Kernel Performance on QCDOC

Joy Khoriaty

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2005

Authorship declaration

I, Joy Khoriaty, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Signed:

Date:

Matriculation no:

Abstract

The QCDOC system is a high end resource for computational Quantum Chromodynamics available today. This powerful system is purpose built for solving QCD problems. It is interesting to examine whether one could take advantage of such a system for other types of scientific codes.

In this thesis we present results from our investigation on porting computational kernels onto the QCDOC system, highlighting the issues involved in the porting process which include understanding the specifics of the system such as the communications network and the memory subsystems.

Both computational kernels considered mimic the behavior of scientific applications. We achieve good results for our Jacobi kernel with nearest neighbor interaction which maps directly onto the system network. The Fast Fourier Transform kernel shows good scalability of the serial part of the code, but our conveniently written collective communications need improvement.

This work presents a first step in understanding both the QCDOC system as well as the porting issues related to it. Future work can take advantage of our results to accelerate exploiting this powerful QCD resource.

Acknowledgements

It seems that it is only when we reach the end of the work that we learn what ideally we should have started with.

I would like to thank Dr. Joachim Hein for his patience, guidance and support throughout this project.

Acknowledgments also go to Dr. Peter Boyle, Dr. Balint Joó and Dr. Michael Creutz for sharing their insight into the QCDOC system and parallel architectures.

Additional thanks goes to Sean Murphy for useful suggestions and enjoyable discussions.

Thanks goes to Laure Sayyed Kassem for her firm support throughout my MSc year.

Thanks to my family, Noha, Joseph and Jad for making it all possible.

Contents

List of Tables

List of Figures

1. Introduction

The QCDOC (Quantum Chromodynamics on a Chip) architecture is a special purpose supercomputer with more than 12000 processors and providing a peak performance of over 10 TFlop/s. At the time of writing it is the largest special purpose built system in the UK. [1]

QCDOC is optimized for the needs of lattice QCD. The memory subsystem, communication interconnects, system software, and programming libraries are all designed to strictly satisfy that goal. Any additional feature not needed for lattice QCD that would make the machine more general purpose has been left out to keep the project costs to a minimum.

We investigate the issues encountered when porting computational kernels that mimic the behavior of a wide class of scientific applications onto QCDOC. Now that the system is available, this allows us to see whether applications QCDOC was not initially designed for can use the system. The process of porting involves understanding and dealing with any software and hardware issues encountered along the way.

Another aspect of this work involves benchmarking the ported kernels against other High Performance Computing (HPC) systems available at the Edinburgh Parallel Computing Centre (EPCC); they include a p690 cluster (HPCx) [3] and a BlueGene/L system (Bluesky) [4].

Our work is related to the work of Michael Creutz at the Particle Theory Group at Brookhaven National Laboratory (BNL) who wrote a minimal port of MPI to QCDOC using his own memory copy routine [2]. His works initially began on QCDSP (QCD on a Digital Signal Processor), a predecessor of QCDOC.

The reader is expected to be familiar with the Message Passing Interface (MPI) [5]. All supplied source code and programs are written in C/C++. `Courier New` font is used to indicate source code snippets and the names of subroutines.

In section 2 we introduce some background information on QCD. This should give the reader an idea of the type of problems QCDOC was designed to solve. We then present the QCDOC hardware architecture, the software architecture, the approaches to programming the system, and its message passing libraries.

Section 3 presents the different computational kernels that will be examined. The first kernel is a Jacobi iteration type code. The second kernel is a two-dimensional fast Fourier transform (FFT). Section 3 also presents our work with collective data movement routines. Lastly we present the issues that we faced in the porting process.

Section 4 presents our results and analysis. A performance comparison is done against ports onto the different HPC platforms available at EPCC for the image analysis and the FFT kernels. Different collective communication routines based on varying

message passing libraries are also benchmarked.

Section 5 summarizes our work, its limitations, and suggests future directions of research.

2. Background

2.1. Quantum Chromodynamics

QCD is a theory which describes one of the fundamental forces of nature called the strong interaction. It was proposed in the early 1980s by David Politzer, Frank Wilczek and David Gross as a theory to understand the structure of protons, neutrons and other particles. It uses Quantum Field Theory (QFT) to describe the interactions of quarks and gluons which are considered today as elementary particles. QCD is an important contribution to the Standard Model of elementary particle Physics.

Investigating QCD for low momenta as typical for Hadrons such as Protons and Neutrons using analytic techniques in continuous space-time proves difficult. This is why Lattice QCD as a discretisation of QCD onto a lattice mesh is used instead [6]. By formulating QCD on a space time lattice this opens the problem to be tackled through simulations using modern computers.

A numerical method primarily used in lattice QCD is the Hybrid Monte Carlo algorithm to perform the Feynman path integral in a space-time discretised within a four-dimensional Cartesian lattice or Cartesian grid. The input parameters consist of quark masses and the strength of the coupling constant [7].

The nature of the problem involved in lattice QCD suggests that a minimal set of features are required for a special purpose QCD machine. These include efficient nearest-neighbor communications, a minimal set of collective communications, modest input/output (I/O) requirements, and as much computational throughput as possible.

2.2. The QCDOC System

The QCDOC system was developed by a collaboration of Columbia University, UKQCD, the RIKEN-BNL Research Centre and IBM. It builds on the work of the QCDSP supercomputer [8], a four-dimensional mesh machine built in 1998 which won the Gordon Bell Prize award that year for best price/performance machine. QCDSP proved that a massively parallel processor could be built to provide low latency, high performance, low power consumption, and high integration.

The design goals of QCDOC are the same as its predecessor, the main difference being greater transistor density and clock speeds, as well as a higher bandwidth, lower latency, and a more flexible communications network. QCDOC achieves an equally important price/performance ratio as part of its design goal by having 1$ per sustained Mflops. This is done by having processors clocked at around 450MHz with 2flops per cycle at half of their efficiency costing around $400 each [X] (1).

$$\frac{\$400}{2\,flops/cycle \cdot 450MHz \cdot 0.50eff} = \$0.89/Mflops \qquad (1)$$

It is in this spirit of high performance/price ratio of a single processor coupled with a network that allows high scalability that the QCDOC supercomputer was designed.

2.2.1. Hardware Architecture

The QCDOC massively parallel computer is based on an Application Specific Integrated Circuit (ASIC) which uses IBM System-on-a-Chip (SoC) technology. The ASIC allows a complete system with processor, memory, and network controllers to be integrated into a single chip.

The main component of the QCDOC ASIC (figure 1) is an embedded IBM PowerPC 440 CPU core. This is a 32-bit processing core that does not handle floating point operations, crucial for scientific number crunching. To achieve that, a 64-bit IEEE floating point unit (FPU) is added which can supply 0.8 GFlops.

An embedded DRAM chip (EDRAM) of 4 MB is connected to the cores through a high speed bus of 8GB/s bandwidth, in addition to a 2.6 GB/s interface to the external memory which is of 128MB on the UK QCDOC system.

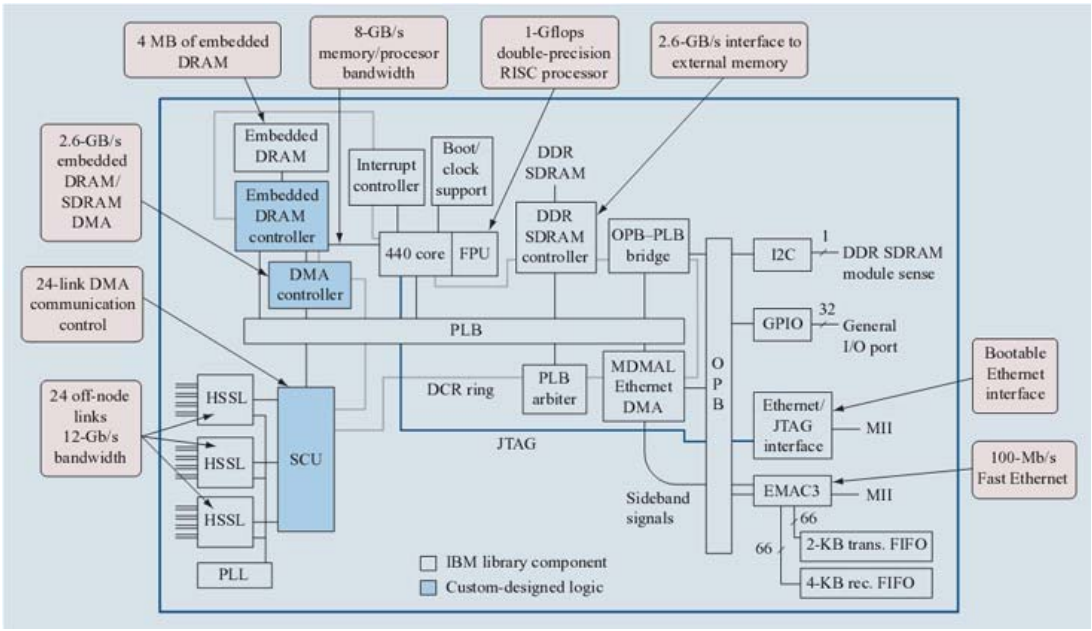

Figure 1 QCDOC ASIC – a System on a Chip [1]

Inter-node communications on QCDOC are handled through high-speed serial links at 12 Gb/s of bandwidth. Additionally an Ethernet interface allows booting the nodes as well as diagnostics and I/O.

