing a single operation over and over again on different input data sets (e.g. addition of vectors). However, in the case of a cluster (like the one built upon testing) using the available GPUs doubles the number of available processors. Of course, making use of such processors requires different programming tools and techniques (e.g. CUDA). These require further studying, so that eventually an application can be executed in parallel on all types of available processors.

Finally, having an HPC cluster does not always provide the best resources for testing and development. This is why, the achieved cluster can be included into a large-scale network like, for example, the Austrian Grid. The grid is an innovative undertaking that aims to combine computing resources distributed geographically. It incorporates techniques for distributing workload among machines which fall under different administrative control and follow different security policies. In fact, the project is relatively young and still requires research. This is why, including the cluster in such a network can provide further opportunity for testing. Grid-oriented parallel applications can be executed and analyzed. Parallel algorithms can be tuned to run faster in such an environment. What is more, ROCKS already supports grid job submission with Condor.

References

[Barley, 2009] Blaise Barney, Lawrence Livermore National Laboratory, *Message Passing Interface (MPI),* https://computing.llnl.gov/tutorials/mpi/#MPI2, Last Modified: 01/26/2009, Last accessed in April 2009

[Buytaert, 2004] Kris Buytaert, *Introducing openMosix*, 2004, O'Reilly Media, Inc, SysAdmin, Last accessed in April 2009, http://www.linuxdevcenter.com/pub/a/linux/2004/02/19/openmosix.html

[Buyya vol.1, 1999] Rajkumar Buyya (editor). *High Performance Cluster Computing: Architectures and Systems, Vol. 1*. ISBN 0-13-013784-7, Prentice Hall PTR, NJ, USA, 1999.

[Buyya vol.2, 1999] Rajkumar Buyya (editor*) High Performance Cluster Computing: Programming and Applications Vol.2*. ISBN 0-13-013785-5, Prentice Hall PTR, NJ, USA, 1999.

[CAOSHome, 2009] Home page of CAOS http://www.caoslinux.org/, last accessed in May 2009

[CAOSWiki, 2009] Wiki page of CAOS: http://wiki.caoslinux.org/Main_Page, last accessed in May 2009

[CentOS, 2009] CentOS home page: http://www.centos.org/, last accessed in June 2009

[Condor, 2009] Condor Home Page: http://www.cs.wisc.edu/condor/ , last accessed in April 2009

[Condor-Manual, 2009] Ross Moore, Nikos Drakos, Condor Version 7.0.5 Manual: http://www.cs.wisc.edu/condor/manual/v7.0/, last accessed in July 2009

[Dague, 2002] Sean Dague, *System Installation Suite, Massive Installation for Linux.* Proceedings of the Ottawa Linux Symposium 2002, June 26th–29th, 2002 Ottawa, Ontario, Canada. , Pages 93-106. Online copy: http://www.kernel.org/doc/ols/2002/ols2002-pages-93-106.pdf

[Dongarra, 2004] Jack Dongarra and Alexey Lastovetsky, *An Overview of Heterogeneous High Performance and Grid Computing*, in Engineering the Grid, Edited by Beniamino Di Martino, Jack Dongarra, Adolfy Hoisie, Laurence Yang, and Hans Zima, Nova Science Publishers, Inc., 2004

[Foster, 2005] Ian Foster*, A Globus Primer. Or, Everything You Wanted to Know about Globus, but Were Afraid To Ask. Describing Globus Toolkit 4,* An Early and Incomplete draft, 5/8/2005, www.globus.org/primer

[Geist, 1994] Geist A., Beguelin A., Dongarra J., Jiang W. , Manchek R., Sunderam V., *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. Online copy at: http://www.netlib.org/pvm3/book/pvm-book.html

[Gropp, 1999] William Gropp, Ewing Lusk, Anthony Skjellum, Using MPI – Portable Parallel Programming with the Message Passing Interface, 2nd edition, MIT Press, Nov. 1999, ISBN-10: 0-262-57132-3 ISBN 13: 978-0-262-57132-6

[Hadoop, 2009] Hadoop Home Page, last updated on 02/24/2009, last accessed in April 2009

[Hadoop HDFS, 2009] HDFS Architecture last accessed in April 2009: http://hadoop.apache.org/core/docs/current/hdfs_design.html

[Hadoop M/R, 2009] Map/Reduce Tutorial, last accessed in April 2009: http://hadoop.apache.org/core/docs/r0.19.1/mapred_tutorial.html

[Hoffman-Condor, 2003] Forrest Hoffman, *Cluster Management with Condor, Part I,* article in Linux Magazine October 15th, 2003, online copy: http://www.linux-mag.com/id/1483, © Linux Magazine 1999-2009. LinuxMagazine.com v4.0, last accessed in July 2009.

[Hoffman-Ganglia, 2003] Forrest Hoffman, *Cluster Monitoring with Ganglia*, article in Linux Magazine August 15th, 2003, online copy: http://www.linux-mag.com/id/1433, © Linux Magazine 1999-2009. LinuxMagazine.com v4.0, last accessed in July 2009.

