FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

JULI Project - Final Report

Ulrich Detert, Andreas Thomasch, Norbert Eicker**,*
*Jeff Broughton*** (Eds.)*

FZJ-ZAM-IB-2007-05

March 2007

(last change: 2.4.2007)

(*) IBM Deutschland GmbH
(**) ParTec Cluster Competence Center GmbH
(***) QLogic Corporation

JULI Project

Final Report

- March 2007 -

Ulrich Detert (Forschungszentrum Jülich GmbH)

Andreas Thomasch (IBM Deutschland GmbH)

Norbert Eicker (ParTec Cluster Competence Center GmbH)

Jeff Broughton (QLogic Corporation)

(Editors)

The JULI project aimed at developing and evaluating a parallel compute cluster based on IBM's BladeCenter H with JS21 nodes, QLogic's InfiniPath network components and ParTec's ParaStation software. The project was carried out as a collaboration of Forschungszentrum Jülich GmbH, IBM Deutschland GmbH, QLogic Corporation[1] and ParTec Cluster Competence Center GmbH. This report details the milestones and results of the project.

1 Project Overview

The JULI project was targeted to evaluate a prototype of a next generation of cluster computing as a joint research activity of partners from industry and academia. The aim was to integrate a first-of-a-kind cluster architecture based on PowerPC processor technology, InfiniPath interconnect and ParaStation cluster middleware. While each of these components existed individually before the JULI project, their combination into a "best-of-breed" cluster was new. It required the development of an InfiniPath adapter card in a blade form factor, firmware, driver, and MPI support as well as porting of various software components to SLES10 for PowerPC. Partners of the project were: Forschungszentrum Jülich GmbH, IBM Deutschland and IBM Deutschland Entwicklung GmbH, QLogic Corporation and ParTec Cluster Competence Center GmbH. The project started in March 2006 with the availability of the first prototype hardware and ended in December 2006 with the evaluation of the integrated system. Some concluding work was done in the beginning of 2007.

The project was structured in two phases, phase 1 being focused on hardware and basic software integration, phase 2 being related to software integration, cluster development and application case studies. The following list marks important milestones in the two phases:

Phase 1

- Development of hardware prototypes: QLogic InfiniPath HighSpeed daughter card, Voltaire Pass-Through module (IBM, QLogic, Voltaire)

[1] Former PathScale before acquired by QLogic in April 2006 and becoming QLogic's *System Interconnect Group.*

- Enable basic system functions: Operating system and device driver support, basic low-level communication (IBM, QLogic, FZ Jülich)
- More complex integration tests: CSM, QLogic MPI on a larger system of up to 14 JS21 blades (IBM, QLogic, FZ Jülich)
- Final integration with full size system of 62 blades; cluster ready for prototype interconnect evaluation (IBM, QLogic, FZ Jülich)
- Shipment of full size cluster to Forschungszentrum Jülich; cluster ready for further evaluation by FZ Jülich (IBM, QLogic, FZ Jülich)

Phase 2

- Verification of the prototype cluster by a larger set of synthetic benchmarks (FZ Jülich)
- Integration of batch system into the prototype cluster (IBM, FZ Jülich)
- Integration of ParaStation into the prototype cluster (ParTec, FZ Jülich)
- Early user access to prototype cluster with selected real-world applications (FZ Jülich)
- Integration of GPFS (General Parallel File System) into the prototype cluster (IBM, FZ Jülich)
- Verification of usability, stability, scalability and maintainability of the integrated system (FZ Jülich)

At the time of delivery, this special solution was not expected to have production-ready reliability and performance, but to allow a serious evaluation with customer application codes running on this new cluster computing platform.

2 System Configuration *(Ulrich Detert, Olaf Mextorf, Andreas Thomasch)*

The following section describes the target system architecture and its components. This includes hardware and software components. The system was developed in several steps as indicated in the previous section. The configuration given here outlines the final target system's architecture [3].

Figure 1 depicts the basic system configuration. The system includes four BladeCenter H chassis for compute nodes, each equipped with 14 JS21 blades, and one chassis with 6 blades for management, front-end and I/O nodes.

Two Nortel switches in each chassis provide for Ethernet connectivity: they connect 14 ports that attach to the blades to six external ports. The external ports can be connected to other Nortel switches in other BladeCenter-H chassis, or to a separate Ethernet switch. The Gig-E network comprises two physical networks, one for administrative tasks and cluster control, the other for GPFS I/O. The InfiniPath host channel adapters in each node connect via Voltaire IB Pass-Through modules in each chassis to the central Voltaire ISR9096 IB switch. The IB 4x network has a peak transfer rate of 1 GB/s per link and direction.

Each JS21 compute blade forms an SMP node with 2 dual core PowerPC 970 MP processors (2.5 GHz) and 4 GB memory (533 MHz DDR2/ECC). Each PPC 970 core can execute 2 Multiply-Add instructions per clock cycle. Thus, a node delivers 40 GFLOPS peak, and the full system with 56 compute nodes has 2.24 TFLOPS peak. The JS21 includes two on-board Ethernet ports. In the compute nodes and login node, the single-port InfiniPath HCA (in *CFF-e* form factor) is used as the HPC interconnect for MPI traffic.

The physical layout of the system is given in Figure 2. The system comprises two racks, one holding the

56 compute blades, the corresponding Pass-Through modules and the Voltaire switch, the other containing one blade chassis with the 6 service nodes, a DS4100 storage subsystem and the 48-port Gig-E Cisco switch. The DS4100 storage subsystem is equipped with 14 disks SATA 400 GB, 7200 rpm. It is directly connected to the four I/O nodes through Fibre Channel, using a BladeCenter optical pass-through module and a small form-factor (*SFF*) Qlogic FC adapter card in the I/O nodes.

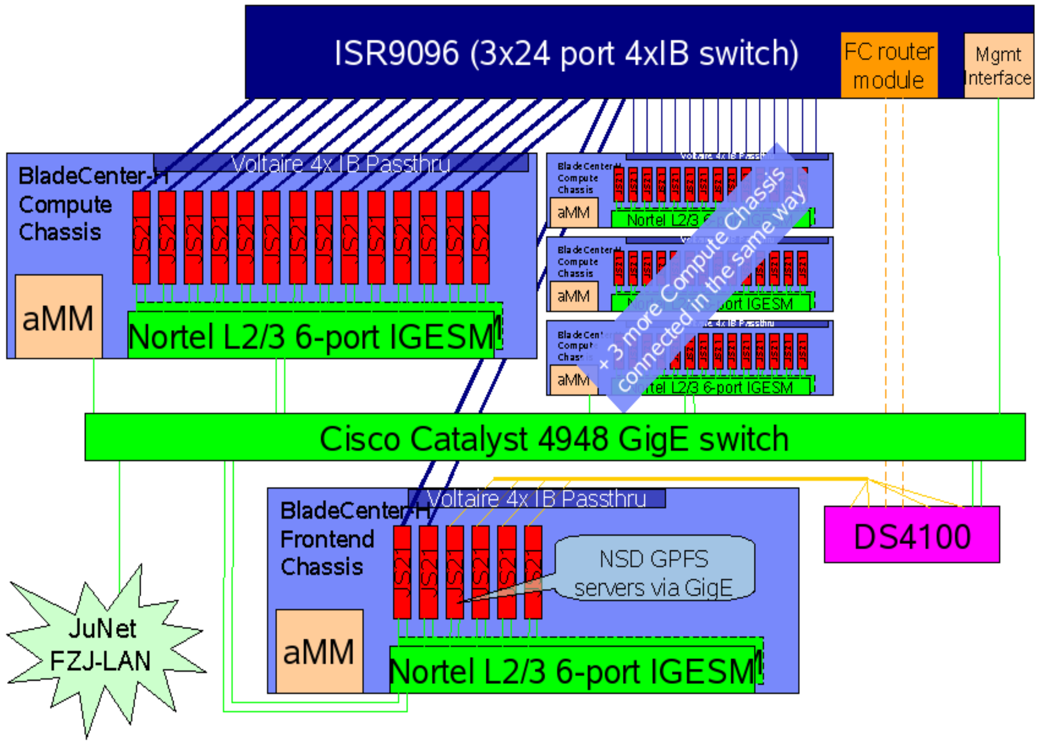

ISR9096 (3x24 port 4xIB switch)
FC router module
Mgmt Interface
BladeCenter-H Compute Chassis
Voltaire 4x IB Passthru
JS21
aMM
Nortel L2/3 6-port IGESM
+ 3 more Compute Chassis connected in the same way
Cisco Catalyst 4948 GigE switch
BladeCenter-H Frontend Chassis
NSD GPFS servers via GigE
DS4100
JuNet FZJ-LAN

Figure 1: *System configuration*

The software stack of the JULI system includes the following components:

- Linux operating system SLES10, kernel 2.6.16, with modifications for InfiniPath adapter cards and GPFS-specific kernel modules
- IBM Blade firmware SLOF 645 with InfiniPath support
- IBM Cluster Systems Management (CSM) 1.5
- ParaStation 4
- QLogic InfiniPath drivers 2.0
- QLogic MPI 2.0
- IBM GPFS 3.1
- IBM XLF 10.1 Fortran Compiler
- IBM XLC 8.0 C/C++ Compiler
- gcc 4.1 GNU C Compiler, gfortran Fortran Compiler
- Mathematical libraries: IBM ESSL 4.2, GotoBLAS 1.07, LAPACK 3.0, ScaLAPACK 1.7.2

- Torque 2.1 batch system
- IBM LoadLeveler 3.4 batch system
- IBM TSM 5.2 backup client

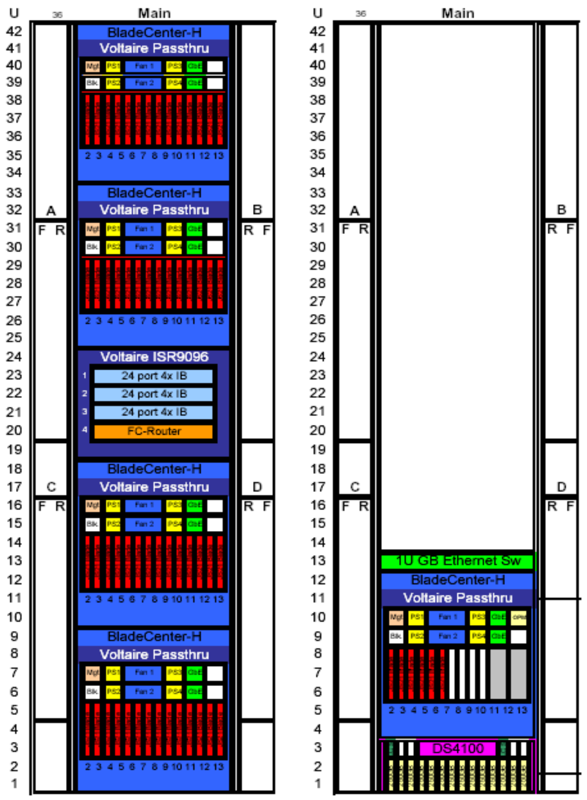


Figure 2: *Physical layout*

3 System Evaluation

System evaluation was done during all steps of the project. This included functionality and performance tests of the integrated hardware and software components, assessment of the usability and maintainability of all significant procedures and features and the evaluation of performance and scalability of selected applications and benchmarks on the cluster.

The following sections summarize the results in selected areas.

3.1 Development and Bring-Up

3.1.1 Hardware Bringup (Heiko Schick)

The following paragraphs describe how the hardware bringup was done for project JULI.

For the JULI project we used Slimine Open Firmware (SLOF) for the IBM JS21 instead of the official firmware image, because support for PCI Express devices, Message Signaled Interrupts and the QLogic InfiniPath InfiniBand HCA was not in place when the project was started.

To support the QLogic InfiniPath/InfiniBand HCA it was necessary to initialize the PCI Express bus correctly and enable Message Signalled Interrupts (MSIs). The device is addressed via the memory space: the firmware has to program the BASE Address Registers (BARs) by writing the corresponding address to the configuration space of the PCI device.

In SLOF this initialization is done by routines that are responsible to do the whole bus walk. During the bus walk SLOF also geographically addresses the PCI slots via the PCI controller. The PCI bus is enumerated according to the vendor ID, device ID for each possible combination of buses, devices, and functions.

To make sure that this setup was done properly, the first step was to verify that the IBM JS21 and the QLogic InfiniPath device are working together. Problems in this area will prevent firmware and Linux to recognize the device. After the device was recognized, the next logical step was to verify, if the BAR setup was done correctly and no overlapping memory windows occurred.

Message Signaled Interrupts were verified via a small inbound IB (loopback) tests, because the InfiniBand subnet management is complex and problem determination is very difficult. Because of that, an InfiniBand loopback connector was plugged to the QLogic InfiniPath device instead of a switch. This test also verified all data paths (for PCI Express it is essential that all lanes are working in all possible combinations without problems).

An InfiniBand subnet consists of Host Channel Adapters (HCA), switches and routers. The subnet management is done by an application, which is called Subnet Manager (SM). The SM is responsible for discovering, configuring, activating and managing the subnet. Typically the Subnet Manager is an application that is running directly on the InfiniBand switch. The counterpart of the SM is the Subnet Manager Agent (SMA). A Subnet Manager Agent is in every device (or system) and generates or responds to control packets called Subnet Management Packets (SMPs). In most cases the SMA configures all local components via SMPs that are sent via the unreliable datagram service. To activate an InfiniBand port the SM communicates with the SMA and sets the port attributes (e.g. LID) via SMPs.

In general the outbound test has passed successfully when the IB port is active, because of the communication, which was already done between the SMA and the SM. Problems in this area mostly indicate that DMAs or interrupts (MSIs) are not working correctly.

The last step of the hardware bringup was to execute further IB tests (e.g. stress-runs and link handling on disruption) and to do micro-benchmarking on the main scenarios (e.g. latency).

3.1.2 ***System Installation*** *(Torsten Bloth)*

To provide a tested and fully configured hardware environment to FZ Jülich, IBM installed the prototype cluster in the IBM labs in Böblingen. The InfiniBand cabling itself was the tricky part of the hardware installation. Such cabling was never done before at same scale within the IBM BladeCenter environment. The challenge was to place all 62 thick cables in a regular 19" server rack and not disturb the airflow of each of the BladeCenter chassis.