The ASIC is power efficient and has a small footprint. It consumes 1-2 Watts of power and the size of a processor die is at 1 cm per side. This gives the system a very small electrical and floor footprint.
The system packaging allows for high density and integration. Two ASICs or nodes are packaged into a daughterboard (figure 2) with an associated dual inline memory

module (DIMM) socket for each node. 32 daughterboards (figure 3) make up a motherboard, 4 motherboards are inserted in a backplane (figure 4), and two backplanes make up a crate, and two crates make up a rack to make up a complete system (figure 5).

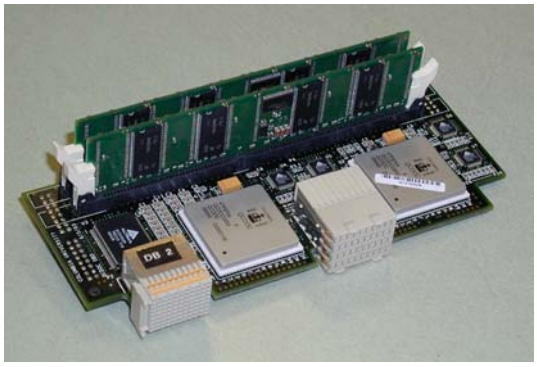

Figure 2 A QCDOC daughterboard with two ASICs and DDR DIMMs [1]

Figure 3 A QCDOC motherboard mounted with 32 daughterboards [1]

The system consists of three networks. The physics network consists of a six-dimensional bit-serial nearest-neighbor mesh. This is the high performance network onto which QCD problems map well. It is a very low latency network of the order of a fraction of a microsecond, and offers high bandwidth connections to the neighboring nodes. The two dimensions above the fourth dimensions offer flexibility in repartitioning the machine using space filling curves to reduce dimensionality (ref topology). This is an improvement on the QCDSP system where for each varying partition configuration the system had to be re-wired by hand.

The control network is a 100 MB/s Ethernet tree used to boot the compute node kernels, load programs and perform I/O. The global interrupt network is used for initial synchronisation of the system.

The machine is accessed via a front end machine on which jobs are compiled and submitted to QCDOC. I/O on QCDOC is also handled by an external dedicated file server.

2.2.2. Software Architecture

Three main layers make up the QCDOC software architecture [1]. Essentially these layers allow for writing and building programs for the compute nodes on the front end, the interaction between the front-end with the backend nodes and the execution of the compiled programs on the backend nodes with the results outputs redirected accordingly.

The first layer runs on the front end machine and allows interaction with the compute nodes. This involves booting the backend nodes via the JTAG Ethernet interface and loading a run time kernel onto the systems. Managing and controlling the backend nodes is then done via a multithreaded program called the `qdaemon` and an adapted shell called the `qcsh`.

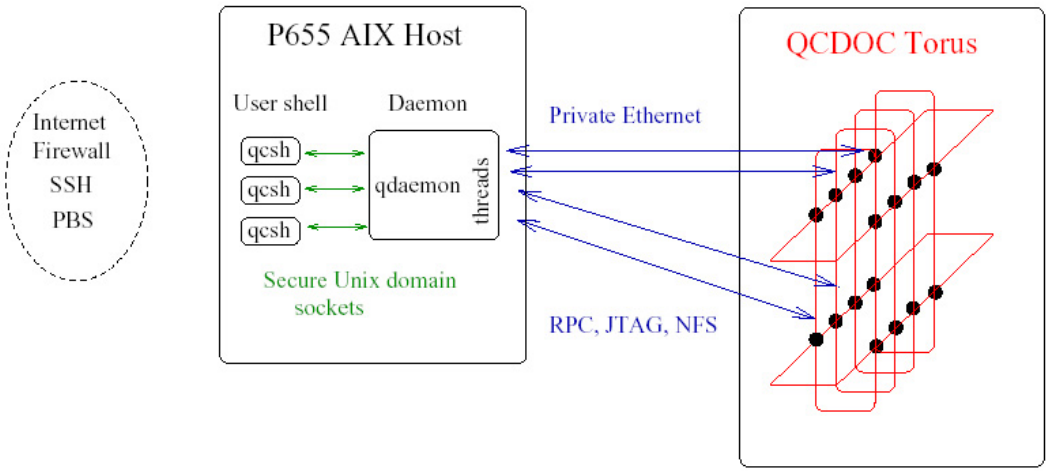

Figure 4 QCDOC Frontend and Backend Interaction

The second component is the computational node run kernel. This is a lightweight operating system which offers basic services such as loading programs into memory, accessing the on-chip network devices and servicing system calls and I/O. The lightweight kernel runs two threads only, one for the kernel itself, and the other for the running application. It does no job scheduling or swapping so that the compute node resources are entirely dedicated to the executing program.

The third software layer is the user environment. It offers a familiar and standard environment for users to edit, compile, debug, and submit their codes to the backend. The GNU C++ and IBM xlC are offered as cross-compilers to the QCDOC compute nodes. Two message passing libraries allow the user to describe the exchange of messages between the compute nodes: the serial Communications Unit (SCU) and the QCD Message Passing Interface (QMP). SCU provides Direct Memory Access (DMA) commands to send and receive data from a compute node to the other, while QMP offers a higher level and more general interface to messaging passing which includes collective communications regularly used in QCD applications.

3. Working on QCDOC

3.1. Submitting Jobs

The process of submitting jobs to QCDOC is directly linked to understanding the machine hardware configuration required for the application to be run. Documentation for using the system is available through various online short manuals [12] [13] [14] which are in no way all-inclusive, but rather scarce. Because of main conceptual similarities between QCDSP and QCDOC, an authoritative guide on QCDSP [11] offers a lot of additional insight. In this section we try to briefly round-up and complement the information representative of working with the machine from both what is available online an our own experience with the system.

We look at the steps involved in a typical job submission and then explain the fundamental concept of mapping the physically allocated partition to the desired logical partition.

Running a job on QCDOC is a straightforward procedure. The first step consists of reserving a partition on the system, where a partition is a six-dimensional subset of the available compute nodes on the system. Partitioning allows more than one job to run at a time. Typically small development partitions of up to 256 nodes are available for code development, whereas bigger partitions of 1024, 2048 and 4096 partitions are reserved for production QCD runs.

Number of Nodes	Allocation Limits	Typical Naming
8/64/256 node machines	Time limited	dev/slot0-3 rack32/crate0/slot0
64 node machines	Time unlimited	acc3/slot0 status
1024/2048/4096 “monsters”	Time unlimited	rack33

Table 1UK-QCDOC Partition Types and Limits

No batch or queuing system is available on QCDOC. Currently a web-based partition reservation system (figure 5) is used where a user flags a partition as reserved for a certain amount of time to run jobs. The allocated partition should be released when done so that it can be allocated by another user. This process works well because of the relatively small and tightly integrated QCD community but it would face problems with a larger number of users. The allocations are sometimes reinforced with a time limit after which the partition is released and marked available automatically; this concerns smaller partitions as detailed in (table 1).

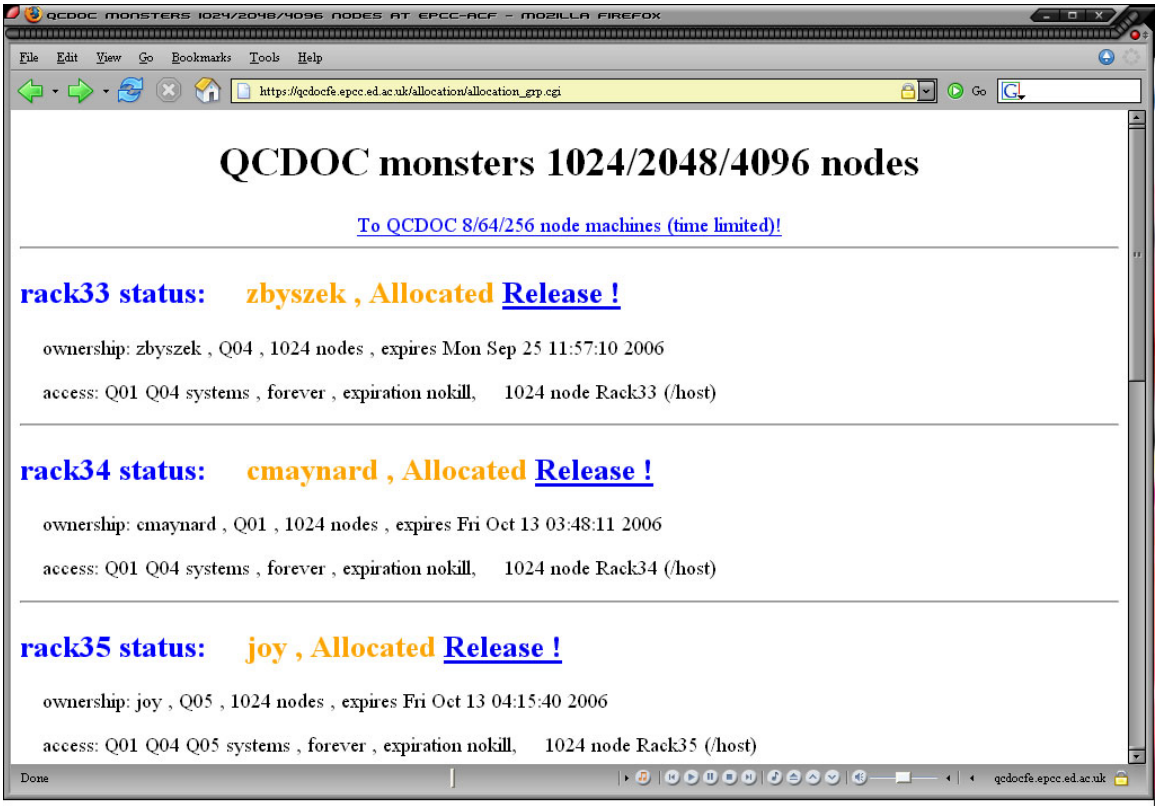

Figure 5 Web allocation interface

QCDOC is accessible through its front end machine system, via secure shell connections (SSH). The front end is typically a symmetric multiprocessing (SMP) server which can handle high loads of users in a reliable way. The user connects to the front end system to edit, compile, debug, and submit their codes to the backend nodes.