[Layton, 2009] Jeffrey B. Layton**,** *CAOS NSA and Perceus: All-in-one Cluster Software Stack,* Linux Magazine, February 5th 2009, online article at: http://www.linux-mag.com/id/7239, last accessed in May, 2009

[Layton, 2008] Jeffrey B. Layton, *Perceus/Warewolf: Tres Cool Cluster Tool*, Linux Magazine, July 7th 2008, online article at: http://www.linux-mag.com/id/6386, last accessed in May, 2009

[Liedert, 2005] Daniel Liedert, *Debian Repository mit debarchiver HOWTO. Ein eigenes und komfortables APT(-GET-bares)-Repository mit debarchiver aufsetzen.* Version 0.96, Online Tutorial: http://debian.wgdd.de/howto/howto-aptrep, last updated on 2005-12-03, last accessed in June 2009

[Massie, 2004] Matthew L. Massie, Brent N. Chun, David E. Culler, *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience*, Available online since 15 June 2004: http://ganglia.info/papers/science.pdf, Parallel Computing, vol. 30, Issue 7, July 2004, 817–840

[MAUI, 2009] MAUI Scheduker Home Page, last accessed in April 2009, http://www.clusterresources.com/products/maui/docs/mauiusers.shtml

[Meglicki, 2004] Zdzislaw Meglicki, *High Performance Data Management and Processing*, *Document version Id: I590.tex, v. 1.197 2004/04/29*, Section 7462, Course notes, University of Indiana, USA, http://beige.ucs.indiana.edu/I590/

[Moche, 2002] Moche Bar, S. Cozzini, M. Davini, A. Marmodoro, *openMosix vs. Beowulf: A Case Study*, paper presented at HPC, the linux cluster Revolution 2002 (St. Petersburg, Florida, USA, october 2002)

[MPI-IO, 2003] Maui High Performance Computing Center – SP Parallel Programming Workshop: MPI-IO, 2003, http://www.mhpcc.edu/training/workshop2/mpi_io/MAIN.html, Last Modified on 09/06/2003, last accessed in April 2009.

[Open MPI, 2009] Open MPI Home Page: http://www.open-mpi.org, Page last modified: 19-Mar-2009, last accessed in April 2009

[OSCAR, 2009] Oscar Administration Guide-
http://svn.oscar.openclustergroup.org/trac/oscar/wiki/AdminGuide, last accessed in April 2009

[Papadopoulos, 2002] Philip M. Papadopoulos, Caroline A. Papadopoulos, Mason J. Katz, William J. Link and Greg Bruno, *Configuring Large High-Performance Clusters at Lightspeed: A Case Study*, December 2002 , Clusters and Computational Grids for Scientific Computing 2002, Faverges, France, last edited: May 7, 2004, online copy: http://www.rocksclusters.org/rocks-doc/papers/workshop-clusters-lyon/paper.pdf

[Perceus, 2009] Perceus User Guide: http://www.perceus.org/docs/perceus-userguide-1.5.0.pdf, February 10th 2009, published by www.infiscale.com, last accessed in May 2009

[PVFS2, 2009] Parallel Virtual File System Ver. 2 home page: http://www.pvfs.org/ , last accessed in April 2009

[PVFS, 2009] The Parallel Virtual File System Project: www.parl.clemson.edu/pvfs/ , last accessed in April 2009

[ROCKS, 2009] ROCKS home page: http://www.rocksclusters.org/, last accessed in June 2009

[Sector-Sphere, 2009] Sector-Sphere Home Page: http://sector.sourceforge.net/ , and User Manual: http://sector.sourceforge.net/doc/index.htm, Last modified: 19-Feb-2009. Both last accessed in April 2009

[Sloan, 2004] Joseph D. Sloan, *High Performance Linux Clusters With Oscar, Rocks, OpenMosix & Mpi*, O'Reilly, Nov.2004, ISBN 10: 0-596-00570-9 / ISBN 13: 9780596005702

[Torque-Manual, 2009] Torque Admin Manual, ©2001-2009 Cluster Resources, Incorporated, http://www.clusterresources.com/products/torque/docs/, last accessed in July 2009.