To get started rapidly with the initial software bring-up the whole cluster was installed with the help of SuSE's AutoYast. The setup of CSM was planned as a future task. Later on, the kernel was patched with necessary patches for MSI and SLOF support and distributed to the whole cluster. With this new kernel image it was now possible to flash the new firmware and to enable the InfiniBand HCAs.

For getting a first feeling of the whole system, the cluster was stressed with some basic performance tests. The InfiniBand HCA came with a fully featured package, including the driver, the MPI library and low level test programs. Those test programs were very helpful to get the IB interconnection up and running. We started with simple loopback tests and scaled up to all nodes.

The next test was done with the IMB suite [9], a set of benchmarks targeted at measuring the most important MPI functions. The benchmark starts with simple point-to-point tests (like ping-pong) and ends up at an all-nodes "All-To-All" test.

The last test and also an indication for the computational performance of the cluster was the popular HPL (High Performance Linpack) benchmark [11]. As for other tests we started with settings for only one node and scaled up to the whole 56 node setup. The scale-up was as expected and as estimated before. More details on the IMB and Linpack benchmarks are given later in this report.

*3.1.3 **Network** (Olaf Mextorf)*

The network of the JULI cluster was set up using three separate IP subnets mainly for cluster management traffic (CSM, ParaStation), for filesystem traffic (NFS, GPFS) and for the evaluation of IP-over-InfiniBand (IPoIB).

Regarding the network design given in Figure 1, the maximum throughput of Ethernet traffic between any two blade chassis is limited to 6 GBit/s (in each of the two Gigabit networks). By default, the IBM Cluster1350 configuration contains only a single Ethernet wire from each of the 2 Nortel switches at each chassis to the central Cisco switch, resulting in only 1 GBit/s in each network. Especially when looking at the four file servers located in a single chassis, this is a potential bottleneck for blades from other chassis in accessing services (NFS, GPFS) from these servers. To provide more aggregate Ethernet bandwidth between chassis, we established and tested Gigabit Ethernet channeling (IEEE 802.3ad LACP) between the Nortel and the Cisco switch (the Cisco Catalyst 4948G is able to handle 48 GBit/s full duplex Ethernet traffic).

For a further increase in peak-bandwidth available for a single blade, other BladeCenter hardware options exist (like a copper pass-through module which would provide full GE bandwidth to each blade). Such options have not been studied in project JULI.

Concerning the management of the components, the CLI of the Nortel switches, based on a set of "full screen menus", are a little bit unusual and not as convenient as the de facto standard - the line oriented Cisco IOS - is. Especially the configuration is a kind of unreadable compared to IOS. At the chassis management (aMM) we suffered a little from a very primitive IP-stack implementation, giving not even the possibility to define dedicated routes but only a default route. Assuming a certain necessity for accessing the aMM from some workstations outside the dedicated and isolated management IP-subnet for management reasons (especially in the case of all blades of a chassis being in trouble), the only possibility of protecting the aMM is by externally implemented Access Control Lists (ACL), e.g. at the Cisco Catalyst.

Regarding the network configuration of the blades we decided after some trouble at the beginning, especially related to hardware changes, to have the configuration of the Ethernet adapters based on their unique and system wide identical PCI slot position instead of the MAC address, giving even a better environment for scripting all over the cluster. In addition we raised the MTU size at the blades and configured Jumbo frames at the network components to increase the network throughput, especially during the GPFS tests. During the GPFS-tests we used the SPAN-feature (Switch Port Analyzer) of the central Cisco Catalyst switch for an in depth view into the GPFS traffic and some analysis of the performance.

3.2 CSM *(Karsten Kutzer, Michael Hennecke)*

As CSM is being used as the management software on the FZJ supercomputers, one of the project goals was to set up CSM [12] on the JULI cluster. Initially, an early version of CSM 1.5 for SLES10 has been used and later updated to the generally available CSM 1.5.1. Most of this work was standard CSM installation. Some insights from the installation are summarized below.

3.2.1 Hardware Control

To perform hardware control functions for a node, CSM needs to access a hardware control point and a console server for the node. For BladeCenter nodes, these functions are both performed by the BladeCenter's Advanced Management Module. In the node definition, *PowerMethod=blade* and *ConsoleMethod=blade* are set, and the *HWControlPoint* and *ConsoleServerName* fields are set to point to the aMM's IP name/address. A blade within that chassis can then be addressed by the *HWControlNodeId* and *ConsolePortNum* fields. After this setup, the CSM HW control commands *rpower* and *rconsole* worked as expected, as did the derived comands like *csmstat*.

One lesson learned with this setup was that there are actually two separate "service users" required on the aMM, which CSM uses to connect for HW control and console access:

- For *rpower*, the default superuser profile *USERID* is assumed by CSM, and no setup is needed on the aMM.
- For *rconsole*, a new ID with only "Blade Server Remote Console Access" authority is needed on the aMM. CSM assumes a profile named *RMTCON*, and also assumes a default password for that user.

While CSM assumes default passwords for these users, they should be changed to site-specific passwords in the aMM menus. To change the passwords on the CSM side, you can use the *systemid* command. Assuming the password is the same for all chassis, this password can be set globally for all nodes with the *blade PowerMethod*, using the *-p PowerMethod* option:

```
systemid      -p blade  USERID  # then enter USEIRD's password
systemid  -c  -p blade  RMTCON  # then enter RMTCON's password
```

If individual chassis have different passwords, those can still be set individually using aMM's hostname rather than the *-p PowerMethod* option.

3.2.2 Provisioning and Node Installation

With CSM, the *lshwinfo* command can be used to aquire information about the blades (like their UUID identifiers), which can then be used to create the node definitions for CSM. It can be run with the *-p PowerMethod* option to provide information for all blades, or the *-c <aMM-ip-addr>* option to access a specific BladeCenter H management module.

The *getadapters* command can be used to get the MAC addresses of the nodes, which are needed for node installation. For BladeCenter nodes, the collection method can be specified as *-m hwstat*. The *csmsetupyast* command will automatically invoke *getadapters* to determine the information on network adapters, if it is not already stored in the node definiton at the time *csmsetupyast* is invoked.

To prepare node installation in a SLES environment, the CSM command *csmsetupyast* is used. It will copy the SLES product CDs to a location that CSM can later use (typically */csminstall/Linux/SLES*). This step only needs to be done for the first node and can be suppressed for subsequent node installs. The *csmsetupyast* command then prepares the installation through the SUSE *AutoYaST* mechanism: it sets up necessary system services like DHCP (adding the nodes to be installed into */etc/dhcpd.conf*) and *tftpboot*, configures an Apache or NFS server, and also adapts the image files to be used with *AutoYaST*. CSM ships some *AutoYaST* XML tempates which can be customized by the administrator if needed.

3.3 System Administration and Maintenance *(Ulrich Detert)*

3.3.1 Hardware Control

As outlined in the previous section, hardware control on the JULI cluster is implemented as a two-level procedure. On the blade center level, the Advanced Management Module (aMM) gives access to each single blade center chassis comprising up to 14 JS21 blades, switch modules, pass-through modules, power modules, fan packs, blowers, the front panel controls and a media tray. On the cluster level, CSM combines aMM functionality into a cluster-wide hardware monitoring and control system. This is realized by remote logins from the CSM management node to the aMM instance related to a given blade or blade center component.

aMM comes in two flavours: GUI based or CLI based. The aMM GUI is very handy for all sorts of hardware maintenance on individual blades or blade center components. The following list summarizes the most important features of the GUI for this purpose:

- System status (power status, LEDs, event log, power consumption, hardware and firmware vital product data)
- System control (blade and I/O module power on/power off/restart, firmware upgrade)
- MM control (login profiles, alert configuration, network settings)

The command line interface comprises essentially the same functions as the aMM GUI. On the JULI cluster it is mainly used to gain console access to individual blades during HW maintenance.

CSM combines aMM functions into cluster-wide management functions. The following list highlights some important CSM commands:

- *rpower* - power on/power off/reset/power status of individual blades or blade selections
- *csmstatus* - cluster status
- *rconsole* - console login
- *reventlog* - collect individual blade center event logs (useful for archiving and monitoring cluster-wide hardware event logs)

3.3.2 Maintenance Procedures

During the development and evaluation phase of the JULI cluster, maintenance and administration procedures were somewhat different from regular procedures as expected for production-like systems. This was partly caused by the numerous changes applied to hardware and software during the project, partly by the fact that JULI runs with modified firmware and kernel. This especially complicated maintenance tasks like replacing a blade by new hardware. For the same reason, RAS features like the mirroring of system disks could not be tested during the course of the project. It is expected that future firmware releases supporting InfiniPath will fully comply to standard maintenance procedures and, thus, eliminate deficiencies of the prototype JULI firmware.

Another inconvenience with respect to system administration showed up in the storage area. When defining additional Fibre Channel LUNs on the DS4100 storage device, the device numbering on the Linux nodes connected to DS4100 was shifted by the number of newly allocated LUNs. The reason for this is that the Fibre Channel devices are detected before the local disks. The only current workaround for this problem is to rename the local devices in the boot loader configuration and also in the Linux file system table (*/etc/fstab*); otherwise the renumbering will render unbootable nodes (unless the new boot device is manually specified as an option to the boot loader at boot time).

Apart from these complications, maintenance and administration of the JULI cluster is well supported by the respective software components. CSM, as already mentioned, provides for full hardware control and, in addition to that, cluster configuration and monitoring features and proved to be very reliable and flexible in the tested Linux cluster environment. The integration of additional software components for user administration (NIS), data management (NFS and GPFS) and job handling (*Torque* and *LoadLeveler*) was straightforward and the coexistence of QLogic MPI and the IBM and GNU compilers revealed no problems.

3.4 CPU and Memory *(Norbert Eicker, Thomas Lippert)*

3.4.1 Architecture

JULI consists of 56 IBM JS21 BladeServers. Each BladeServer is equipped with 2 Dual-Core PowerPC 970MP CPUs running at 2.5 GHz [17]. The PowerPC CPU has a pipelined, super-scalar architecture. Each core features two full-blown floating-point units (FPU) with 21 stages. Every FPU allows one double-precision multiply-add operation per cycle. Altogether this leads to a theoretical peak-performance of 10 GFlop/s per core.

Furthermore, each PPC 970MP CPU carries a vector-extension, called VMX-unit. It is - more or less - comparable with Intel's and AMD's SSE units. However, a VMX can only handle single-precision numbers. Since the VMX-unit is also fully pipelined and capable to conduct one multiply-add operation on 128-bit registers in every cycle, the peak-performance for single-precision operations even reaches 20 GFlop/s per core. (In real life this theoretical number is limited by memory bandwidth, however.)

Each core has its own L1 and L2 cache hierarchy. While the L1 cache is segmented into 32 kB for data and 64 kB for instructions, the 1 MB L2 cache is used for both data and instructions. The main difference besides its size is the latency of the cache access. While it takes 2 cycles to fetch data from L1 cache, the processor has to wait 14 cycles until data from the L2 cache are available.

The latency of the main memory (4 GB) is significantly larger and amounts to O(100) cycles. Of course, the bandwidth of the main memory is much smaller than that of the cache as well, and depends on the actual type of memory used. For the JS21 blades two varieties of DDR2 memory are available; slower SDRAM modules, running at 400 MHz, and faster ones, clocked with 533 MHz. In fact, we were able to test both types of memory. This enabled us to study the memory sub-system in detail and to analyze the effects of the different memory-speeds on both synthetic low-level benchmarks and real-world applications.

3.4.2 The Lmbench Test

In order to determine the on-node capabilities of JULI we ran the *lmbench* suite [6, 7] containing low-level performance benchmarks on one of JULI's compute-blades. We will discuss two sets of results in more detail: on the one hand, results concerning the raw compute-performance, i.e. the critical parameters of the FPU engines of the PPC CPU, on the other hand, measurements of the capabilities of the memory sub-system [8].

As mentioned above, during the JULI project the compute-nodes have been equipped with two different types of memory modules: JULI started with 400 MHz DDR2 SDRAM modules and upgraded to faster memory running at 533 MHz. This enables us to make interesting comparisons concerning the effects of the memory bandwidth.

FPU Performance

The latencies and throughput values of the PPC 970MP processor are given in the data-sheet. They are redisplayed in columns 2 and 3 of Table 1. We confirmed the specifications by carrying out the corresponding test within the *lmbench* suite.

Operation	Latency	Throughput	Single	Double
Add	6	2/cycle	2.38	2.38
Mult	6	2/cycle	2.38	2.38
Div	33	2/28 cycles	13.1	13.1

Table 1: *FPU execution times in ns for PPC 970MP from lmbench*

The benchmark results are presented in columns 4 and 5. Based on a cycle-time of 0.4 ns - in correspondence with a 2.5 GHz processor clock - the results agree with the values in the data-sheet. This gives us confidence that results reported from *lmbench* are reliable and the PPC CPUs work properly.

Memory Bandwidth

Another set of results from the *lmbench* suite is discussed with the aim to understand the memory sub-system of the JS21 blades. The corresponding numbers are presented in Table 2.

The upper part of the table shows results obtained with the older and slower memory (400 MHz), the lower part results are obtained with the faster modules (533 MHz). The four lines of each block show the outcome of the test running with 1, 2 or 4 instances simultaneously. Column 2 denotes the number of instances. The difference between the two lines referring to two instances is due to the cores that are used within a single test as indicated in column 3.[2]

Columns 4 and 5 of table Table 2 show the results for consecutive reads and writes to the main memory testing the memory bandwidth of the system. The results for read operations are significantly larger than for write operations. In HPC practice this should be no major problem since for most algorithms the reading access to memory dominates the write operations.

Furthermore, it is interesting to see that for one instance of the test the performance between the two types of memories only differs slightly. For the two instances on the same socket it doesn't change at all. The interpretation of these results is that in these cases the actual bottleneck is not the main memory but the sustained bandwidth of the processor socket.

On the other hand, using all four cores of the JS21 system we see a bandwidth gain of more than 20% between slower and faster memory.