Once logged in, the user should use the most up-to-date QCDOC Operating System (QOS), by sourcing the appropriate directory

```
source /qcdoc/sfw/qos/v2.6.0/aix5.2g/scripts/setup.sh
```

This sets up various environment variables and paths for compiling. The user now has access to variety of commands (table 2) to that are used to configure and control the machine.

We will explain these commands in the remainder of this section by going through a typical job submission run that consists of connecting to the machine, setting up the partitions, compiling a program, submitting it to the backend, and finally detaching for the partition.

`qsession $QMACHINE`	Script to start qdaemon and qcsh
`qinit $QMACHINE`	Connect a qcsh session to a qdaemon
`qpartition_connect`	Connect to a partition
`qreset_boot`	Resets and boots a partition
`qdiscover`	Find the topology of a partition
`qpartition_remap`	Map a machine topology

qrun	Run a program
qnodes_print	Print nodes information
qdetach	Disconnect from a partition
qkill	Kill a running user program
qhelp	Displays help information

Table 2 Commonly Used Machine Configuration Commands

The user can now bring up a partition reserved online by connecting to it using its unique name. The name is of the type /dev/slot0-3 or /rack32/crate0/slot0. For our explanation we use the environment variable name $QMACHINE to represent the machine's unique name. The user then loads the desired run time kernel onto the compute nodes and initializes them by running

```
$ qsession $QMACHINE
$ qinit $QMACHINE
$ qpartition_connect –p 0
```

Where the `qsession` script starts `qdaemon` and `qcsh` and sets the $QMACHINE variable to the qsession argument. `qinit` starts up communications with the qdaemon for on the allocated partition. `qpartition_connect` establishes a connection with the machine. This typical sequence of commands can be place into a `.qcshoc` file so that they get executed automatically once a `qsession $QMACHINE` command is called.

The user is now connected to his partition and should define the mapping between the physical machine and the desired logical machine

```
$qdiscover
$qpartition_remap –X45 –Y0123
```

Where `qdiscover` discovers the physical machine topology by counting how many nodes exist in each of the six physical machine dimensions. And `qpartition_remap` allows changing the mapping between the machine and application directions. We present the concept of mapping between the physical machine and the application in further detail in section 3.2. The machine setup is now complete with the network communications up, the application axes mapped to the physical axes, and all the nodes run kernels ready.

The final step involves loading the appropriately cross compiled program onto the processing nodes.

```
$powerpc-gnu-elf-g++ myprogram.C –o myprogram
$qrun myprogram
```

`powerpc-gnu-elf-g++` is the cross compiler which takes the `myprogram.C` C++ program source code as an argument and returns a PowerPC compiled executable called `myprogram`.

```
$ qrun
```

`qrun` then loads the program into each compute node and runs it. The output from node 0 or the root processor will be echoed to the users' shell or `qcsh` session only. The outputs from the other nodes can be read by using the `qnode_print` command.

Running or hanging jobs can be signaled to stop by hitting CTRL-C from the shell. The `qkill` program can then be called to make sure that no run-away processes are still running.

Once a job is completed one can detach from the machine partition using `qdetach` or simply by exiting `qcsh`. Lastly the user should release a time unlimited partition if it is no longer in use so that other users can reserve it.

3.2. Machine Topologies

Once a partition is allocated as described above, its topological configuration can be discovered with the `qdiscover` command (figure 6). This shows how many nodes are present in each of the six physical dimensions available.

```
QD:Partition::DiscoverTopology.M> Dimension 0 has length 1 nodes
QD:Partition::DiscoverTopology.M> Dimension 1 has length 1 nodes
QD:Partition::DiscoverTopology.M> Dimension 2 has length 1 nodes
QD:Partition::DiscoverTopology.M> Dimension 3 has length 2 nodes
QD:Partition::DiscoverTopology.M> Dimension 4 has length 2 nodes
QD:Partition::DiscoverTopology.M> Dimension 5 has length 2 nodes
```

Figure 6 Output of qdiscover on an 8 processor partition

Consider the six physical dimensions available as numbered 0 through 5 and the six logical or application dimensions referred to as T, X, Y, Z, S and W respectively. Each node has a mapping between the application dimensions and the machine dimensions (figure 7). This way, the topology and dimensionality seen by the application code can be changed by remapping the machine and application directions as many times as needed. This is done by consecutively folding the machines axes together into a single application axis until the desired topology is reached.

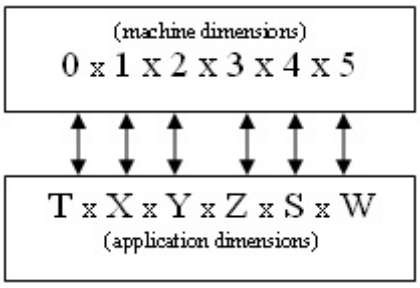

Figure 7 Mapping between machine and application dimensions

For example (figure 8), we have the configuration for 2 processors in the Z dimension, 2 processors in the S dimension and 2 processors in the W dimension. Given the available machine space, one can setup the required application space or logical machine by using the `qpartition_remap` command to map the allocated machine

to the logical one.

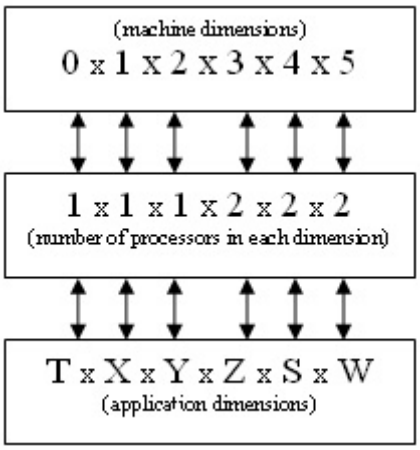

Figure 8 Mapping an 8 processor six dimensional partition to a 4x2 logical partition

In order to have a 4x2 two-dimensional logical machine, we would use the command

```
$ qpartition_remap -X45 -Y0123
```

This would map the physical dimensions 4, and 5 to the X logical dimension making it of size 4 (2x2) processor dimension, and this would map the 0, 1, 2 and 3 physical dimensions to the Y logical dimension making it of size 2 (1x1x1x2).

In the arguments of `qpartition_remap,` each letter which represents a logical machine dimension is followed by one or more numbers that represent the physical dimensions. As you see, more than one physical dimension can be mapped or folded against the same logical dimension. This mapping of several hardware dimensions into a single logical dimension is done by using space filling curves. We can see in (figure 8), how a two-dimensional topology is folded into a one-dimensional one.

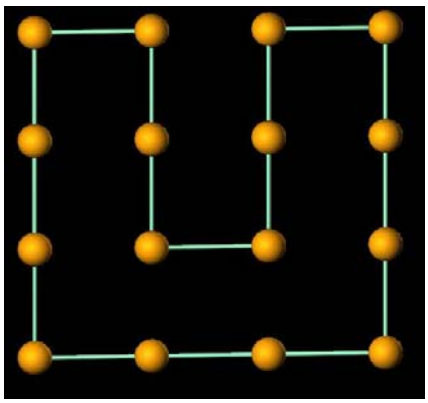

Figure 9 Folding from 2D to 1D [1]

This folding of machine dimensions into logical dimensions which is done by mapping more than one physical axis to the same logical axis is a great improvement in flexibility over the previous QCDSP system where a time consuming rewiring process was required for each configuration.

If we consider another example, running

```
qpartition_remap -T345 -X1 -Y2 -Z0
```

This will give a configuration for an 8x1x1x1x1x1 machine logical machine.

With the six-dimensional physical machine, we can therefore map the physical axes into dimensionalities ranging from one dimension to 6 dimensions, such as for example (figure 10) mapping folding a three dimensional machine into a one dimension. If any mappings are incorrect, `qpartition_remap` will notify the user that the mapping cannot be made.

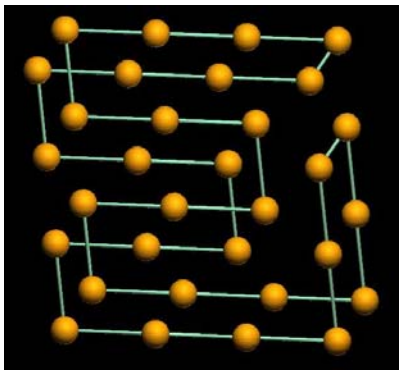

Figure 10 Folding from 3D to 1D [1]

3.3. Memory Subsystems

There are two accessible memory subsystems on QCDOC: a fast memory, and a slow memory. The fast memory uses the 4MB of EDRAM and the slower memory consists of the 128MB DDR-SDRAM. 96MB of which are reserved for dynamic allocation and around 16 MB for static allocation as of QOS 2.6.0. The system L1 cache is partitioned into 31KB of normal data + 1 KB of streaming data. As noted earlier, the bandwidth of the EDRAM is around 3 times that of the DDR-SDRAM.

In order to place data on the fast memory, we use the `qalloc()` routine by linking against the `qalloc.h` header file. It allows for the allocation of data on the fast and slow memories. The memory can in turn be freed by using the `qfree()` routine.