[Yunhong, 2008] Gu Yunhong, Robert Grossman, *Exploring Data Parallelism and Locality in Wide Area Networks, Workshop on Many-task Computing on Grids and Supercomputers (MTAGS), co-located with SC08, Austin, TX. Nov. 2008.*

Appendix A

Parallel implementation of Numerical Integration of a function, Chapter 4 “Tool Evaluation”, Section 4.2 MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>
#include "mpi.h"

#define NUM_PIECES 2147483647

long double calculateArea(long double a_step, int a_myid, int a_numprocs, char*
a_processor_name)
{
        unsigned long int i = 0;
        long double sum = 0.0;
        long double area;
        long double x_old, x_new;
        clock_t start = clock();
        double end;

        for(i = a_myid; i < NUM_PIECES; i+=a_numprocs)
        {
                x_new = (long double)(i+1) * a_step;
                x_old = i * a_step;
                area = ( ( ((long double)4/(1+(x_new*x_new))) + ((long
double)4/(1+(x_old*x_old))) ) * a_step ) / 2;
                sum+=area;
        }
        end = ( (double)clock() - start) / CLOCKS_PER_SEC;
        printf ( "Calculation Time on %s: %lf sec.\n", a_processor_name, end);

 return sum;
}

int main(int argc, char* argv[])
{
      int myid;
      int numprocs;
      int namelen;
      char processor_name[MPI_MAX_PROCESSOR_NAME];
      double wtime;
      clock_t start_time = clock();
      double end;
      long double step;
      long double sum = -1.0;
      long double mysum = -1.0;

      MPI_Init(&argc, &argv);

      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
      MPI_Get_processor_name(processor_name, &namelen);

      //Start Measuring Total Computation Time
      MPI_Barrier(MPI_COMM_WORLD);
      if(myid == 0){
            wtime = MPI_Wtime();
            puts("Parallel Numerical Integration of 4/(1+sqr(x)) in C");
      }
	//DOWORK
	step = 1.0 / NUM_PIECES;
	mysum = calculateArea(step, myid, numprocs, &processor_name);
	MPI_Reduce(&mysum, &sum, 1, MPI_LONG_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
	//END OF DOWORK

	MPI_Barrier(MPI_COMM_WORLD);
	if(myid == 0){
		printf("Elapsed Wall Clock time is %.10lf sec.\n", MPI_Wtime()-
wtime);
		printf("Area is %.56Lf\n", sum);
	}

	MPI_Finalize();

	return 0;
}
```

Curriculum Vitae

PERSONAL INFORMATION

Name	Georgiev, Stefan Georgiev
Date of birth	17 February 1985
Birth place	Varna 9000, Bulgaria
Telephone Number	(+359) 89 831 20 57
E-mail	stefan.georgiev.bg@gmail.com
Nationality	Bulgarian
Marital Status	Single

WORK EXPERIENCE

July 2007 – September 2007	Internship at **IBM** Bulgaria /Software Tester on IBM *Rational Functional Tester*/ (Address: World Trade Center, 36 "Dragan Cankov" blv., 9th fl.) www.ibm.com/bg
July 2006 – April 2007	Service Technician/remote technical support/at **Siemens Enterprise Communications EOOD** Bulgaria (Address: Kukush 2 str., 1309 Sofia, Bulgaria) www.siemens.bg
October 2004 – January 2005	Internship at **Haemimont AD** Bulgaria /juniour web developer/ www.haemimont.com

EDUCATION AND TRAINING

September 2008 – Current	MSc/Master of Science/ in **Informatics**, International School of Informatics (ISI) Hagenberg, Johannes Kepler University Linz, Upper Austria, Austria
October 2004 – July 2008	BSc/Bachelor of Science/ in **Computer Science**, Sofia University "St. Kliment Ohridski", Faculty of Mathematics and Informatics, Sofia, Bulgaria, http://www.fmi.uni-sofia.bg
1999 – 2004	High School of Mathematics "Dr. Petar Beron", Varna, Bulgaria, **major**: Informatics, Mathematics, English

<u>SIEMENS training courses:</u>

July 2006	HiPath 4000 – Basic course for Service
July 2006 – August 2006	HiPath 4000 - Basic Networking for Service
August 2006	Introduction to Cisco Networking technologies

August 2006	ICND – Interconnecting Cisco Network Devices
August 2006	QoS – Quality of Service training
August 2006	VoIP – Voice over IP Workshop
September 2006	CISCO Certified Network Associate Certificate **/CCNA/**
September 2006 – October 2006	HiPath IP Technology

PERSONAL SKILLS AND COMPETENCES

- Easily adaptive
- Communicative
- Fast-learning
- Responsible
- Devoted

LANGUAGES

Bulgarian	Native
English	
• Reading skills	Advanced
• Writing skills	Advanced
• Verbal skills	Advanced

TECHNICAL SKILLS AND COMPETENCES

- Operating Systems – Windows, DOS, Linux
- Programming Languages

C	very good
C++	basic
C#	basic
SQL	basic

- Database management – basics in MS SQL 2005
- Network Management Skills

theoretical knowledge of routed protocols (TCP/IP) and routing protocols(RIP, OSPF);
building and managing small office/home office LAN networks;
basic configuration of CISCO routers and managable switches.

OTHER SKILLS AND COMPETENCES

August 2003	**Driving License**, category „B” and „M” /for driving cars and small motorcycles/

HOBBIES AND PERSONAL INTERESTS

Sports, Music, Cinema

Erklärung

Ich erkläre an Eides statt, das ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. Die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe

Hagenberg am 17 Juli 2009

……………………….
Stefan Georgiev