[2] The *lmbench* suite allows to pin processes on processor-cores. In our runs two different tests with 2 instances were made: One running the processes on core 0 and 1, the other putting them on cores 0 and 2. Accordingly, in the first case, we run two processes on cores residing on the same processor-socket and in the second case we execute on a single core on the two different sockets of the JS21 system

Nevertheless, it is clearly visible that for all applications with a performance characteristic that is sensitive to memory bandwidth - as is the case for many applications in HPC - we cannot expect a linear scaling within a node, even if there is no communication between the processes. Even for the faster memory the total read-bandwidth obtained for four processes is only twice as big as the one we see for one process. This is due to two effects: on the one hand, it is a consequence of the JS21 system architecture; the main memory is connected to the northbridge of the system and has a peak bandwidth of 8.5 GB/s while each processor socket is able to handle 5.0 GB/s of throughput[3]. On the other hand, a single core can read from memory with 2.8 GB/s while the two cores on the same socket can increase this value only by less than 50%.

MHz	Procs	Cores	BW [MB/s]		Latency [ns]		
			read	write	L1	L2	mem
400	1	any	2750	1740	1.19	5.2	40.7
	2	0 1	4000	2280	1.19	5.24	60.5
		0 2	4480	2295	1.19	5.24	52.2
	4	0 1 2 3	5090	2280	1.19	5.24	99.5
533	1	any	2830	1810	1.19	5.2	39.1
	2	0 1	4020	2590	1.19	5.24	60.0
		0 2	4870	2730	1.19	5.24	45.3
	4	0 1 2 3	6141	2635	1.19	5.24	81.6

Table 2: *Results for bandwidth and latency from lmbench's memory test suite*

In the last 3 columns of Table 2 results for the memory latency are presented. As naively expected, the results for the L1 and L2 caches in columns 6 and 7 show no dependence concerning the type of memory used. The 8th column exhibits the latency of the access to main memory. Here effects of the type of memory only show up if the instances of the test make use of both sockets. We interpret this as follows: while for one process we see the actual latency of 40 ns, the number of 60 ns in the case of two instances on one socket is due to congestion within the processor socket. Furthermore the latency for two processes pinned on different sockets is increased due to rivaling accesses to the main memory, which - at least partly - can be weakened by faster memory.

3.5 InfiniPath Network

3.5.1 Concept (Norbert Eicker)

The InfiniPath network [18, 19] of JULI is devoted to MPI applications. 10-Gigabit technology is used for this network: QLogic's implementation of the InfiniBand standard called InfiniPath. Its most remarkable feature is an extremely low latency compared to other implementations of InfiniBand. The main difference of the InfiniPath implementation lies in the architecture of the host channel adapter (HCA). While all other solutions use a full-fledged CPU in order to implement the protocol, QLogic was

[3] This problem is also found on the current Intel XEON architecture. AMD's Opteron platform has a memory controller within each processor socket and can avoid these problems.

able to map the logic on a state machine and to realize it within an ASIC.

While QLogic's solution is based on a special hardware implementation, the wire-protocol is perfectly conforming with the InfiniBand standard. As a result, standard InfiniBand switches can be employed. In the case of JULI, a Voltaire ISR 9096 switch has been chosen with three line-cards summing up to a total of 72 ports.

*3.5.2 **MPI Ping-Pong and Ring Test Performance** (Ulrich Detert)*

Ping-pong and ring message tests have been used to asses the MPI performance on a basic level. More detailed MPI performance data are given in the “Benchmarks and Applications” section below. Figure 3 shows the uni-directional bandwidth of a ping-pong message within a SMP node (intra) and between nodes (inter). In addition, the mean bandwith per communication pair is given for a ring message sent in store-and-forward manner across all 56 nodes (224 tasks). The latter test has been used mainly as a functionality test and not so much as a performance test. Still, it gives some hints as to which communication performance to expect for real-world applications. The communication pattern for the ring test is such that messages are sent from processor 0 to 1, 2 and 3 within a SMP node, and then crossing borders from processor 3 in one node to processor 0 in the next node. Thus, one inter-node message is sent after three intra-node messages on each node.

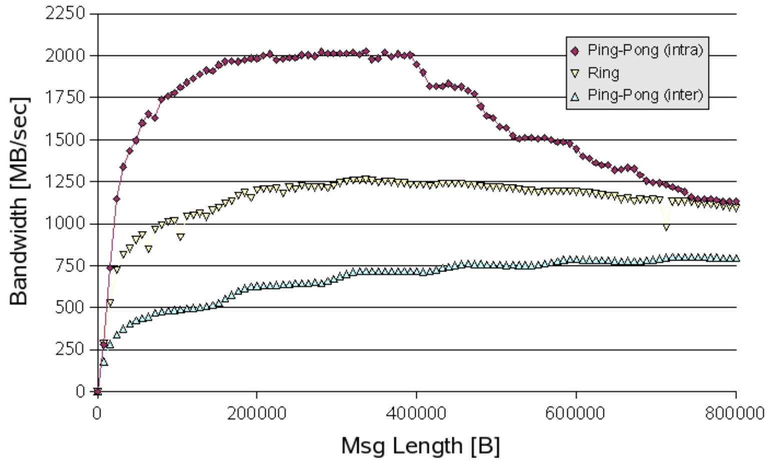


Figure 3: *MPI performance*

*3.5.3 **IP over InfiniPath** (Ulrich Detert)*

The InfiniPath network implementation allows to use the IP protocol over InfiniBand/InfiniPath by way of specific kernel modules (*ipath_ether*). Even though this was not formally part of the project, IP over InfiniPath has been configured and tested on the JULI cluster. It is not used under production conditions, however, since the InfiniPath network is mainly devoted to MPI trafic. Furthermore, IP over InfiniPath can currently not be used for GPFS data transfer: Due to the limited space in a blade, the *CFF-e* InfiniPath adapter and the current *SFF* FC adapter do not fit into one blade simultaneously.

The following table compares TCP/IP performance for Gigabit Ethernet and InfiniPath. Notice that no specific effort has been undertaken to tune the IP over InfiniPath communication performance.

	MTU	Ping Latency	Bandwidth
Gig-E	1500	~ 0.1 ms	118 MB/s
Gig-E	9000	~ 0.1 ms	124 MB/s
InfiniPath	16384	~ 0.05 ms	244 MB/s

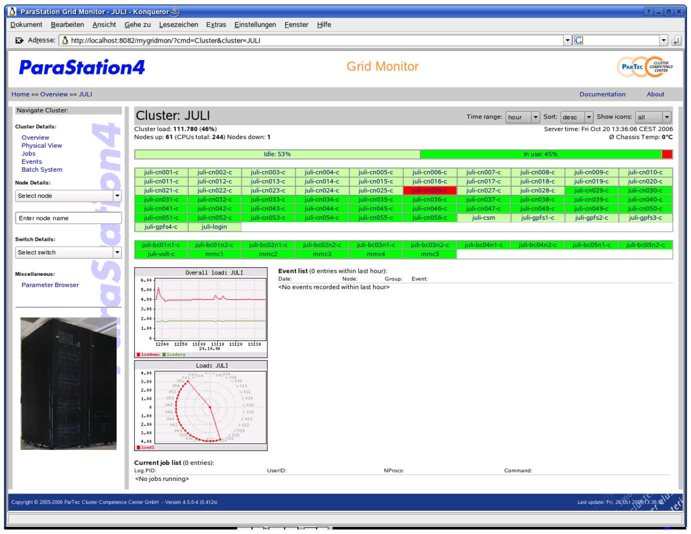
Table 3: *IP Network performance*

3.6 **ParaStation** *(Ralph Krotz)*

3.6.1 JULI Cluster Monitoring

The *ParaStation*® *GridMonitor* is a versatile system monitor for Linux-based compute clusters. A multiplicity of information from different devices and services from a cluster may be read, evaluated and stored. The *GridMonitor* provides the administrator with various aspects of the available information, from an overall status of all configured clusters to in-depth details of nodes and devices. Data can be grouped with respect to different aspects and are visualized using a web browser.

Furthermore, parameters may constantly be monitored and the administrator may be informed, if required.



http://localhost:8082/mygridmon/?cmd=Cluster&cluster=JULI

Cluster: JULI

Time range: hour Sort: desc Show icons: all

Cluster load: **111.780 (46%)**
Nodes up: **61** (CPUs total: **244**) Nodes down: **1**

Server time: Fri Oct 20 13:36:06 CEST 2006
Ø Chassis Temp: **0°C**

Idle: 53% In use: 45%

juli-cn001-c	juli-cn002-c	juli-cn003-c	juli-cn004-c	juli-cn005-c	juli-cn006-c	juli-cn007-c	juli-cn008-c	juli-cn009-c	juli-cn010-c
juli-cn011-c	juli-cn012-c	juli-cn013-c	juli-cn014-c	juli-cn015-c	juli-cn016-c	juli-cn017-c	juli-cn018-c	juli-cn019-c	juli-cn020-c
juli-cn021-c	juli-cn022-c	juli-cn023-c	juli-cn024-c	juli-cn025-c	juli-cn026-c	juli-cn027-c	juli-cn028-c	juli-cn029-c	juli-cn030-c
juli-cn031-c	juli-cn032-c	juli-cn033-c	juli-cn034-c	juli-cn035-c	juli-cn036-c	juli-cn037-c	juli-cn038-c	juli-cn039-c	juli-cn040-c
juli-cn041-c	juli-cn042-c	juli-cn043-c	juli-cn044-c	juli-cn045-c	juli-cn046-c	juli-cn047-c	juli-cn048-c	juli-cn049-c	juli-cn050-c
juli-cn051-c	juli-cn052-c	juli-cn053-c	juli-cn054-c	juli-cn055-c	juli-cn056-c	juli-csm	juli-gpfs1-c	juli-gpfs2-c	juli-gpfs3-c
juli-gpfs4-c	juli-login								

juli-bc01n1-c	juli-bc01n2-c	juli-bc02n1-c	juli-bc02n2-c	juli-bc03n1-c	juli-bc03n2-c	juli-bc04n1-c	juli-bc04n2-c	juli-bc05n1-c	juli-bc05n2-c
juli-volt-c	mmc1	mmc2	mmc3	mmc4	mmc5				

Event list (0 entries within last hour):
Date: Node: Group: Event:
<No events recorded within last hour>

Current job list (0 entries):
Log.PID: UserID: NProcs: Command:
<No jobs running>

Copyright © 2005-2006 ParTec Cluster Competence Center GmbH - Version 4.5.0-4 (0.412s)
Last update: Fri, 20.Oct 2006 13:36:06

Data Gathering Process (Collector)

All available data is retrieved and managed by a so called collector. All data is retrieved using dedicated agents for each device or service. These agents support various protocols, like e.g. SNMP. To minimize overall system and network load, only data requested by a client application is read from the agents by the collector. If no one is interested, no data is transferred and therefore no compute cycles and network bandwidth are used. The collector is especially designed to handle problems like dead nodes, broken network connections and limited network bandwidth, commonly found in cluster environments.

The collector gathers various data from different information sources available within a cluster, describing:

- compute nodes
- file servers and front-end nodes
- network devices
- storage devices
- runtime systems

Available parameters not only include operating system values, like system load, network counters or temperatures, but also parameters supplied by runtime systems like batch queuing systems, information provided by network switches or Blade Center Management Modules.

Each parameter is cached within the collector for the configured decay time. Using intelligent caching algorithms, multiple reading of data is avoided.

Data can be stored to and retrieved from a database. Thereby, a data history is available, e.g. for plotting diagrams. Parameters and sample frequencies can be configured independently. 'Virtual' parameters can be computed, monitored and stored to the database based on actual read data, e.g. the total system load as sum of all node load values.

Each known numerical value can be compared against an upper and lower limit. In case this value under-runs or over-runs those limits, an event will be generated. To constantly monitor these parameters, reading cycles can be defined.

Events describing abnormal situations within a cluster can be generated by monitoring parameter limits, node availability, etc. Events will be stored within the database and reported by email.

Beside the actual data, the collector also provides information about the type of available data. Based on this parameter type system, it's easy for a graphical user interface to construct dynamic selection boxes without actually reading the data and thereby wasting network bandwidth and compute cycles. This parameter type system also enables a user interface to include new parameters without modifying the scripts or page layout. E.g., newly added nodes will be recognized and shown automatically.

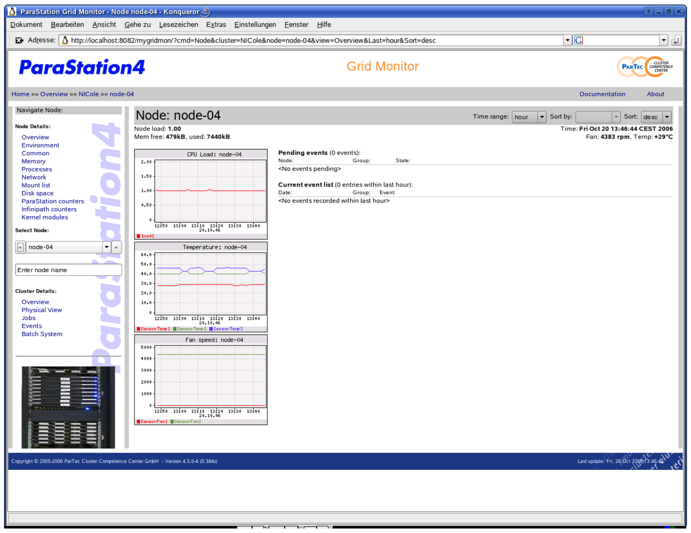
Monitoring data like temperatures or fan speeds within a node requires the *lmsensors* package or IPMI access for reading the values. Currently *lmsensors* is not supported by the JULI cluster hardware. Alternatively those values are available via SNMP. Due to the inconvenient format (output of human readable text strings instead of simple numbers) adapting the collector is more complicated but currently under development.

Graphical client (GUI)

The graphical user interface (GUI), based on a web server and PHP scripts, provides a comfortable access to the data, provided by one or more *collectors*. The information are grouped within various views, like multi-cluster or cluster dedicated overview and details, node overview and details, or cluster-wide parameter lists. Each web browser may be used to display these pages.

Views may be modified by pull-down menus defining the time range for lists and graphs, which icons to show, sort order, or columns to sort by.