`qalloc()` allows the programmer to choose between the memory to be used by passing one of three flags (figure 11), `QCOMMS`, `QFAST`, and `QNONCACHE` which respectively represent the slow or DDR-SDRAM memory, the EDRAM and no caching.

```
void * qalloc (int flags, size_t bytes)
enum {
    QNONCACHE = 0x01,
    QOMMS = 0x02,
    QFAST = 0x04,
};
```

Figure 11 The three memory sections qalloc can reference

The standard `malloc` routine can also be used and the memory allocated will be placed on the DDR-SDRAM.

Taking advantage of the fast memory is crucial to obtaining better performance from the system, and we will be investigating this in our analysis and benchmarks. We will investigate the effects of using the different kinds of memory available in the benchmarks.

3.4. Message Passing

Two message passing libraries are available on QCDOC. The first one is know as the SCU, and the second one QMP. At the lower layer, calls to the SCU give access to send and receive functionality. These are asynchronous, allowing the overlapping of communication and computation. On top of the SCU layer is the QMP interface, a standard developed by the LQCD research community to provide fast nearest-neighbor messaging and some general communications routines. We focus on QMP for our work.

3.4.1. QMP

"The goal of QMP is to provide portable, low-latency, high-bandwidth communication routines suitable for Lattice QCD" [9] QMP is an application programming interface (API) optimized for the style of communication used in LQCD which consists of regular, repetitive communications between nearest neighbor nodes in an n-dimensional torus with periodic boundary conditions.

This application domain specific interface is implemented in three flavors. QMP-MPI implemented on top of MPI is used clusters; QMP-QCDOC uses the SCU and is designed for QCDOC and QMP-MVIA implemented is used on gigabit Ethernet or VIA (Virtual Interface Architecture) clusters. For our work, we only consider QMP-QCDOC and refer to it as QMP.

The basic capability requirements (figure 12) of QMP include the availability of a barrier call to synchronize all the partition nodes, sending contiguous messages to neighboring nodes along a specified axis and direction, and sending non-contiguous messages to a neighboring node where the message consists of a set of strided blocks.

QMP also offers a few collective communication routines such as broadcast, global summation, global max, global reductions. Additionally QMP offers allocated partition configuration functions.

- Point-to-point communication
- Non-blocking (computation and communication can be overlapped)
- Simultaneous, multi-directional transfers
- Chained block/strided transfers
- Separate routines for initialization and commencement of transfers, so that opened channels can be reused to minimize overheads for repeated transfers
- Global operations: global sum, maximum, minimum operations for integers, single and double precision numbers, and binary reductions, broadcast, barriers.
- Basic machine topology configuration and control

Figure 12 QMP Capabilities on QCDOC

QMP performance in latency is close to QCDOC native calls. This allows for small software overhead. Global operations are implemented using the store-and-forward capability of QCDOC. The implementation of QMP on QCDOC is complete but does not contain non-nearest neighbor communications.

We present a non-comprehensive list of the calls we used throughout our work. More details are available in the QMP standard [9].

A group of QMP calls are dedicated for initializing the message passing environment, specifying the machine layout, and terminating the work

```
QMP_init_msg_passing()
QMP_finalize_msg_passing()
QMP_get_number_of_nodes()
QMP_get_node_number()
```

These calls allow the discovery of the allocated machine configuration, the node number of the current node.

Additional calls allow the configuration of the logical layout of the machine (number of nodes in each direction) based on the constraints of the underlying allocated machine.

```
QMP_declare_logical_topology()
QMP_get_logical_dimensions()
QMP_get_logical_coordinagtes()
```

And optimally partition the lattice problem onto the logical machine

```
QMP_layout_grid()
```

Nearest Neighbor Communications can be declared through the following routines. These communications are intended to be highly repetitive in order to achieve high performance

```
QMP_allocate_memory()
QMP_delcare_strided_msgmem()
QMP_declare_msgmem()
QMP_free_msgmem()
```

The basic send/receive operations are done using the following calls

```
QMP_declare_receive_relative()
  QMP_declare_send_relative()
```

Where `QMP_declare_receive_from()` and `QMP_declare_send_to()` which are defined in the QMP standard are not implemented on QCDOC because they are non-nearest-neighbor.

The `QMP_declare_multiple()` function improves performance in initiating multiple sends by collapsing them into a single call.

Communications are started with the following calls

```
QMP_start()
QMP_wait()
QMP_wait_all()
```

Collective communications in QMP also include the following global operations for reduction of maximum/minimum and collective synchronisation of the processors

```
QMP_sum_int()
QMP_broadcast()
QMP_barrier()
```

3.4.3. QMP and MPI

It is natural to compare QMP with MPI, since MPI is a standard message passing interface used in HPC. MPI is not supported because the generality it offers is not needed for QCD specific codes. It is designed to offer a lightweight, efficient message passing system without implementing for example all of the semantics of MPI which are not needed in the context of LQCD.

The main differences with the QMP for QCDOC implementation and MPI are that there are no non-nearest neighbor communications. Also, MPI is more general. The generality of MPI is not needed for QCD since leaner libraries can run with higher performance. There could have been an implementation of MPI on QCDOC but what was wanted was an optimized messaging interface for LQCD only.

Despite that, some work has been done to port MPI to QCDOC by Michael Creutz [2]. His MPI library for QCDOC is not complete, but offers a minimal number of MPI calls (table 3). The library is built on top of the SCU, and the `qmemcpy` memory copy routine. They allow point to point communications to be performed at the price of synchronization of between all the nodes, which makes it a sub-optimal yet simple approach.

MPI_Init	MPI_Finalize
MPI_Comm_rank	MPI_Send
MPI_Comm_size	MPI_Isend
MPI_Get_processor_name	MPI_Irecv
MPI_Barrier	MPI_Issend
MPI_Abort	MPI_Attr_get
MPI_Reduce	MPI_Bcast
MPI_Wait	MPI_Allreduce
MPI_Recv	

Table 3 MPI Functions Offered by Creutz MPI library

We have added the following collective communication routines (table 4). Details of our basic implementation of these functions using the qmemcpy routine are described in (3.2).

MPI_Gather	MPI_Scatter
MPI_Allgather	MPI_Alltoall

Table 4 Collective Communications Added

3. Porting Computational Kernels

3.1. STREAM Benchmark

Many scientific applications rely on memory streaming applications, as we will see later on in the image processing code we study. STREAM is a well known, standard memory benchmark. It is ideal for studying memory performance in isolation.

We consider the STREAM benchmark in order to highlight the memory bandwidth on QCDOC and see how it compares with our other benchmark systems. Specifically, we expect the STREAM benchmark to show us what kind of performance gains we can obtain when using the fast EDRAM memory instead of the DDR-SDRAM.

The STREAM benchmark program measures the sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels [17]. The rule of thumb is to allocate an amount of memory so that each array is at least 4 times the size of the sum of all the last-level caches used in the run.

Routine	Kernel	Bandwidth (bytes/iteration)	Computational Intensity (Flops/iteration)
COPY	a(i) = b(i)	16	0
SCALE	a(i) = q*b(i)	16	1
SUM	a(i) = b(i) + c(i)	24	1
TRIAD	a(i) = b(i) + q*c(i)	24	2

Table 5 STREAM kernels details

This is a serial performance code with no challenges for a parallel system per se, but the results will reflect the performance of the memory subsystems.

We want to test the different memories so we use the `qalloc` routine. Some changes had to be made to the code to take advantage of the dynamic memory. We start by including the `qalloc.h` header file which contains all the information required to use the `qalloc` memory allocation function

```
#include <qalloc.h>
```

We define `QFLAGS` so that we can easily switch between testing for fast memory and for slow memory without having to do many changes to the code

```
#define QFLAGS (QCOMMS|QFAST)
```

The one-dimensional buffers can then allocated by doing the following

```
/* dynamically allocate the arrays onto the fast memory of
QCDOC*/
a = (double *) qalloc(QFLAGS, (N+OFFSET)*sizeof(double));
b = (double *) qalloc(QFLAGS, (N+OFFSET)*sizeof(double));
c = (double *) qalloc(QFLAGS, (N+OFFSET)*sizeof(double));
```

The basic changes to the STREAM C code involve changing the static allocation of the vectors to dynamic allocation so that we can specify the fast memory using the `QCOMMS|QFAST` flag, or the DDR-SDRAM memory using the `QCOMMS` flag.

3.2. Collective Communication Routines

When considering one porting the two-dimensional fast Fourier transform we noticed that collective communications that offer Scatter, Gather, All Gather, and All to All operations would be required. Since they are not immediately available through the QMP library, we implemented them to make the kernel port possible.

For the kernels we study later, we need collective communication routines that are not readily available on the system. We therefore have to implement this functionality.

3.2.1. Description

The collective communications we look at are mainly collective operations that deal with data movement across the available processors [5]. They are blocking collective operations, where all the processors are involved. They build on the foundation of message passing, which is point-to-point communications to offer simpler communication primitives that involve all the processors.

In a scatter operation, a root processor (P0) sends a message that is split into equal segments. The *i*th segment is sent to the *i*th processor.

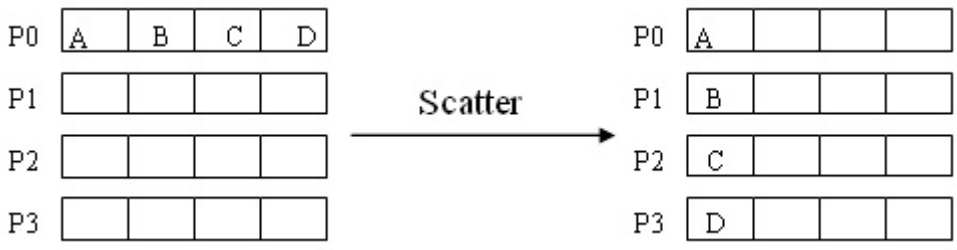

Figure 13 Scatter operation

The gather operation is considered the inverse of the scatter operation. Here messages are passed back from all processors to a root processor (P0). In this the individual segment sent by each processor is concatenated in rank order on the root processor.