The graphical user interface also provides information about currently pending events and event history. Parameters can be graphically shown as history diagrams for periodically sampled data, or bar and radar charts for current data.



http://localhost:8082/mygridmon/?cmd=Node&cluster=NICole&node=node-04&view=Overview&Last=hour&Sort=desc

Home »» Overview »» NICole »» node-04

Node Details:
Overview
Environment
Common
Memory
Processes
Network
Mount list
Disk space
ParaStation counters
Infinipath counters
Kernel modules

Select Node: node-04

Enter node name

Cluster Details:
Overview
Physical View
Jobs
Events
Batch System

Node: node-04

Node load: **1.00**
Mem free: **479kB**, used: **7440kB**

Time: **Fri Oct 20 13:46:44 CEST 2006**
Fan: **4383 rpm**, Temp: **+29°C**

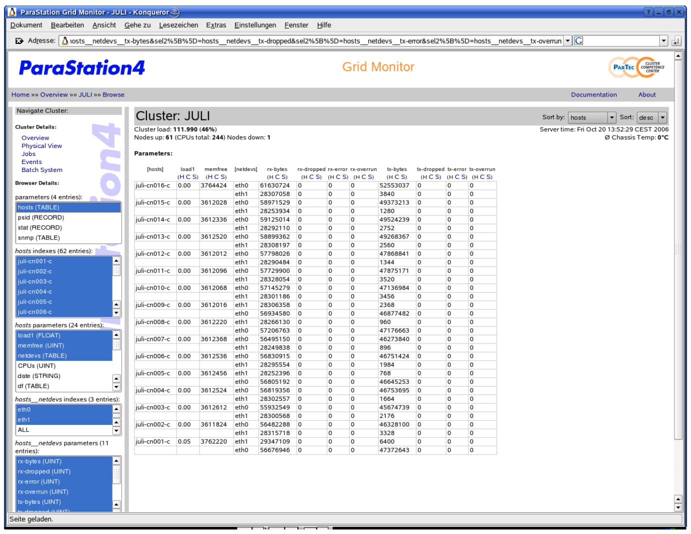
Pending events (0 events):
Node: Group: State:
<No events pending>

Current event list (0 entries within last hour):
Date: Group: Event:
<No events recorded within last hour>

Copyright © 2005-2006 ParTec Cluster Competence Center GmbH - Version 4.5.0-4 (0.384s)

Last update: Fri, 20 Oct 2006 13:46:44

Beside the predefined data views, a "parameter browser" is implemented within the graphical user interface. The parameter browser is a generic tool to display each particular parameter known to the collector.


ParaStation Grid Monitor - JULI - Konqueror
Dokument Bearbeiten Ansicht Gehe zu Lesezeichen Extras Einstellungen Fenster Hilfe
Adresse:
ParaStation4
Grid Monitor
ParTec Cluster Competence Center
Documentation
About
Navigate Cluster:
Sort by:
hosts
Sort:
desc
Seite geladen.

ıosts__netdevs__tx-bytes&sel2%5B%5D=hosts__netdevs__tx-dropped&sel2%5B%5D=hosts__netdevs__tx-error&sel2%5B%5D=hosts__netdevs__tx-overrun

Home »» Overview »» JULI »» Browse

Cluster Details:
Overview
Physical View
Jobs
Events
Batch System

Browser Details:

parameters (4 entries):
hosts (TABLE)
psid (RECORD)
stat (RECORD)
snmp (TABLE)

hosts indexes (62 entries):
juli-cn001-c
juli-cn002-c
juli-cn003-c
juli-cn004-c
juli-cn005-c
juli-cn006-c

hosts parameters (24 entries):
load1 (FLOAT)
memfree (UINT)
netdevs (TABLE)
CPUs (UINT)
date (STRING)
df (TABLE)

hosts__netdevs indexes (3 entries):
eth0
eth1
ALL

hosts__netdevs parameters (11 entries):
rx-bytes (UINT)
rx-dropped (UINT)
rx-error (UINT)
rx-overrun (UINT)
tx-bytes (UINT)

Cluster: JULI

Cluster load: **111.990** (**46%**)
Nodes up: **61** (CPUs total: **244**) Nodes down: **1**

Server time: Fri Oct 20 13:52:29 CEST 2006
Ø Chassis Temp: **0°C**

Parameters:

[hosts]	load1 (H C S)	memfree (H C S)	[netdevs]	rx-bytes (H C S)	rx-dropped (H C S)	rx-error (H C S)	rx-overrun (H C S)	tx-bytes (H C S)	tx-dropped (H C S)	tx-error (H C S)	tx-overrun (H C S)
juli-cn016-c	0.00	3764424	eth0	61630724	0	0	0	52553037	0	0	0
			eth1	28307058	0	0	0	3840	0	0	0
juli-cn015-c	0.00	3612028	eth0	58971529	0	0	0	49373213	0	0	0
			eth1	28253934	0	0	0	1280	0	0	0
juli-cn014-c	0.00	3612336	eth0	59125014	0	0	0	49524239	0	0	0
			eth1	28292110	0	0	0	2752	0	0	0
juli-cn013-c	0.00	3612520	eth0	58899362	0	0	0	49268367	0	0	0
			eth1	28308197	0	0	0	2560	0	0	0
juli-cn012-c	0.00	3612012	eth0	57798026	0	0	0	47868841	0	0	0
			eth1	28290484	0	0	0	1344	0	0	0
juli-cn011-c	0.00	3612096	eth0	57729900	0	0	0	47875171	0	0	0
			eth1	28328054	0	0	0	3520	0	0	0
juli-cn010-c	0.00	3612068	eth0	57145279	0	0	0	47136984	0	0	0
			eth1	28301186	0	0	0	3456	0	0	0
juli-cn009-c	0.00	3612016	eth1	28306358	0	0	0	2368	0	0	0
			eth0	56934580	0	0	0	46877482	0	0	0
juli-cn008-c	0.00	3612220	eth1	28266130	0	0	0	960	0	0	0
			eth0	57206763	0	0	0	47176663	0	0	0
juli-cn007-c	0.00	3612368	eth0	56495150	0	0	0	46273840	0	0	0
			eth1	28249838	0	0	0	896	0	0	0
juli-cn006-c	0.00	3612536	eth0	56830915	0	0	0	46751424	0	0	0
			eth1	28295554	0	0	0	1984	0	0	0
juli-cn005-c	0.00	3612456	eth1	28252396	0	0	0	768	0	0	0
			eth0	56805192	0	0	0	46645253	0	0	0
juli-cn004-c	0.00	3612524	eth0	56819356	0	0	0	46753695	0	0	0
			eth1	28302557	0	0	0	1664	0	0	0
juli-cn003-c	0.00	3612612	eth0	55932549	0	0	0	45674739	0	0	0
			eth1	28300568	0	0	0	2176	0	0	0
juli-cn002-c	0.00	3611824	eth0	56482288	0	0	0	46328100	0	0	0
			eth1	28315718	0	0	0	3328	0	0	0
juli-cn001-c	0.05	3762220	eth1	29347109	0	0	0	6400	0	0	0
			eth0	56676946	0	0	0	47372643	0	0	0

The JULI cluster utilizes Apache 2 as web server running on the management node.

Parameters are organized in a hierarchical way by using table and record entries. Each table provides indices and parameters, whereas each record only holds parameters. These parameters may be scalar parameters, like integers, floats or strings, or again may be tables or records. Examples for indices are host names or switch names.

Parameters may be provided by each data source known to the collector, therefore the parameter browser is e.g. also a SNMP (or MIB) browser. The data source is transparent to the parameter browser; therefore the user does not have to take care from which source the actual data is read. Arbitrary parameters may be shown together, like network counters of a node and switch.

The selected data is shown using a matrix layout, similar to a calculation sheet. It is organized in rows and columns, where the scalar parameters and indices show up in columns. Using pull-down menus, the matrix may be sorted increasing or decreasing by columns. All column data may be visualized using diagrams.

Software packaging

The *GridMonitor* is available as three RPM packages. These packages are:

- *pscollect:* includes the central *collector*, database and agents,
- *psgridmon:* comprises all PHP scripts and configuration files for the web server providing the graphical user interface,
- *psgridmon-doc:* provides all the documentation (HTML, PDF and man pages).

Installation of the *pscollect* package is required on all cluster nodes, the *psgridmon* packages must be installed on the web server node. Installing the *psgridmon-doc* packages is not required, but highly recommended. In particular, installing this package on the web server will provide online documentation within the *GridMonitor*.

3.7 Storage

*3.7.1 **NFS** (Ulrich Detert)*

As a first step into user-oriented cluster utilization, NFS was used for cluster-wide access to user data residing on the DS4100 storage subsystem. NFS proved to be stable especially when statically mounted. The use of the automounter worked, but seemed not to be very appropriate for a large number of nodes and home directories. Cross-mounting of multiple file systems from different server nodes, temporarily, lead to *NFS stale file handle* problems. This turned out to be a known bug in the Linux kernel and could be solved by an appropriate workaround (*no_subtree_check* server export option). No specific effort was made to further investigate the functionality or performance of NFS, since the main focus for cluster storage management was on GPFS.

*3.7.2 **GPFS** (Karsten Kutzer, Michael Hennecke)*

IBM GPFS [13] has been the global parallel filesystem shared across the IBM supercomputers at FZ Jülich for a couple of years already. As part of project JULI, GPFS version 3.1 has also been implemented on the JS21 cluster.

GPFS is IBM's parallel cluster filesystem, available on AIX since 1998 and on Linux since 2001. It provides parallel access to the same file or different files from a heterogeneous set of nodes, within a single GPFS cluster or even across multiple GPFS clusters. Because data and metadata traffic can be distributed across many servers as well as acoss disks, performance of data and metadata operations can be scaled with the available hardware.

GPFS cluster nodes mount GPFS as a local filesystem with full POSIX semantics, and the GPFS daemons running on the cluster nodes coordinate access to the disk storage. GPFS can use either a *SAN attachment* mode where all nodes in the cluster have direct access to the disk storage, or a *Network Shared Disk* (NSD) mode where only a few NSD servers have direct access to the disk storage, and data transport between the NSD servers and the other GPFS nodes is through a TCP/IP based network. The latter is more typical for HPC clusters, it is also used in project JULI where Gigabit Ethernet is used as the LAN.

By defining a primary and a backup NSD server for each disk, GPFS can also be configured to be highly available, so it will keep operational even in the event of a complete loss of a server. Disk storage can be made highly available both by redundancy and RAID levels in the storage controllers, as well as through replication mechanisms within GPFS which can be set individually for data and metadata. These features are already used in production on the other GPFS clusters at FZJ and have not been evaluated further on JULI.

One DS4100 storage controller with SATA disks has been configured with four RAID5 disk arrays, and the DS4100 is connected to four JS21 nodes which act as GPFS servers. The QLogic Fibre Channel adapter used in the JS21 is a small form factor (SFF) card, so it cannot be used in a blade in conjunction with the InfiniPath card which has a CFF-e form factor. For this reason, the second 1 Gig-E interface on each JS21 blade is used as the network for GPFS traffic. As an initial test, the user's home directories have been put under GPFS on the local storage subsystem. Eventually, the GPFS Multi-Cluster feature will be used to access production user data that resides on the *Jump* p690 cluster.

To improve the TCP/IP performance, Jumbo frames have been configured on the GPFS network. This required configuration of the Nortel switches in the BladeCenter H chassis, as well as setting the MTU='9000' attribute in the adapters' configuration file (*/etc/sysconfig/network/ifcfg-eth-id-<mac-addr>*).

The GPFS configuration itself was not different from other GPFS setups. The management blade and the four GPFS/NSD server nodes were configured as quorum nodes, and two of the GPFS/NSD servers also act as cluster configuration servers and filesystem managers.

Some time was spent analysing the network performance. The network topology is different from other clusters due to the Nortel switches in the BladeCenter H chassis, which connect 14 internal Ethernet ports of the blades with 6 external 1Gig-E connections. Not all six ports are used in our setup.

3.8 Batch Systems

3.8.1 ***Torque*** *(Birgit Naun, Norbert Eicker)*

Since *LoadLeveler* for PowerPC under Linux was not yet available at the time the Juli project started, a substitute batch system had to be used to start off. *Torque* with scheduler Maui was chosen, since this software is Open Source and is being used on other Linux clusters at FZJ already. *Torque* server version 2.1.2-5 was installed on the management node. Additionally, the *Torque* client and Mom software (version 2.1.2-5) had to be installed on all compute nodes and the login node. *Torque* is based on PBS (Portable Batch System), an Open Source batch and resource management system, and is available as rpm package for Linux.

Torque consists of four major components: Commands, Job Server, Job Executor and Job Scheduler. The most important user commands are *qsub* for submission, *qstat* for monitoring and *qdel* for job deletion. Operator or administrator commands like *qrun*, *qterm*, *qstart*, *qstop* or *qmgr* allow a very easy way to modify the configuration or start and stop *Torque* daemons. For cluster computing, the *pbsnodes* command is very important. It lists information about the configured nodes, or, with *–l* option, all nodes marked as down. The *qstat –a* command can be used for job summary; job details are displayed with *qstat –f <jobid>*.

Job Server (*bps_server*) is the central component of *Torque* and runs on the management node. All commands and daemons use the administrative Ethernet network configured on JULI. The Server's main function is to provide the basic batch services such as receiving/creating a batch job, modifying the job, protecting the job against system crashes, and running the job (placing it into execution). The accounting information are stored in directory */var/torque/server_priv/accounting*. The Job Server has to know the list of job execution nodes declared in a file in the server private directory *PBS_HOME/server_priv*. They can be modified by the *qmgr* command and are listed by *pbsnodes*. The communication between the server and all its defined *pbs_mom* processes allows *Torque* to react easily and automatically if nodes crash.

The Job Executor *pbs_mom* places a job into execution when it receives a copy of the job from the Server and therefore must be run on every node that can execute jobs. Mom creates a new session identical to the user's login session and returns the job's output to the user.

The job scheduler Maui is implemented as a daemon controlling the site's job execution policy. Currently, a standard and simpel FIFO policy is used on JULI. The policy and also the current batch queue structure are subject to modification for future production requirements.

Very important for the JULI cluster is the possibility to run parallel MPI jobs executed under QLogic's *mpirun*. The node information required for execution is transported from *Torque* to *mpirun* via a nodes file residing in the user's file space and named in the environment variable *$PBS_NODEFILE*.

Following is a typical example for a batch job script under *Torque*:

```
#PBS -l nodes=4:ppn=4
#reserve 4 nodes with 4 processors per node
#PBS -j oe
 ...
# shell script
NSLOTS=$(cat $PBS_NODEFILE | wc -l)
mpirun -np $NSLOTS -m $PBS_NODEFILE $PBS_O_WORKDIR/myprog
```

3.8.2 LoadLeveler (Ulla Ehrhart)

LoadLeveler [14] version 3.4 (*LoadL-full-SLES10-PPC64-3.4.0.1-0*) was installed on 8 compute nodes and the login node. There were only very few changes made to the standard configuration file: *ssh* was chosen as remote shell command, accounting turned on and the location for global history files was changed, which contain LL accounting information. The administration file was kept simple, all nodes defined had the same specification with the login node serving as central manager. The central manager was not running a startdaemon, thereby no jobs were started on the login node. There was one class defined on all nodes with the possibility to use all available resources, i. e. all CPUs. Serial jobs (initially no MPI jobs) were submitted, scheduled and executed by *LoadLeveler* without problems.

To start MPI jobs, i.e. parallel applications, the *LoadLeveler* keyword *job_type=MPICH* had to be defined in the job command file. *LoadLeveler* will then automatically set the environment variable *LOADL_HOSTFILE* with the filename that contains the host names assigned to the parallel job. Parallel jobs were thus submitted, scheduled and executed without problem, on single nodes as well as on multiple nodes.

Some limitations concerning memory limits under Linux should be considered. Memory cannot be limited in *LoadLeveler* under Linux for the following reasons:

- As documented in the *LoadLeveler*-manuals, *ConsumedMemory* and *ConsumedVirtual Memory* do not work under Linux as they require *Workload Manager* (WLM), which is available on AIX
- Data limit does not work (*LoadLeveler* can do no better than the Operating System)
- Stack limit works for stack segment areas only
- rss_limit does not work

The issue was transferred to IBM development. IBM intends to add more virtual memory limit support in future releases of *LoadLeveler* for Linux.

3.9 Compilers and Libraries

3.9.1 Compilers (Karsten Kutzer, Michael Hennecke)

Both the *gcc* compiler suite and the IBM XL compilers (XLF 10.1.1, XLC/C++ 8.0.1) [15, 16] have been used during the project. Initially, the QLogic MPI v1.3 software stack only supported *gcc*. With the availability of the XL Compilers for SLES10, QLogic/PathScale also provided support for the XL compilers starting with the QLogic MPI v2.0 release. Apart from minor issues, the porting went smoothly.

3.9.2 Mathematical Libraries (Inge Gutheil)

The BLAS library from K. Goto Release 1.07 (GotoBLAS) was installed in 64-bit addressing mode and with the Fortran interface for the XLF compiler. The performance of these BLAS routines was compared with that of the ESSL BLAS (ESSL version 4.2). The BLAS 1 routine AXPY performed better with ESSL than with GotoBLAS, whereas the BLAS 2 Routine DGEMV and the BLAS 3 Routine DGEMM performed slightly better with GotoBLAS (Figure 4, Charts 1, 2, 3).

We also installed LAPACK version 3.0 in 64-bit addressing mode. We only tested this library with the tests delivered with the library. All the tests were successful with GotoBLAS as well as with ESSL BLAS.

The only parallel mathematical library installed was ScaLAPACK version 1.7.2 together with BLACS version 1.1. Both were installed in 32-bit and 64-bit addressing mode.

We only made performance tests with the routine PDGEMM calling BLAS from ESSL. Here, we found that the performance per node of PDGEMM was much slower than the sequential performance of DGEMM. The performance degradation was worse, when the parallel program was executed on the four processors of one node than when executed on four nodes with only one process per node (Figure 4, Charts 4, 5, 6, 7).

3.10 Tools

3.10.1 Performance Tools (Bernd Mohr)

The group *Performance Optimization and Programming Environments* of ZAM investigated the porting and installation of various commercial and open-source performance tools. In a first step, the standard portable open-source performance analysis toolkits KOJAK (ZAM, Jülich, [1]), Scalasca (ZAM, Jülich, [2]) and TAU (University of Oregon, Eugene) were ported to the JULI platform. This required only minor changes in the configuration and installation procedures of the tools. TAU is an extensive code profiling package. KOJAK is an event trace generation and automatic performance bottleneck search toolkit. Scalasca is a new, more scalable version of KOJAK. All three tools were tested with various C and Fortran MPI benchmarks and application codes in 32-bit and 64-bit execution modes. In addition to MPI performance analysis, KOJAK and TAU also allow to measure and analyze CPU hardware counter data through the use of the PAPI library (ICL, University of Tennessee). This however requires Linux kernel modifications (perfmon patches) which are not available on JULI yet. The KOJAK measurement system also supports the generation of event traces for the commercial tools Vampir (Technical University

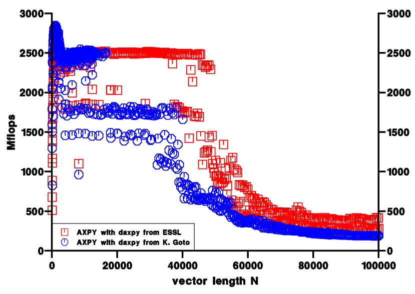


Chart 1

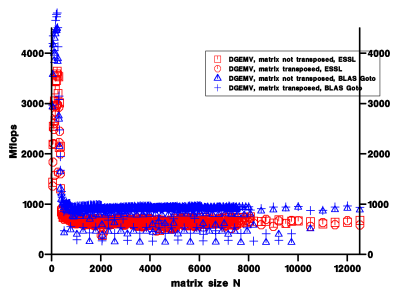


Chart 2

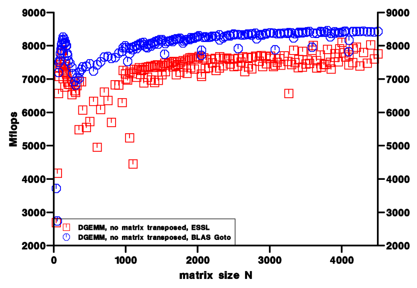


Chart 3

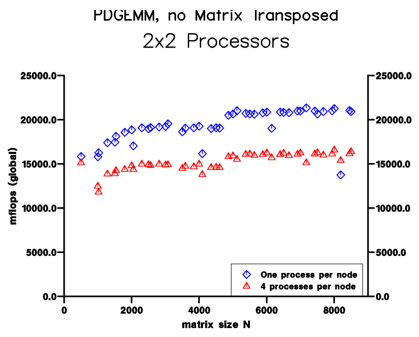


Chart 4

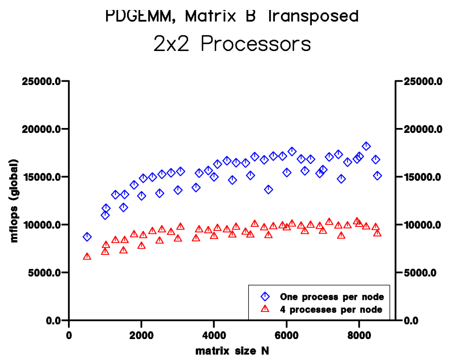


Chart 5

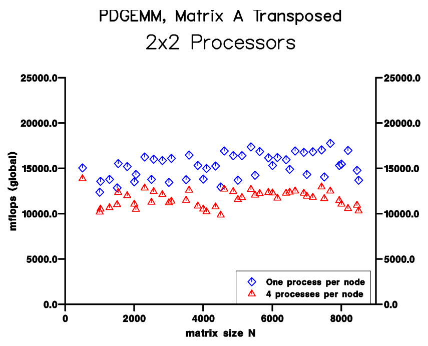


Chart 6

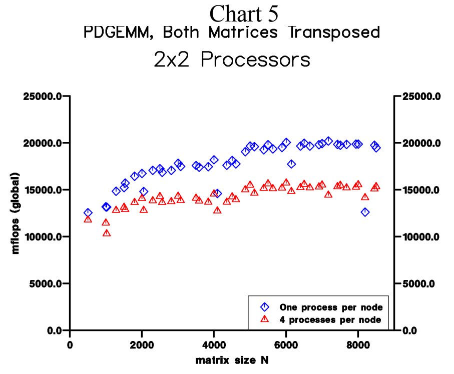


Chart 7

Figure 4: *Performance of BLAS routines in ESSL and Goto library*

Dresden, Germany) and Paraver (CEPBA, Barcelona, Spain). In collaboration with the respective vendors, these event trace visualization and analysis tools were ported to the JULI platform without any problems.

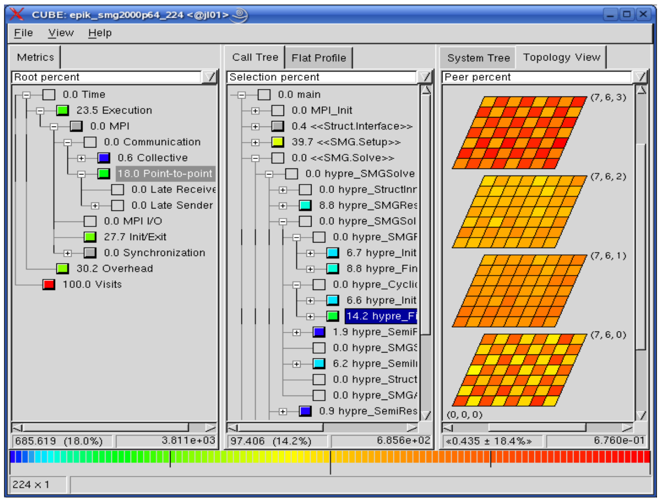
3.10.2 SCALASCA (Brian Wylie)

The current SCALASCA toolset (broadly equivalent to v0.5 with fixes) was built and tested on JULI with the ASC SMG2000 benchmark. The subject MPI application code was built with IBM XL compilers in 64-bit mode with the standard compilation options *-O3 -qarch=ppc970 -qtune=ppc970* (version *u*), along with two instrumented versions: one using selected POMP annotation directives of key routines and phases (version *p*), and the other using the XL compilers' undocumented automatic instrumentation (version *k*). A final measurement option was the specification of a *blacklist* of instrumented functions that were not to be traced (version *x*).

The SMG2000 run configuration was chosen as *n(64,64,32) c(0.1,1.0,10.0)* and 5 solver iterations for relatively short execution times. The 3-dimensional process mapping was determined from the MPI prefered Cartesian topology.

Measurements were made of these four configurations with varying numbers of processes to investigate scalability of the application itself, overheads associated with trace collection and parallel trace analysis (with SCOUT). Similar measurements were done on JULI (up to 224 processes), *Jump* (up to 1024 processes) and *Mare Nostrum* (up to 512 processes) (see [4]).

The uninstrumented SMG2000 showed good scalability of its parallel solver on each system, and superior performance on JULI (4.4s vs 4.6s and 6.9s with 128 processes). The instrumented versions showed the expected dilations proportional to the amount of (traced) instrumentation. Maximal process trace sizes grow slowly with the number of configured processes, whereas the total volume of trace data increases linearly. The largest trace collected on JULI was some 36 GB. Opening a trace file for each process was roughly 5 times more expensive than the serialised gathering and unification of definitions, however, both operations scaled well and introduced insignificant overheads, compared to the final flush and close of the trace files. Parallel trace flushing performance on JULI was typically 120 MB/s, with a best performance of 150 MB/s (compared to 150-170 MB/s on *Mare Nostrum* and up to 3200 MB/s on *Jump* with 1024 processes).



```
0.0 Time
  23.5 Execution
    0.0 MPI
      0.0 Communication
        0.6 Collective
        18.0 Point-to-point
          0.0 Late Receive
          0.0 Late Sender
      0.0 MPI I/O
      27.7 Init/Exit
      0.0 Synchronization
  30.2 Overhead
100.0 Visits
685.619 (18.0%)    3.811e+03
```

```
0.0 main
  0.0 MPI_Init
  0.4 <<Struct.Interface>>
  39.7 <<SMG.Setup>>
  0.0 <<SMG.Solve>>
    0.0 hypre_SMGSolve
      0.0 hypre_StructInr
      8.8 hypre_SMGRes
      0.0 hypre_SMGSol
        0.0 hypre_SMGF
          6.7 hypre_Init
          8.8 hypre_Fin
        0.0 hypre_Cyclic
          6.6 hypre_Init
          14.2 hypre_F
        1.9 hypre_SemiF
        0.0 hypre_SMGS
        6.2 hypre_SemiI
        0.0 hypre_Struct
        0.0 hypre_SMGA
      0.9 hypre_SemiRes
97.406 (14.2%)    6.856e+02
```

```
«0.435 ± 18.4%»    6.760e-01
```

Figure 5: *SCALASCA Sceenshot*

SCALASCA features parallel trace analysis using the SCOUT analyzer, an MPI application that loads corresponding subject rank traces and replays communication events when calculating performance metrics. The example screenshot from the SCALASCA analysis report explorer (CUBE) shows the distribution of point-to-point communication time in a key section of the SMG2000 solver when configured with 224 processes on JULI.

Initial SCOUT analyses of the larger traces suffered a significant performance degradation on JULI with more than 64 processes, which was identified as paging when the 1GB of memory per processor was exceeded. (*Mare Nostrum* and *Jump* have more memory per processor and avoided paging even when handling significantly larger traces.) A minor adjustment of SCOUT data-structures provided sufficient memory improvement to avoid paging when analyses were repeated with the new version. The result was broadly similar performance on JULI and *Mare Nostrum*, both significantly faster than *Jump* at comparable configurations.

The number of communication events recorded for each process configuration was the same for each trace experiment variation, however, the number of region enter/exit events varied considerably, from a small fraction of the total for the POMP "*p*" experiment to constituting the vast majority for the fully-instrumented (non-blacklisted) experiment "*k*". SCOUT trace loading times are proportional to the (maximum) number of events in the traces, and dominate total analysis times on JULI for all of the "*k*" experiments and for the "*p*" experiments with more than 100 processes (presumably due to filesystem bandwidth saturation). Similar behaviour may occur on *Mare Nostrum*, but only at more than 500 processes. Replay analysis performance is identical for JULI and *Mare Nostrum* with the communication-rich "*p*" experiments, however, *Mare Nostrum* is consistently 50% slower with the communication-poor "*k*" experiments, apparently indicating superior CPU off-loading of communication processing on JULI.