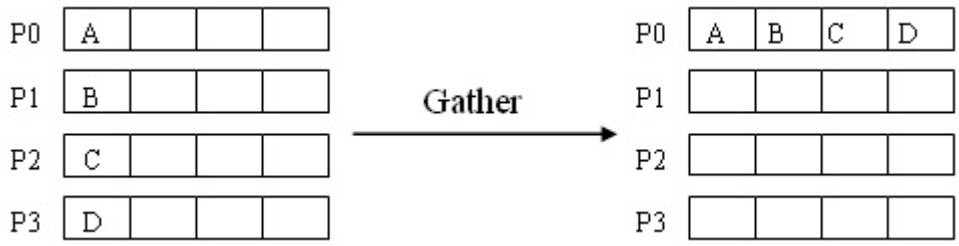

Figure 14 Gather operation

In an All Gather operation, the messages are passed back to all of the available processors.

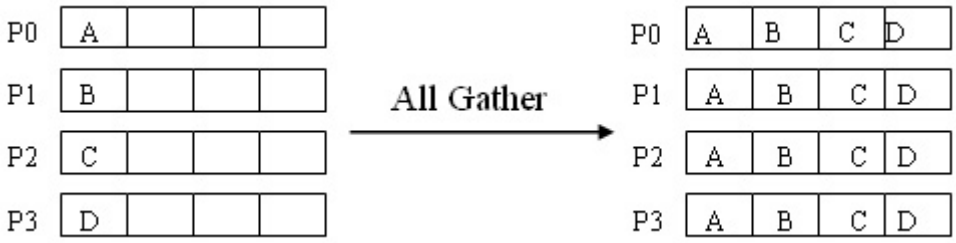

Figure 15 All gather operation

The AlltoAll operation is an extension the All Gather operation, where each processor sends distinct data to each of the receivers. The *j*th block sent from process I is received by process j and is placed in the *i*th block.

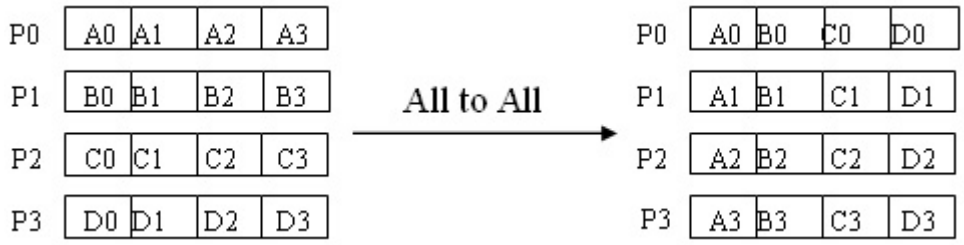

Figure 16 AlltoAll operation

3.2.2. Implementation

We therefore wrote the above collective communications (figure 17, 18, 19, 20). The communication routines were written using the memory copy function written by Michael Creutz [2].

```
qmemcpy(int destproc, void *  dest, int srcproc, void * src, int size);
```

It is worth briefly looking at how the qmemcpy routine works [10]. It allows a message to be sent from one node (source) to another node (destination) while specifying the specific size of the data to be transferred, that is the number of elements and their type. The routine then handles routing the message appropriately across the physical network, using one of the bi-directional links for sending, and the

other for listening. A lookup table is generated with the optimal path the message should follow, if at some point the line is busy then the message enters a FIFO based queue. The communications are implemented using the SCU library, where the exchanged messages are written to a send address register, global interrupt on a line flag store/fetches, and synchronizations.

```
/*
 * scatter(int numnodes, void * destbuf, int srcnode, void * srcbuf, int sendcount, int datatype_size)
 *
 * [ IN numnodes] number of nodes to scatter to in order
 * [ OUT destbuf] address of destination buffer
 * [ IN srcnode] rank of sending process
 * [ IN srcbuf] address of source buffer
 * [ IN sendcount] number of elements to send to each process
 * [ IN datatype_size] size of data type to send/receive
 *
 */
void
gencom::scatter(int numnodes, void * destbuf, int srcnode, void * srcbuf, int sendcount, int datatype_size) {
   printf("");
   int destnode=0;
      for (destnode=0;destnode<numnodes;destnode++) {
         if(processor == srcnode) {
            mycom.qmemcpy((int)destnode,      destbuf,      (int)srcnode,      &(((double      *)srcbuf)[sendcount*destnode]),
(int)sendcount*datatype_size);
         }
         mycom.finishcopy();
         mycom.maxqueue=0;
      }
}
```

Figure 17 Scatter Routine implemented with qmemcpy

```
                                                    e, void * srcbuf, int sendcount,
  int datatype_size)
   *
   * [ IN numnodes] number of nodes to scatter to in order
   * [ OUT destbuf] address of destination buffer
   * [ IN dstnode] rank of receiving process
   * [ IN srcbuf] address of source buffer
   * [ IN sendcount] number of elements to send to each process
   * [ IN datatype_size] size of data type to send/receive
   *
   */
  void
  gencom::gather(int numnodes, void * destbuf, int dstnode, void * srcbuf, int
  sendcount, int datatype_size) {

    int srcnode=0;

      for (srcnode=0;srcnode<numnodes;srcnode++) {
        if(processor == dstnode ) {
          mycom.qmemcpy((int)dstnode,   &(((double   *)destbuf)[srcnode*sendcount]),
  (int)srcnode, srcbuf, (int)sendcount*datatype_size);
        }
        mycom.finishcopy();
        mycom.maxqueue=0;
      }
    }
```

Figure 18 Gather routine implemented with qmemcpy

```
/* * allgather(int numnodes, void * destbuf, void * srcbuf, int sendcount, int
datatype_size)
 *
 * [ IN numnodes] number of nodes to scatter to in order
 * [ OUT destbuf] address of destination buffer
 * [ IN srcbuf] address of source buffer
 * [ IN sendcount] number of elements to send to each process
 * [ IN datatype_size] size of data type to send/receive
 *
 */
void
gencom::allgather(int numnodes, void * destbuf, void * srcbuf, int sendcount, int
datatype_size) {

  int i=0, j=0;

  for (i=0;i<numnodes;i++) {
    for (j=0;j<numnodes;j++) {
      mycom.qmemcpy((int)i, &(((double *)destbuf)[j*sendcount]), (int)j, srcbuf,
(int)sendcount*datatype_size);
      mycom.finishcopy();
      mycom.maxqueue=0;
    }
  }
}
```

Figure 19 Allgather routine implemented with qmemcpy

```
/*
 * alltoall(int numnodes, void * destbuf, void * srcbuf, int sendcount, int datatype_size)
 *
 * [ IN numnodes] number of nodes to scatter to in order
 * [ OUT destbuf] address of destination buffer
 * [ IN srcbuf] address of source buffer
 * [ IN sendcount] number of elements to send to each process
 * [ IN datatype_size] size of data type to send/receive
 *
 */
void
gencom::alltoall(int numnodes, void * destbuf, void * srcbuf, int sendcount, int
datatype_size) {
  int i=0,j=0;

  for (i=0;i<numnodes;i++) {
    for (j=0;j<numnodes;j++) {
      mycom.qmemcpy((int)j, &(((double *)destbuf)[sendcount*i]), (int)i, &(((double
*)srcbuf)[sendcount*j]), \
                  (int)sendcount*datatype_size);
      mycom.finishcopy();
      mycom.maxqueue=0;
    }
  }
}
```

Figure 20 AlltoAll routine implemented with qmemcpy

3.3. An Image Processing Kernel

The first computational kernel we port is a parallel lattice based image analysis program which reconstructs an image from its edge data by Jacobi iteration over a two-dimensional domain. We start by describing the work that is done in this type of program and then show the porting implementation onto the QCDOC system.

3.3.1. Description

The program consists of three main parts: startup, main computation loop and cleanup (figure 17). At startup, the message passing environment is initialized, preliminary checks such as the number of processors available, reading-in the edge data file, and setting up the appropriate data-structures are done.

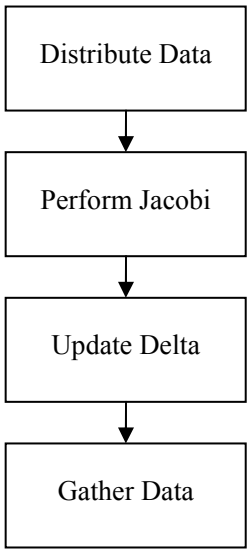

Figure 21 Schematic kernel design of the Jacobi Code

The main computational loop of the kernel involves using the Jacobi algorithm to reconstruct an image from its MxN two-dimensional edge data set using the 5-point stencil described in equation (2)

$$new_{i,j} = 0.25 \times (old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j}) \quad (2)$$

Where *edge* is the edge input data, *old* is the image value at the current iteration and with the *new* is the final value of the reconstructed image at the iteration.

The streaming code is computationally intensive but it is easily parallelised using the two-dimensional domain decomposition. In that sense the code can be distributed over the available processors to benefit from parallel computing techniques. Each iteration of the main computational loop requires nearest neighbor communications (figure 22) to update the boundary data of the node. Each processor then computes the value of its local *new* with the data it has received from its neighbors until the required number of iterations is reached or when a preset stopping criterion or tolerance level is reached. The termination criterion is given by (3)

$$\Delta^2 = \frac{1}{MN} \sum_{i=1;j=1}^{i=M;j=N} (new_{i,j} - old_{i,j})^2 \quad (3)$$

This tolerance level indicates that the result obtained is sufficiently accurate and is obtained by performing a global sum over the available processors.