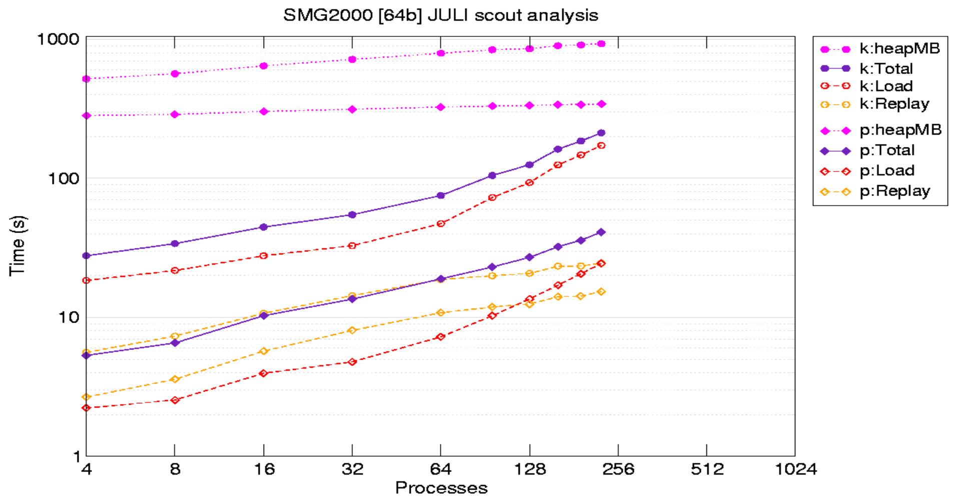


Figure 6: *SCALASCA performance on JULI*

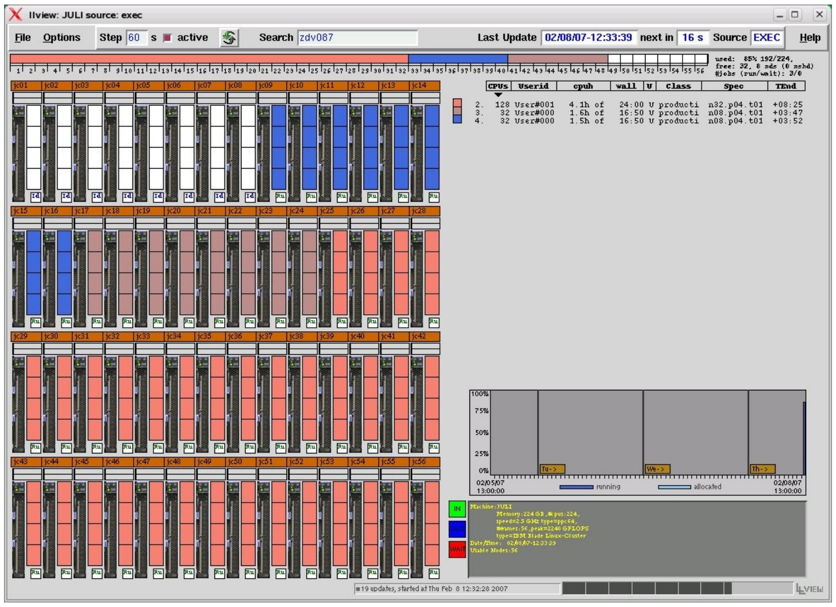
3.10.3 Graphical Job Monitor LLview (Wolfgang Frings)

LLview has been installed and adapted to JULI. LLview is a client-server based application which allows to monitor the utilization of clusters controlled by batch systems like IBM *LoadLeveler*, *PBS Pro* [10], *Torque*, or IBM *Blue Gene/L system data base*. It has been developed at the Central Institute for Applied

Mathematics, Research Centre Jülich.

The main part of the LLview client is the *node display*. It shows a small box for each processor of a SMP node. The box color represents the job running on this processor. Furthermore, the node display contains additional elements displaying global information about the node, like status, memory usage and CPU load for each SMP node. When moving the mouse pointer over a processor box, the corresponding information is highlighted in the other display elements of LLview. These elements are *job list*, *usage bar*, *information panel*, and a *utilization chart* (Figure 7).

The data access will be done by the server part of LLview, which, for *LoadLeveler*, uses the data access C-API of *LoadLeveler* to get the information about the node usage, running and waiting jobs. For the batch system *Torque*, LLview uses a Perl script for data extraction. In both cases the information is stored in XML format. The client part of LLview can access the data directly if the client runs on the same machine. However, the usual way is to distribute the data by a Web server to support clients running on local desktops. In this case, LLview accesses the data from the Web server with a user/password authentication method. (For more information and software download see [5].)



```
used:  85% 192/224,
free: 32, 8 nds (0 nshd)
#jobs (run/wait): 3/0
```

		CPUs	Userid	cpuh	wall	U	Class	Spec	TEnd
🟥	2.	128	User#001	4.1h of	24:00	U	producti	n32.p04.t01	+08:25
🟫	3.	32	User#000	1.6h of	16:50	U	producti	n08.p04.t01	+03:47
🟦	4.	32	User#000	1.5h of	16:50	U	producti	n08.p04.t01	+03:52

```
Machine: JULI
        Memory:224 GB ,#cpus:224,
        speed=2.5 GHz type=ppc64,
        #frames:56 ,peak=2240 GFLOPS
        type=IBM Blade Linux-Cluster
Date/Time:  02/08/07-12:33:39
Usable Nodes:56
```

Figure 7: *LLview - Graphical monitoring of batch jobs*

3.11 Benchmarks and Applications

3.11.1 The Intel MPI Benchmark (Norbert Eicker, Thomas Lippert)

Depending on the programming model, often MPI intra-node communication is required within a parallel application[4]. Therefore we investigate both intra-node and inter-node communication performance. We

[4] For clusters of SMP-nodes two different parallelization strategies are popular: MPI, leading to MPI intra-node communication as examined in this section, or a hybrid model using OpenMP within the SMP-node and MPI for inter-node communication.

employ the Intel MPI Benchmark (IMB) [8].

Intra-node Communication

Communication between processes allocated on the same node can be expected to be highly dependent on the available memory bandwidth. In general, this type of operation will not require the communication-hardware but relies on a segment of shared memory used to copy the data from the address-space of one process to another.

The IMB is another tool well suited to get a feeling of the capabilities of the memory sub-system of the JS21 blades. Within our tests of the intra-node communication we made use of two different implementations of MPI: on the one hand, we employed QLogic's InfiniPath-MPI library that also supports communication between the nodes. On the other hand, we used an implementation of MPI which is part of ParTec's ParaStation suite. ParaStation will serve as a reference implementation of local shared-memory communication.

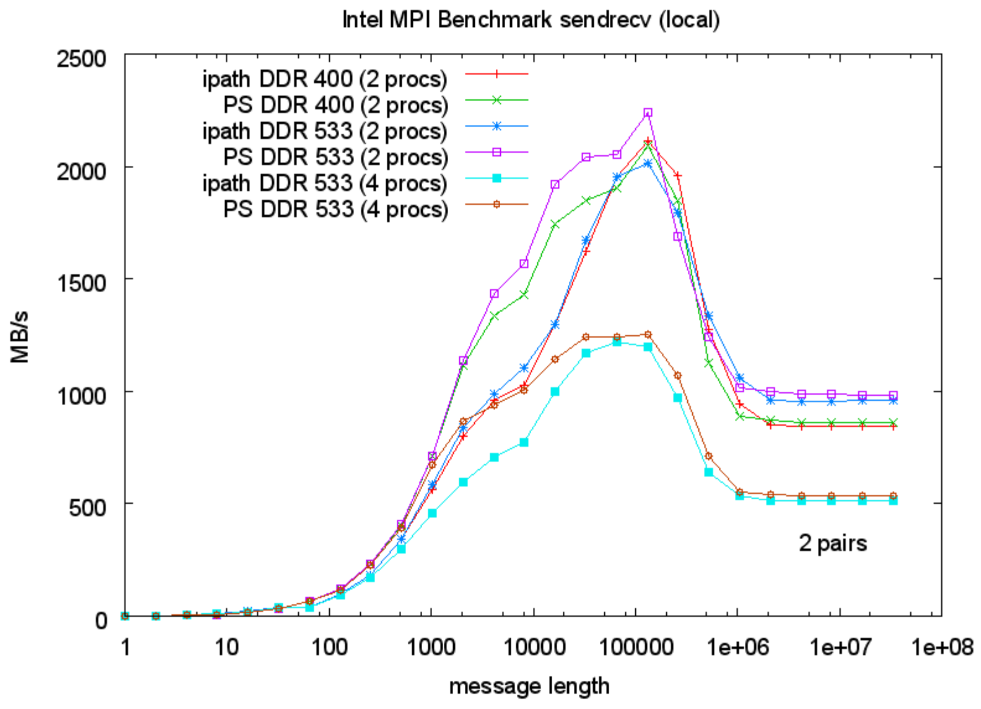


Figure 8: *Intra-node MPI communication bandwidth*

Figure 8 shows results of IMB's *sendrecv* test for various combinations of MPI implementation, type of memory modules and number of processes. We carried out tests between two processes (i.e. one pair passing messages), both types of memory, and with both MPI implementations. The results are plotted in the upper four graphs of Figure 8. The lower two show results obtained from runs with 2 pairs of processes. Here, only the tests with the fast memory are presented.

It can be observed that the ParaStation MPI shows consistently better performance than InfiniPath MPI. This is true for both types of memory and over the whole range of message lengths. In all cases one has to discriminate two regions of results: all message-sizes smaller than 1 MB will be handled within the caches of the involved CPU. Only messages larger than this threshold will actually be sent via the main memory. Accordingly, just results for larger messages are sensitive to the different memory speeds.

As expected, the throughput per process-pair is halved, when going from one communicating pair of processes to two pairs, as all the processes have to share the available memory bandwidth. Since one pair is able to fill the memory channel almost completely, the additional gain of bandwidth for two pairs turns out to be negligible.

Inter-node communication

The inter-node communication makes use of the InfiniPath MPI library. We ran all tests for both types of memory. However, we saw almost no differences. We conclude that the memory bandwidth is not a bottleneck for the inter-node communication.

By means of IMB's *pingpong* benchmark we determined the network latency on MPI level. We notice a half round-trip time of less than 2.75 µs, which is extremely good also for InfiniBand. However, the difference to results with InfiniPath on other PCIe Platforms - less than 2 µs - still is substantial. The complex architecture of the JS21 blades and the long way from the PPC CPUs to the InfiniPath HCAs presumably is responsible for this result[5].

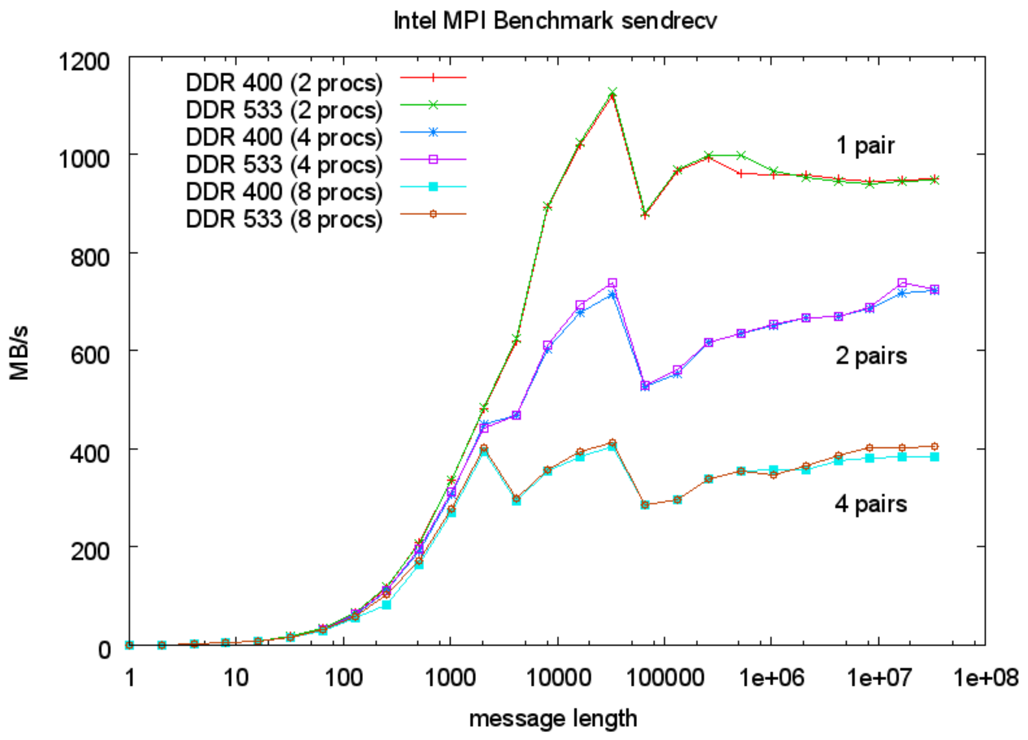


Figure 9: *Inter-node MPI communication bandwidth*

Further tests were carried out in order to analyze the bandwidth of the interconnect. Again we apply the *sendrecv* benchmark of IMB. In Figure 9 one has to distinguish three sets of tests: the two topmost graphs show results for communication between two processes, i.e. one pair of processes passing messages to

[5] A CPU access has to move from the processor to the northbridge then over HyperTransport to the PCIe bridge, then to the InfiniPath HCA.

each other. For the two graphs in the middle two pairs of processes make use of the same HCAs and physical wires at a time. The two graphs at the bottom are for four pairs of processes.

It is remarkable that for four pairs we get approximately the results we would expect naively from wire-speed numbers and estimates of the protocol overhead. Starting from 400 MB/s per pair, i.e. 1.6 GB/s accumulated bandwidth, one would anticipate a result of 800 MB/s for two pairs and 1.6 GB/s for one pair of processes. Neither of these expectations is met, actually, for one pair large messages show a throughput of less than 1 GB/s. This is most probably a software problem, either within the InfiniPath-MPI-library or inside one of the lower-level communication libraries that can hopefully be solved in the future. QLogic is working on this problem.

*3.11.2 **Linpack** (Ulrich Detert, Torsten Bloth)*

The Linpack benchmark [11] had been chosen as a formal validation criterion for the successful completion of Phase 1 of the JULI project. For the validation, no specific performance threshold had been defined. Yet, Linpack performance measurements have been used to assess hardware and software efficiency of the JULI cluster.