The final part consists of reassembling the processed data into a main buffer which is used to write the processed image to disk.

This application of the Jacobi algorithm makes a good benchmark style code [16], since the kernel displays a similarity to a variety of scientific codes with nearest neighbor interactions and global summations. What additionally makes this an attractive kernel is that the simplicity of the work taking place enables us to identify causes of poor performance which could be more difficult to understand and interpret in full applications.

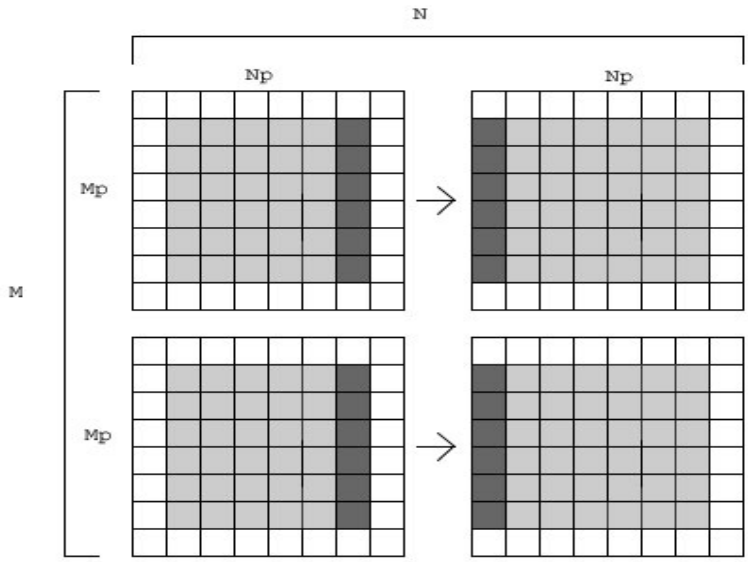

Figure 22 Vertical halo swaps in two-dimensional domain decomposition

Improved versions of the Jacobi algorithm such as the Gauss-Seidel variation and more performant inverters are available for real performance code from a number of high-performance libraries [18].

For our work the efficiency of the algorithm is not our main interest. What we really want is a code that runs well and produces correct results that can be used to reflect the underlying performance characteristics of the system it is running on for our analysis.

We are interested in the typical features of the code such as the nearest neighbor communications, the global post processing, and the porting details of this code to QCDOC which we look at next.

3.3.2. Implementation

At first impression this kernel should be easily ported onto the QCDOC architecture. The bulk of the computation involves nearest neighbor communications which is readily available. The global sum operation required for the calculating the residual is also a routine available from the QMP library. Let's go through the steps involved in porting this code to QCDOC.

The first step in porting the code to QMP on QCDOC involves minor changes to the original code that are not specifically parallel system specific. The original code was written in MPI and C using static two-dimensional arrays. Since there is only a C++ compiler on QCDOC, a simple conflict that arose involved the original code using the variable name *new* for the new buffer, when *new* is actually a reserved C++ keyword. Also The QMP implementation does not offer the equivalent of the MPI timing routine `MPI_Wtime()`. A portable timing routine was therefore added using the standard `gettimeofday()` call.

Adjustments to the code data-structures then have to be made so that they reside on the heap instead of the stack because of the memory limitations static allocation has on QCDOC we mentioned earlier. An example of allocating the main buffer *masterbuf* of size MxN to the heap is (figure 16)

```
double **masterbuf;
masterbuf = (double **) qalloc(QFLAGS, M*sizeof(double));
masterbuf[0] = (double *) qalloc(QFLAGS, M*N*sizeof(double));
for(i=0;i<M; i++){
   masterbuf[i] = masterbuf[0] + (N)*i;
}
```

Figure 23 Dynamically allocating a MxN buffer

We can now focus on message passing aspects of the port. The schematic (figure 18) shows the communication patterns of the original code. Startup and finalization involve scattering the image data for processing to the nodes then gathering the results is done with point to point messages.

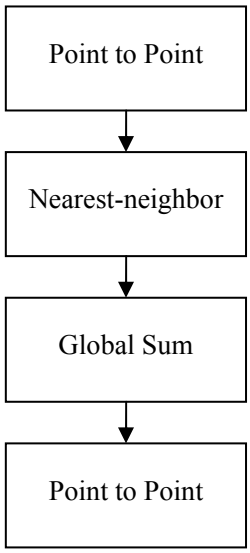

Figure 24 Communication Pattern of Original Code

In the original code, the image dataset is then read into a master buffer on the processor of rank 0, and broadcasted to the other processors using MPI_Bcast. The equivalent in QMP is a `QMP_broadcast()`. We changed the original code, so that all the nodes read the image data from file at a different offset instead. This parallel

read is advantageous in several ways. We first get rid of the inefficient broadcast operation, which in the event the image data cannot fit into the memory of the root node because impractical, and take advantage of a parallel I/O. No time is lost broadcasting or even scattering the input data to the nodes. Instead the nodes fill their buffers themselves.

We can therefore replace the MPI collective communication routines of `MPI_Scatter()` and `MPI_Gather()` with the ones we have written for QCDOC. For this kernel we went for a different approach. The alternative to using the collective routines we implemented in section (3.2) has to do with an algorithmic change: we would like to work on a dataset that is much larger than the memory available on one processor and the gather and scatter routine run into memory problems when reading large amounts of data.
So the approach we opted for was to have the nodes read-in the image data from separate offsets of the data file and separately write their own files at the end of the run. This is not a port change but an algorithmic change which one would expect to perform better on several architectures [19].

This is done in the code by calling the `qmp_datread_offset()` function with the main algorithm for reading at the right offset of the input file shown (figure 20)

```
int top=myrow*ny,
  bottom=(myrow+1)*ny,
  left=mycolumn * nx,
  right=(mycolumn+1) * nx;

/* read data into buffer at different offset */
QMP_barrier();
for (j=0; j<N; j++) {
  for (i=0; i<M; i++) {
    fscanf(fp,"%d", &t);

    if(i<right && i>=left && j< bottom && j>=top)
      x[(j-top) + ny*(i-left)] = t;
  }
}
QMP_barrier();
```

Figure 25 Parallel read of image data snippet

Reading in parallel is easier to implement than writing in parallel because writing brings out coherency issues with buffering at different levels of the system. So we eliminated that difficulty by having each processor write his individual file. The complete image can then be recreated by combining the output images in a post-processing phase using a standard open source image processing package [17].

In addition, instead of always reading data from an image dataset a random number (appendix link) generator was used to fill the node buffers instead. This is done via the function `rndm_data_fill()` which fills the supplied buffer with random data of type double. This is not a performance critical part of the code, but for many runs initializing the buffers with random data is a matter of convenience. This allows us to skip the I/O aspect to focus on the performance at the computation and messaging level. Also, it allows us to vary the dataset size that we are considering without having to generate an image of the required size.

Now that the two point to point steps of the program (figure 18) have been replaced by parallel I/O to fit bigger data into the processing nodes, faster and achieve better scalability, we can focus on the core part of the port, which involves the nearest neighbor communications and the global sum.

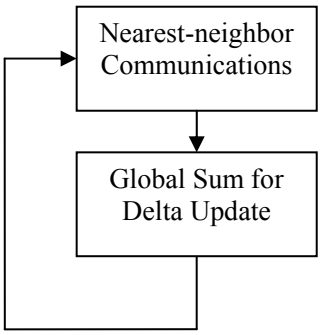

Figure 26 Communication pattern of QCDOC computational loop

The MPI environment is initialized and a two dimensional Cartesian communicator is created with defined data types for vertical (contiguous) and horizontal (non-contiguous) halo swaps. Each node's neighbors (up, down, left, right) are identified using MPI_Cart_shift. The equivalent in QMP involves setting up the QMP environment with a call to `QMP_init_msg_passing()`. There is no equivalent to communicators in QMP, but a Cartesian topology is defined by mapping the machine dimensions and application dimensions accordingly. For example, on a 1x1x1x2x2x2 six dimensional machine partition, we map the application dimensions to a two dimensional topology.

```
qpartition_remap –X45 –Y0123
```

We then have a 4x2 two dimensional application topology.

The horizontal and vertical halo swaps are performed using calls to non-blocking MPI sends and receives using `MPI_Isend()` and `MPI_Irecv()` in the original code.

```
mm = QMP_declare_msgmem(&old[Mp][1], (Np * sizeof(double)));
mh = QMP_declare_send_relative(mm, dir, isign, 0);

QMP_start(mh);
QMP_wait(mh);

mm = QMP_declare_msgmem(&old[Mp + 1][1], (Np * sizeof(double)));
mh = QMP_declare_receive_relative(mm, dir, isign, 0);
```

Figure 27 Halo swapping with QMP

In these send/receive constructs the direction `dir` and the sign `isign` indicate the axis along which data is being communicated and in which direction. This allows one to specify to which neighboring node the messages should be sent.

```
mm = QMP_declare_strided_msgmem(&old[1][Np], sizeof(double), Mp,
((Np+2) * sizeof(double))); /* send down */
     mh = QMP_declare_send_relative(mm, dir, isign, 0); /* send down
*/

QMP_start(mh);
QMP_wait(mh);

mm = QMP_declare_strided_msgmem(&old[1][Np + 1], sizeof(double), Mp,
((Np+2) * sizeof(double))); /* recv down*/
mh = QMP_declare_receive_relative(mm, dir, isign, 0); /* recv down
*/

QMP_start(mh);
QMP_wait(mh);
```

Figure 28 Strided halo swap with QMP

A call is then made to the `update_delta()` function which computes delta (3), the termination criterion which is initially set at a tolerance of 0.03. The update of a tolerance level "delta" requires a global reduction which is done in the orginal MPI code using `MPI_Allreduce()`. The QMP equivalent simply involves calling

The address of the variable to be summed and updated is passed as a parameter.
This shows how the operation of summing data of type double is performed.

Lastly, in the original code, the compute nodes return their data to the processor of rank 0 using non-blocking point to point communications, and the rank 0 rebuilds the master buffer and writes the processed image to disk.

The correctness of the program is verified by the initial image outputs, and tests on different sized images, before switching to filling the buffers with data generated randomly.

3.4. A Fast Fourier Kernel

3.4.1. Description

A fast Fourier transform code is an efficient algorithm to find the one-dimensional discrete Fourier transform (DFT) (4) and its inverse.

$$\widetilde{f}(k) = \sum_{x} f(x) e^{i \cdot \left(\frac{2\pi \cdot kx}{N} \right)} \tag{4}$$

FFTs are of absolute importance to a wide variety of scientific applications ranging from signal processing to computational chemistry and finding solutions to partial differential equations. Despite the reliance on FFTs for scientific applications, they often present a key performance bottleneck.

This makes looking at FFTs interesting in our study. They have collective communication features which include scattering, gather and performing AlltoAll operations on the data. Also the FFT itself is relatively computationally demanding.

Let us briefly look describe FFTs before we look at how we ported the FFT code to QCDOC.

Multidimensional FFTs can be built on top of one-dimensional FFTs. So if we consider a two-dimensional FFT (5), then it can be built as a sequence of two one-dimensional FFTs.

$$\tilde{f}(k,l) = \sum_{y}\sum_{x} f(x,y) e^{i\cdot\left(\frac{2\pi\cdot kx}{N}+\frac{2\pi\cdot ly}{M}\right)} \tag{5}$$

This can be rewritten as (6)

$$\tilde{f}(k,l) = \sum_{y}\left[\sum_{x} f(x,y) e^{i\cdot\left(\frac{2\pi\cdot kx}{N}\right)}\right]\cdot e^{i\cdot\left(\frac{2\pi\cdot ly}{M}\right)} \tag{6}$$

So that reaching a solution involves working with the partial Fourier transform (7)

$$\tilde{f}(k,l) = \sum_{y} \hat{f}(k,y)\cdot e^{i\cdot\left(\frac{2\pi\cdot ly}{M}\right)} \tag{7}$$

This shows how two-dimensional FFTs can be calculated as a series of two one-dimensional FFTs.

What makes this more appealing to parallel computers is that each FFT is trivially parallel in along a row or column of the matrix a fact that can be exploited be highly exploited for parallel performance.

3.4.2. Implementation

The FFT program we consider is a simple FFT code that does not implement a highly performant FFT [19]. The motivation is not to implement a performant FFT but to understand the porting issues and to put to use our simple collective communication routines.

We do not do any changes to the FFT algorithm that is supplied with the code as it is written in C and ran immediately on the different architectures without requiring any changes. In that sense the port involves replacing the collective communication

routines used in the FFT kernel with the appropriate QMP and qmemcpy based calls.

Statically allocated arrays are replaced (figure 28) in order to be able to run bigger problem sizes without facing the memory limitations available on the stack on QCDOC stack.

```
a = (mycomplex **) qalloc(QFLAGS, IMAGE_SIZE*sizeof(mycomplex));
b = (mycomplex **) qalloc(QFLAGS, IMAGE_SIZE*sizeof(mycomplex));
a[0] = (mycomplex *) qalloc(QFLAGS, IMAGE_SIZE*IMAGE_SIZE*sizeof(mycomplex));
b[0] = (mycomplex *) qalloc(QFLAGS, IMAGE_SIZE*IMAGE_SIZE*sizeof(mycomplex));
for(i=0;i<IMAGE_SIZE; i++){
  a[i] = a[0] + (IMAGE_SIZE)*i;
  b[i] = b[0] + (IMAGE_SIZE)*i;
}
```

Figure 29 Allocating the a, b two-dimensional buffers onto the heap

After the image data set is initialized to a point source, and the twiddle factors are pre-computed, the communication environment is setup by calling

```
mycom.start();
```

The two-dimensional image is the distributed from the master processor by rows to the available processors (figure 23).

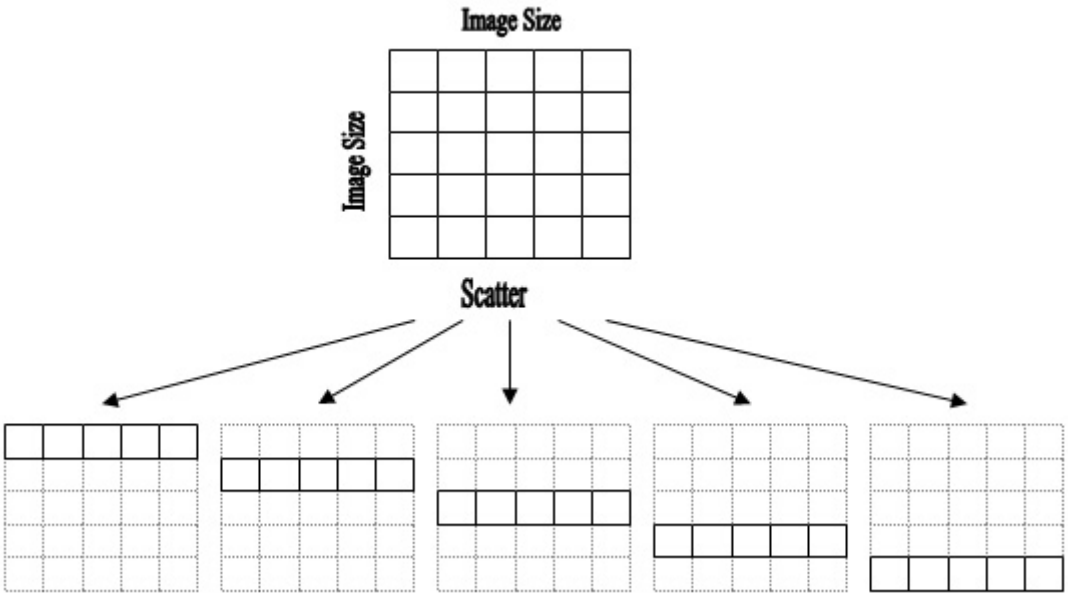

Figure 30 Scattering to available processors of image data by rows

`MPI_Scatter()` is originally called to distribute the input matrix by rows to the available processors.

```
MPI_Scatter((char *) a, IMAGE_SLICE * IMAGE_SIZE * 2, MPI_DOUBLE,
            (char *) a_slice, IMAGE_SLICE * IMAGE_SIZE * 2,
             MPI_DOUBLE,SOURCE_PROCESSOR, MPI_COMM_WORLD);
```

This is replaced by a call to

```
mycom.scatter(numtasks, (char *)*a_slice, SOURCE_PROCESSOR,
          (char *)*a, IMAGE_SLICE * IMAGE_SIZE * 2, sizeof(double));
```

Each processor then performs a one-dimensional FFT on the rows of the local image.

The image is then transposed using the `MPI_Alltoall()` function, which partitions the intermediate image into columns

```
MPI_Alltoall(a_chunks, IMAGE_SLICE * IMAGE_SLICE * 2, MPI_DOUBLE,
             b_slice, IMAGE_SLICE * IMAGE_SLICE * 2, MPI_DOUBLE,
             MPI_COMM_WORLD);
```

This is replaced by a call to

```
mycom.alltoall(numtasks,  *b_slice,  a_chunks,  IMAGE_SLICE  *
IMAGE_SLICE * 2, sizeof(double));
```

Each processor then performs a second one-dimensional FFT but on the columns of the local image.

Finally the columns of the image a collected back to the master processor (figure 27) and the output image is tested for correctness.

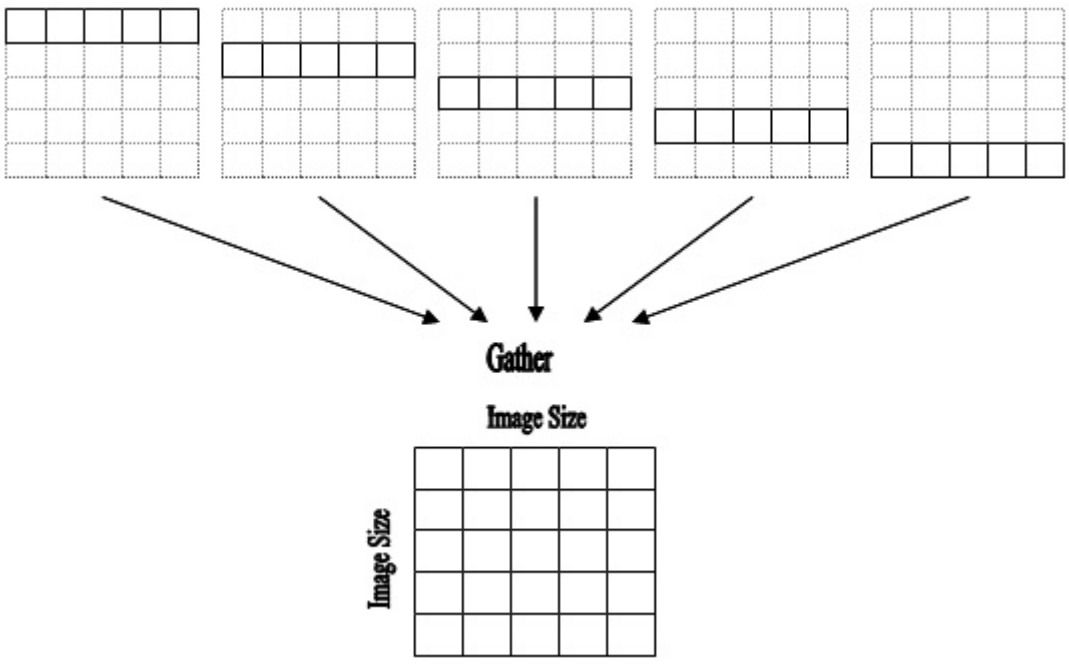

Figure 31 Gathering of output matrix by rows to master processor

The processed image columns are lastly gathered back to the master processor using MPI_Gather, and timing details of the run are printed out to screen.

```
MPI_Gather(a_slice, IMAGE_SLICE * IMAGE_SIZE * 2, MPI_DOUBLE,
           a, IMAGE_SLICE * IMAGE_SIZE * 2, MPI_DOUBLE,
           DEST PROCESSOR, MPI COMM WORLD);
```