With the following parameters

- N=145500 (order of coefficient matrix A)
- NB=200 (partitioning blocking factor)
- P=14 (number of process rows)
- Q=16 (number of process columns)

the code delivers a sustained performance of 1.509 TFLOPS on 56 compute nodes (224 processors), which is 67.36% of peak. JULI-specific optimizations in this code were the use of the GotoBLAS library and appropriate compiler flags:

```
-O5 -qstrict -qtune=ppc970 -qarch=ppc970 -qmaxmem=-1 -DUSE_HUGETLB -DUSE_GOTO_ALIGN
```

Compared to the Power4+ based p690 cluster *Jump* at FZ Jülich, which delivers 5.568 TFLOPS Linpack out of 8.9 TFLOPS peak (62.5%) on 41 nodes with 32 processors each, JULI renders 1.58 times more Linpack performance per CPU and about 11 times the performance per floor space (considering that one frame p690 requires roughly the same floor space as one rack of JULI).

```
T/V                N    NB     P     Q               Time            Gflops
----------------------------------------------------------------------------
WR03R2L4      145500   200    14    16            1360.63         1.509e+03
----------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =       0.0606912 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =       0.0105652 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =       0.0017044 ...... PASSED
```

Table 4: *Linpack output (excerpt)*

*3.11.3 **Parallel Molecular Dynamics Simulations** (Godehard Sutmann)*

A new parallel force-decomposition algorithm for systems, consisting of particles which interact through short-range interactions was developed and tested on the JULI system. Molecular dynamics consist in solving the classical equations of motion for particles, interacting through given potentials. Calculating the forces on every particle makes it possible to integrate the system of ordinary differential equations and thus propagating the velocities and positions of particles in physical space. Using the fact that $F_{ij} = -F_{ji,}$

where F_{ij} is the force of particle j, acting on particle i, it is possible to concentrate on the upper triangular part of the force matrix. In the case of short-range interactions, only a limited region in physical space is needed to be explored in order to find potential interaction partners.

The new algorithm consists in first sorting particles according to a space filling Hilbert curve, thereby placing physically close partners also close in memory. In a second step the force matrix is partitioned according to the rule that every processor has approximately the same number of interactions to calculate. A dynamic load balance strategy ensures to have the same amount of work on each processor within small deviations.

Interaction partners are kept for a certain number of steps in a neighbor list, which is distributed among the processors. Therefore every processor knows, how many particles have to be sent to other processors or how many particles have to be received from remote processors, in order to calculate the total force on every particle. This kind of strategy dramatically reduces the amount of data to be transfered between processors, compared with traditional approaches, where whole vectors, containing particle positions are transferred between processors by global communication protocols.

Another feature of the algorithm is to use asynchronous communication between processors, if possible (Figure 10). Here it is used that while transferring data from remote processor J+1 to local processor I, interactions are calculated already between particles of processor J and I, thereby partly hiding the communication overhead.

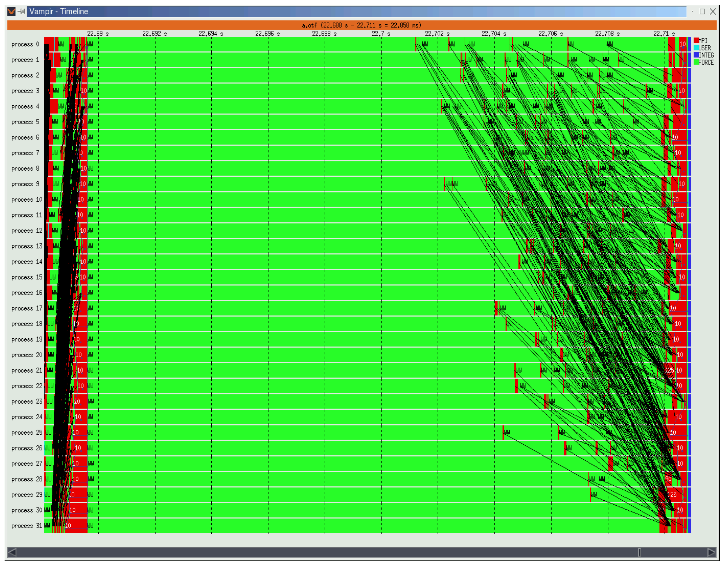


Figure 10: *Communication pattern of the algorithm for 32 processors. On the left side communication is synchronized by a barrier, which optimizes the performance. On the right side, data exchange is initialized by asynchronous send/receive operations, which are performed during the force loop (green color).*

Benchmarks of the new algorithm were carried out on JULI, showing scalability up to 128 processors (the maximum of used processors). Compared with the performance on the IBM p690 cluster *Jump* at FZ Jülich, the execution time speeded up by roughly a factor of two (Figure 11). The feature of asynchronous data transfer between processors could not be exploited on the JULI architecture. It was found that for larger number of processors the waiting times to receive data from other processors increased, showing that data transfer was not handled in the background, while doing computations. Even more, it was found that it was preferable to include an explicit barrier into the code in order to synchronise the processors, before entering into the double-loop (Figure 10), where inter-particle forces are calculated.

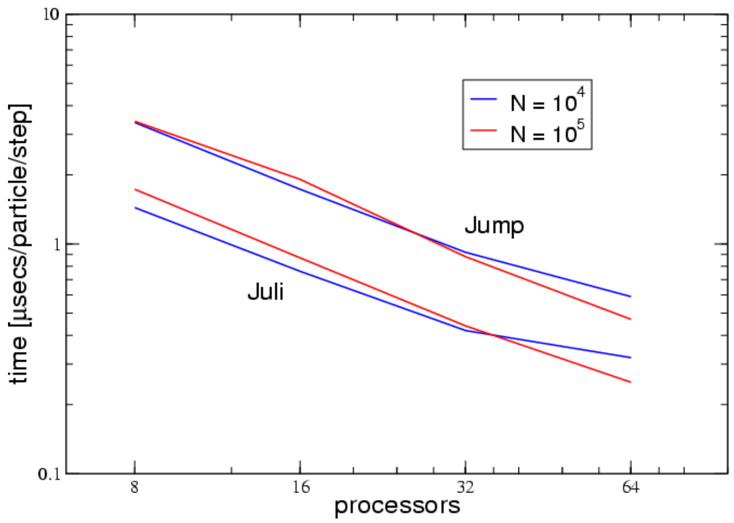


Figure 11: *Comparison between execution times of the force-decomposition algorithm on Jump and JULI for two different problem sizes. For small systems, the ratio between communication and computation becomes larger, so that the scaling gets worse. On 8 processors the performance ratio is 2.34 and 1.97 for the small and large system respectively.*

3.11.4 Performance Results of QCD Code (Stefan Krieg)

Figure 12 displays measurement results for the strong and weak scaling behaviour of a Quantum Chromodynamics (QCD) computational kernel. The measurements for strong scaling have been carried out on a global grid with $32^3 \cdot 64$ grid points. In this case, the fixed workload is distributed among an increasing number of processors. The plot "strong absolute" (Figure 12, Chart 1) shows the achieved total performance in MFLOPS over the number of processors. The given reference lines are related to ideal scaling with the performance achieved on one processor or one node, respectively. The plot "strong relative" (Figure 12, Chart 2) shows the performance in MFLOPS per processor over the number of processors used. The measurements on weak scaling were done with a fixed per-processor problem size on a local grid of 4^4 grid points. Apart from this, the measurements are analogous to strong scaling.

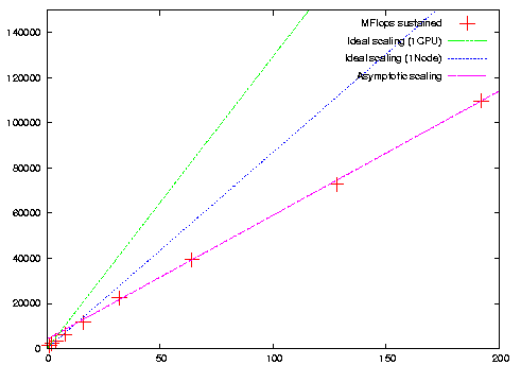


Chart 1: *strong absolute*

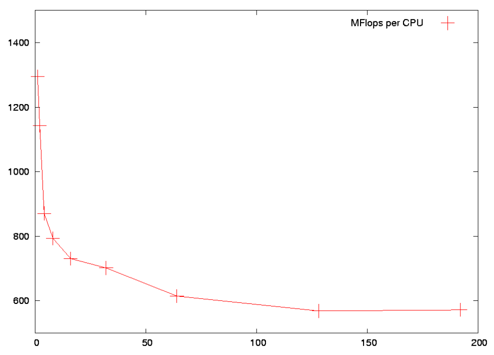


Chart 2: *strong relative*

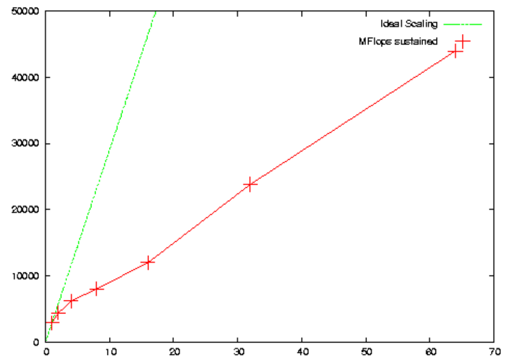


Chart 3: *weak absolute*

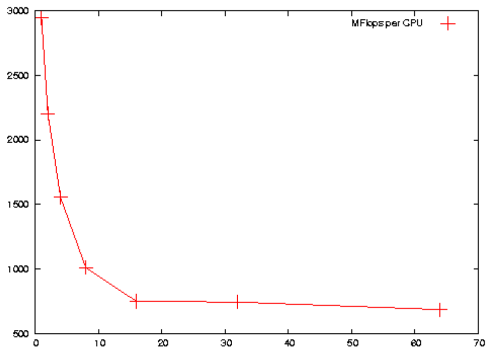


Chart 4: *weak relative*

Figure 12: *QCD strong and weak scaling*

3.11.5 ***Quantum Computer Simulations*** *(Guido Arnold, Marcus Richter, Binh Trieu, Thomas Lippert)*

Abstract

Operational imperfections and decoherence errors are approximately modelled at gate level as an extension of our massively parallel quantum computer simulator. Decomposing a universal set of basic gates into plane rotations and phase shifts allows to introduce gaussian distributed angle- and phase errors as effective imperfections of steering pulses acting on a qubit in a physical system. Combined with a simple decoherence model we investigate the impact and the interplay of operational and decoherence errors. We analyze the robustness of basic quantum operations and several quantum circuits such as quantum Fourier transformation and Grovers search algorithm. We find out very different sensitivities and describe their dependency on the system size. We provide a graphical tool that allows to investigate the resultant error patterns of our large scale simulations in detail.

Introduction

As a first step towards the simulation of realistic many qubit quantum computer devices we implemented a massively parallel gate level simulator which allows to simulate quantum systems up to 37 qubits requiring 3TB of memory on high end systems like IBM Regatta p690+. The simulation of quantum computers is clearly memory bounded. Highly optimized memory access and communication patterns allow for efficient simulation using up to 1024 processors. Within the idealized framework of gate level quantum computer simulation it is possible to approximately model the impact of gate imperfections and decoherence by introducing a simple error model. We used the Jülich Linux Cluster (JULI) to perform stochastic simulations on system sizes up to 32 qubits. Due to the higher aggregate bandwidth these simulations run up to 20 percent faster than on the IBM Regatta system *Jump*.

Error Model

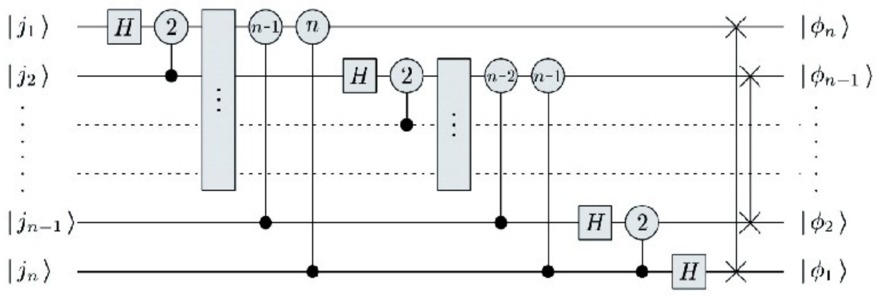
To implement a basic model of operational errors every single qubit gate can be generated from *plane rotations* $R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ and *phase shifts* $P(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$. This decomposition allows to introduce gaussian distributed angle- and phase errors ε with standard deviation σ, such that $R_\varepsilon(\theta) = R(\theta + \varepsilon)$ and $P_\varepsilon(\phi) = P(\phi + \varepsilon)$ respectively. Computation of controlled two- and more qubit gates can be reduced to effective single qubit gate computation acting only on that part of the state vector whose control-bit(s) are set to $|1\rangle$. A simple decoherence error model (depolarizing channel) allows for a *bit-flip* $\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, a *phase-flip* $\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ or *both* $-i\sigma_y = \sigma_x\sigma_z = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$ with probability $p/3$ each. The state vector remains unchanged with probability $1-p$. We assume an approximately constant operation time for every single gate independent of the type and the qubit it operates on. After each serial operation within the quantum circuit *each* of the n qubits (stochastically independent) can be subject to one of the depolarizing operators σ_α. For this we use n independent random sequences each containing m uniformly distributed numbers, where m is the size of the ensemble (=number of experiment repetitions).

In different experiments we study the effects of gate imperfections and decoherence depending on the standard deviation σ and the probability p. Given a certain confidence level we want to find out numerically thresholds for these parameters in real applications such as Quantum Fourier transformation or Grover's search algorithm. We compute the errornorm $e^2(\sigma, p) = |\psi - \psi_{corr}|^2$ with $0 \le e^2 \le 2$ since ψ is normalized to 1.

These results are to be compared with future calculations from realistic (=dynamic) simulations of quantum computer devices, taking into account the full time evolution according to a time dependent Hamiltonian discribing both, the system and the environment.

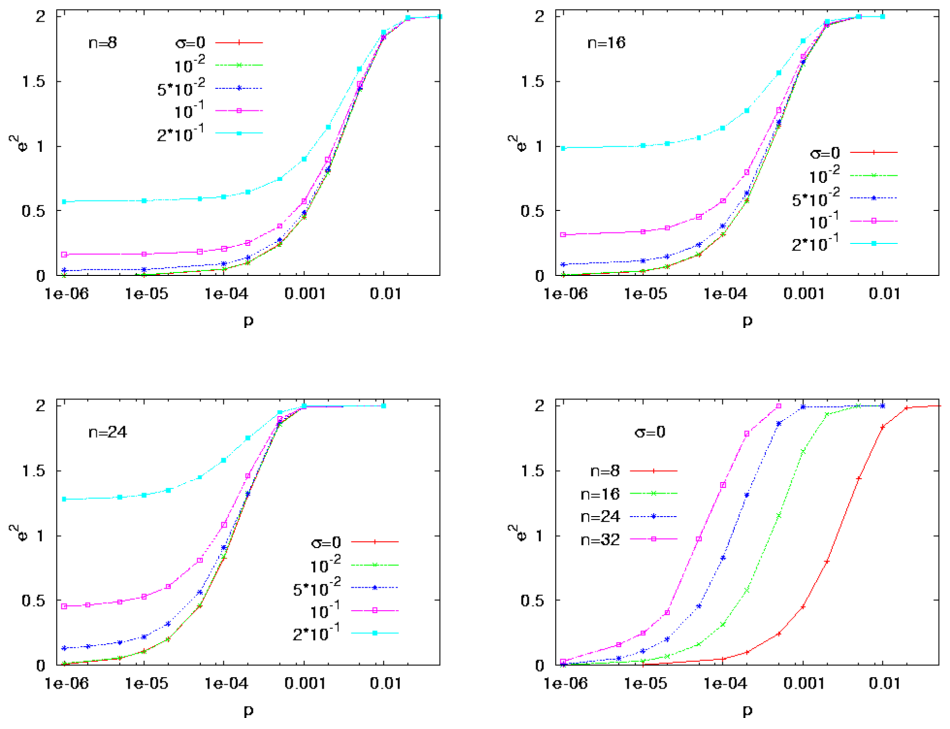
Quantum Fourier Transformation

We want to test the robustness of the Quantum Fourier Transformation (QFT) circuit since it is the quantum kernel of Shor's factorization algorithm. The QFT can be realized by the following circuit using $n = \log_2 N$ qubits:



Erroneous Hadamard operations on qubit q are implemented as $H_\varepsilon^{(q)} = R_{\varepsilon_1}(\pi/4)P_{\varepsilon_2}(\pi)$. The controlled phaseshifts are implemented as conditional (effective) single qubit phaseshift operations $P_\varepsilon(\phi)$ with $\phi = 2\pi/2^k$. To realize the reversal of the qubit order at the end of the circuit we replace each swap operation by a sequence of 3 controlled *NOT* operations: $SWAP(q_1 \leftrightarrow q_2) = CNOT(q_1,q_2)\, CNOT(q_2,q_1)\, CNOT(q_1,q_2)$. Similar to the Controlled Phaseshift these controlled *NOT* operations are implemented as effective *NOT* operations which can be decomposed to allow for operational deviations: $NOT_\varepsilon = R_{\varepsilon_1}(\pi/2)P_{\varepsilon_2}(\pi)$.

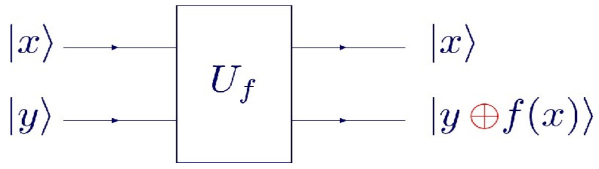
To analyze the error robustness of the QFT cricuit we investigate the errornorm and the qubit vectors in dependence of (σ, p) for system sizes $n = 8, 16$ with 100000 repetitions and $n = 24$ with 10000 repetitions per experiment.



The plots suggest a critical behaviour of the system depending in σ. The system is very robust against operational errors, since we find identical curves of $e^2(\sigma, p)$ for all $\sigma \leq 10^{-2}$ even for the largest system investigated. Larger operational errors increase the errornorm the more the larger the system is. To quantify the dependency of decoherence errors on the system size we have added the simulation results on a 32-qubit system.

Grover's Search Algorithm

In order to search an element in an unstructured database of $N = 2^n$ entries we suppose to have an oracle function $f_k : \{0,1\}^n \to \{0,1\}$ given as $f_k(x) = \delta_{kx}$ marking the searched element at position k .



Using the *f-controlled NOT* gate as depicted with an ancillary qubit 0 preset to $|y\rangle = |0\rangle - |1\rangle$ we can realize Grover's quantum search algorithm as:

- initialize

  $|x\rangle \leftarrow H|0\rangle = \frac{1}{\sqrt{N}} \sum_{j=1}^{N} |j\rangle$

  $|y\rangle \leftarrow H^{(0)} \sigma_x |0\rangle$

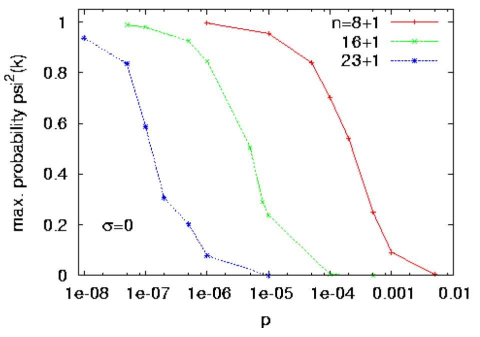
- repeat until $l \approx round\left(\frac{\pi}{4}\sqrt{N} - \frac{1}{2}\right)$:

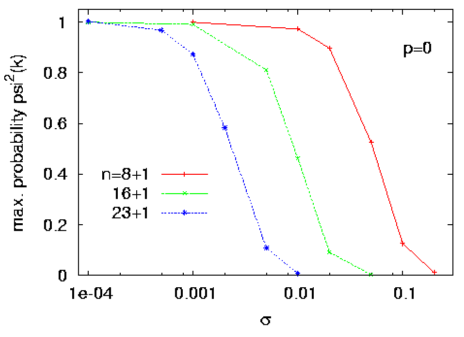
  $|\psi\rangle \leftarrow Q|\psi\rangle = -HU_{f_0} HU_{f_k} |\psi\rangle$

  $l \quad \leftarrow l + 1$

H is realized as a sequence of single qubit Hadamard gates $H_\varepsilon^{(q)}, q = 1, \ldots, n$ whereas U_f is implemented as a generalisation of the *CNOT* and Toffoli gate in our simulator. Here the decomposed NOT_ε acts only on those two components of the $n+1$ qubit state vector that encode k in binary representation.

We simulate system sizes of 8+1, 16+1 and 23+1 qubits demonstrating the effect of operational inaccuracies and decoherence errors on the amplitude $\psi(k)$ of the database element we are searching for, expecting its (first) undisturbed maximum after $l_{\max} = 12, 201, 2274$ Grover iterations respectively. In case of nonvanishing decoherence we can see damping of the value of the amplitude $|\psi_{\max}(p, \sigma = 0)| < |\psi_{\max,corr}| \approx \sqrt{1 - 1/N}$. The superposed decoherence process growing with increasing l leads to a maximum shifted towards $l < l_{\max}$. In case of operational errors we state a more robust behaviour but switching to an appropriate deviation level we also see a clear shift and damping of the maximal amplitude. In contrast to the QFT algorithm the σ -threshold is very sensitive to the system size. We plot the maximal probability of finding the correct amplitude.





Outlook

We have analyzed the impact of gate imperfections and decoherence errors within the idealized framework of a quantum computer simulator at gate level. We have seen that the quantum Fourier circuit is more robust to operational inaccuracies than Grovers algorithm on comparable system sizes. We can quantify the dependence of this sensitivity to both error sources on the system size. Combining decoherence and operational errors we cannot see any deviation from additive errorsums, which means that our results are compatible with non-correlation of the two error sources. Our results are to be compared with future calculations from realistic (=dynamic) simulations of quantum computer devices, taking into account the full time evolution according to a time dependent Hamiltonian discribing both, the system and the environment.

In a next step we will use our massively parallel quantum computer simulator to numerically investigate the characteristics of different error correction schemes.

4 Conclusion

In a rather short period, the JULI cluster developed from a prototype system into a fully functional, reliable high-performance cluster suitable for production. Practically all administrative and management functions required for day-to-day operation are available and extensive tests with real-world application codes have shown that the system meets the expectations with respect to performance, stability and scalability. The high-speed InfiniPath network contributes to system scalability in that it eliminates well-known shortcomings of native InfiniBand implementations by using a low-level communication protocol scaling well with the number of communication nodes. The GPFS parallel file system has proven to be scalable on many other, much larger systems already and proved to be stable and reliable on the JULI cluster also. It works well as the default file system for all user's home directories, even though, some work still needs to be done with respect to performance tuning, at least for the rather small JULI configuration. In comparison to the Power 4+ based p690 cluster *Jump,* installed at FZ Jülich for more than three years, JULI performed very well in most benchmarks. The sustained performance per CPU on JULI was up to twice that of *Jump* for realistic applications, and, measured in GFLOPS per floor scpace, the comparison is even more impressive: 32 Power4+ CPUs on *Jump* deliver a peak performance of 218 GFLOPS per p690 frame, whereas 224 CPUs in one JULI compute rack render 2240 GFLOPS peak on roughly the same floor space. According to Linpack sustained performance, one JULI rack compares to approximately 11 frames of *Jump*. It should be mentioned that the multi-core architecture of the PPC 970 may impose a significant load onto the memory subsystem, depending on the characteristics of the application code. Thus, a high computation/load-store ratio is key to good application performance.

During the course of the project, the replacement of 400 MHz memory modules by 533 MHz DIMMS proved to be very beneficial.

In summary, the JULI project has shown that it is very well possible to build up a highly performing compute cluster based on PowerPC 970 processors and InfiniPath interconnect. The deployed system is ready for production and will continue to be used for this purpose after the completion of this project. The developed cluster software is well suited for all management, monitoring and administrative tasks and shows good usability, stability and scalability. The same is true for the hardware of the JULI cluster, which ran stable for the duration of the project and gave the feeling of a robust system. Some important RAS functions, like the mirroring of system disks, could not be tested during the project due to the prototype character of the used firmware. It it assumed, however, that this functionality will be fully available with future firmware versions.

Acknowledgments

Grateful acknowledgment is made to the large number of people who helped in planning and developing the JULI cluster, in doing investigations, tests and measurements, in finding bugs and solutions and who contributed in many other ways to this project:

FZ Jülich: Guido Arnold, Ulla Ehrhart, Wolfgang Frings, Inge Gutheil, Wolfgang Gürich, Willi Homberg, Stefan Krieg, Olaf Mextorf, Bernd Mohr, Birgit Naun, Franz Josef Schoenebeck, Godehard Sutmann, Felix Wolf, Klaus Wolkersdorfer, Lothar Wollschläger, Brian Wylie

IBM: Torsten Bloth, Hans Böttiger, Luigi Brochard, Kalyan Gunda, Michael Hennecke, Thomas Hesmer, Peter Juerss, Gerd Kaufholz, Karsten Kutzer, Rudolf Land, Christoph Pospiech, Kevin Reilly, Ingolf Salm, Heiko Schick, Otto Wohlmuth

ParTec: Ferdinand Geier, Jens Hauke, Ralph Krotz

QLogic: John Gregor, Greg Lindahl, Dave Olson, Betsy Zeller

Voltaire: Nir Gal

Above all, the authors wish to express their sincere thanks to the initiators of this project, Prof. Dr. Dr. Thomas Lippert (FZ Jülich), Nurcan Rasig (IBM), Art Goldberg (QLogic), Scott Metcalf (QLogic), Hugo R. Falter (ParTec), Reinhold Wagner (FZ Jülich)

References

[1] F. Wolf, B. Mohr: "Automatic performance analysis of hybrid MPI/OpenMP applications", Journal of Systems Architecture, Special Issue "Evolutions in parallel distributed and network-based processing", Volume 49, Issues 10-11, Pages 421-439, November 2003.

[2] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr: "Scalable Parallel Trace-Based Performance Analysis", Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (EuroMPI/PVM 2006), Springer LNCS 4192, Bonn, Germany, Pages 303-312, September 18-20, 2006.

[3] http://www.fz-juelich.de/zam/JULI/

[4] http://www.bsc.es/

[5] http://www.fz-juelich.de/zam/llview/

[6] Larry McVoy and Carl Staelin: “lmbench: Portable tools for performance analysis“, Proceedings Winter 1996 USENIX, San Diego, CA, pp. 279-284.

[7] http://www.bitmover.com/lmbench/

[8] Norbert Eicker and Thomas Lippert: “Low-level Benchmarking of a Novel Cluster Architecture”, to appear in “Tagungsband des KiCC’07 Workshop, Kommunikation in Clusterrechnern und Clusterverbundsystemen”, TU Chemnitz, February 8, 2007.

[9] http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm

[10] http://www.altair.com/software/pbspro.htm

[11] Jack Dongarra: "Performance of Various Computers Using Standard Linear Equations Software", University of Tennessee, Knoxville TN, 37996, Computer Science Technical Report Number CS - 89 - 85, February 18, 2007, url:http://www.netlib.org/benchmark/performance.ps.

[12] IBM Cluster Systems Management for AIX 5L and Linux: Planning and Installation Guide Version 1.5.0 (SA23-1344-01).

[13] IBM General Parallel File System: Concepts, Planning, and Installation Guide Version 3.1 (GA76-0413-00).

[14] IMB Tivoli Workload Scheduler LoadLeveler V3.4: Installation Guide (GI10-0763-03).

[15] IBM XL Fortran Advanced Edition V10.1 for Linux: Installation Guide (GC09-8020-01).

[16] IBM XL C/C++ Advanced Edition V8.0 for Linux: Installation Guide (GC09-8017-01).

[17] PowerPC 970MP Microprocessor: http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_970MP_Microprocessor

[18] Qlogic InfiniPath Install Guide Version 2.0 (IB0056101-00 C).

[19] Qlogic InfiniPath User Guide Version 2.0 (IB6054601-00 C).