This translates to

```
mycom.gather(numtasks, *a, DEST_PROCESSOR, *a_slice, IMAGE_SLICE *
IMAGE_SIZE * 2, sizeof(double));
```

The correctness of the program is verified initially by comparing the outputs of the MPI and QMP runs on different machines, and lastly the communication environment is stopped

```
mycom.stop();
```

3.5. Porting Issues

This is a summary of the problems encountered while porting.

- Initially the lack of a Fortran compiler eliminated a wide class of applications that were interesting to consider as benchmark codes. Also despite the fact that a C++ compiler is present, there is no C compiler, therefore requiring minor adjustment to make the C code work.

- The lack of MPI implementation for QCDOC is clearly a porting drawback. Its availability would have made our work more straightforward, as well as the machine more accessible. Nevertheless we understand that implementing MPI for QCDOC was never part of the project.

- The lack of built-in, high-performance point-to-point communications is a key drawback to making porting a more straightforward task.

- Often used memory automatically goes to the higher levels of cache on cache based architectures for re-use based on the principle of data locality. In the case of QCDOC because of the absence of L2 and L3 cache, we must explicitly place the data onto the EDRAM instead of the DDR-SDRAM to take advantage of fast memory. In the following chapter we discuss the performance benefits we obtain from doing this.

4. Performance Results and Analysis

4.1. Memory Bandwidth

We present the results (figure 32) of the stream benchmark on QCDOC using the fast (EDRAM) and slow (DDR-SDRAM) memory, as well as on the Bluegene system

The codes were compiled on QCDOC with the highest optimization level –O6.
On Bluegene two levels of optimization were used, the default optimization and the optimization level 3.

The problem size considered for the STREAM dataset is of 16MB for the Bluegene system. This number is big enough to be suitable for this benchmark, giving a factor of 4 times the largest cache on Bluegene which is the 4MB L3.

For QCDOC, the problem size is of 3MB in order for it to fit into the EDRAM runs, and it is of 16MB for the runs made on the DDR-SDRAM.

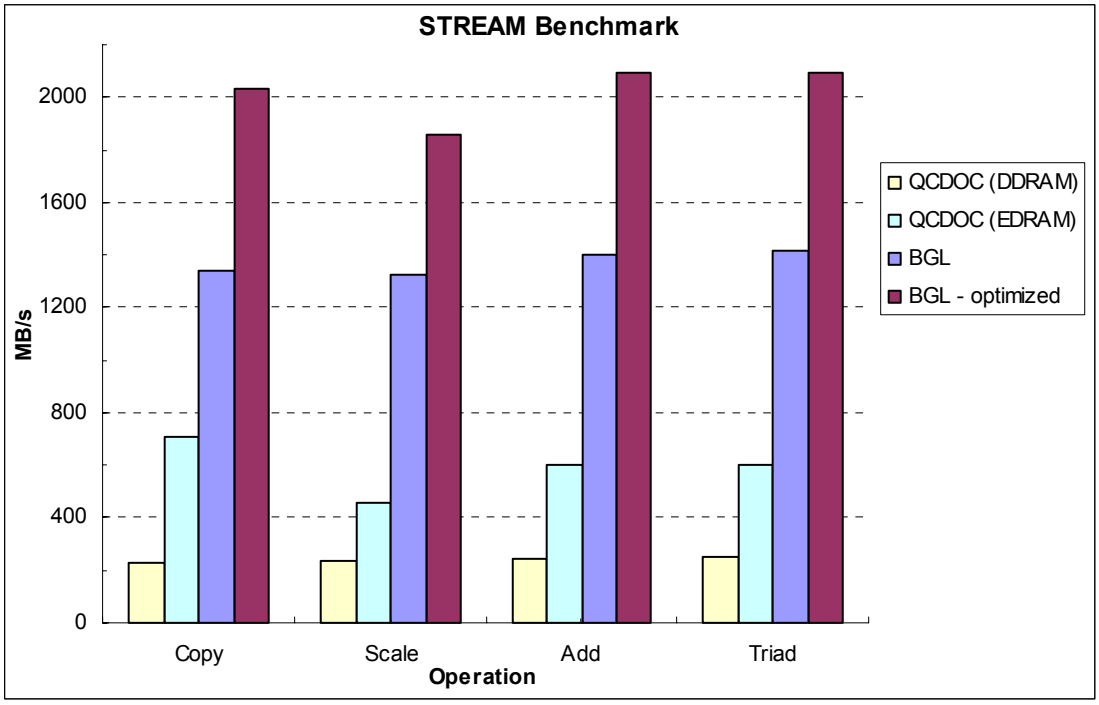

Figure 32 STREAM on Bluegene and QCDOC

The results match our initial expectation of the fast EDRAM memory performing better that the DDR-SDRAM on QCDOC. We notice that the fast EDRAM memory is around 3 times slower than the DDR-SDRAM which matches the system design.

Better results for QCDOC where obtained [1] by using optimization techniques such

as loop unrolling and data prefetching which increased the average memory bandwidth up to 1024 MB/s. Additionally using assembly language optimizations the memory bandwidth was pushed up to 1670 MB/s.

The Bluegene results however are far better than those of QCDOC and in comparison, the QCDOC performance is poorer that the clock ratio of 7/4 that is expected.

This does not meet our expectations. It indicates that the memory architecture of QCDOC is not fully exploited through the GNU compiler, whereas better performance is achieved with the Bluegene system compilers to efficiently exploit the maximum system memory bandwidth.

Nevertheless the results show that despite the lack of L2 and L3 cache on QCDOC using the fast memory yields an appreciable factor increase in performance.

4.2. Image Processing Kernel

In trying to understand the behavior of the image processing code under the different platforms, we looked at a different number of iterations, problem sizes, and optimizations flags.

For QCDOC, the usage of both fast EDRAM memory and slow DDR-SDRAM memory was examined. This is indicated in the graphs by QCDOC (QFAST) for the EDRAM and QCDOC (QCOMMS) for the DDR-SDRAM.

In order to measure the kernel performance we start by looking (figure 33) at the parallel speed-up we are obtaining by considering the equation (8).

$$S(N,P) = \frac{T(N,1)}{T(N,P)} \tag{8}$$

This is the ratio of the execution time on one processor over the execution time on P processors for the problem size N.

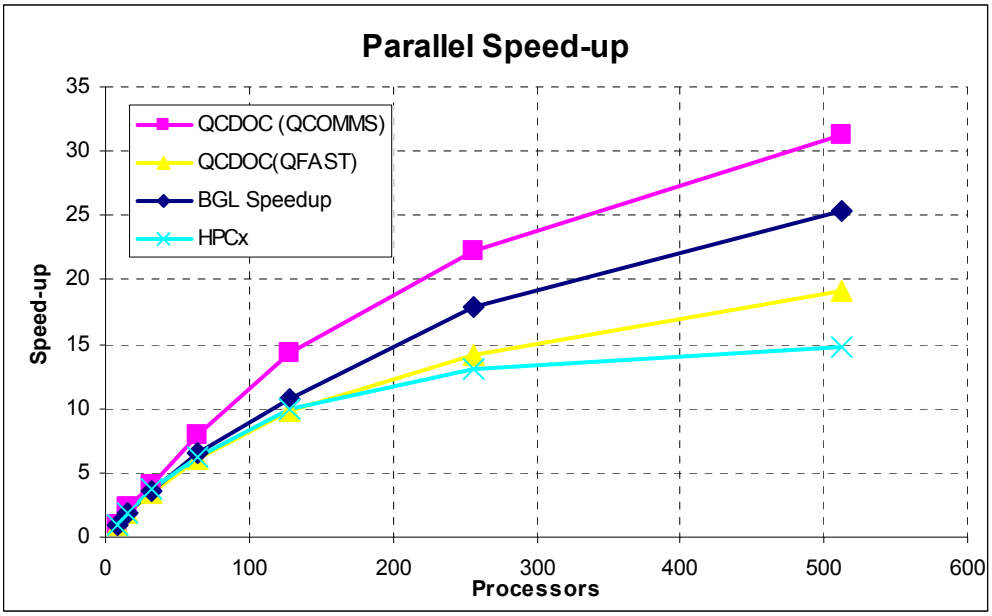

Figure 33 Parallel speed-up of image processing kernel

This shows that QCDOC with the DDR-SDRAM is obtaining a good speed-up factor as the number of processors increase.

Parallel efficiency is equal to the speed-up divided by the number of processors P. It is expressed as in (9)

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{PT(N,P)} \tag{9}$$

Parallel efficiency decreases with the number of processors, but as with the problem sized increases we gain efficiency according to Amdahl's law.

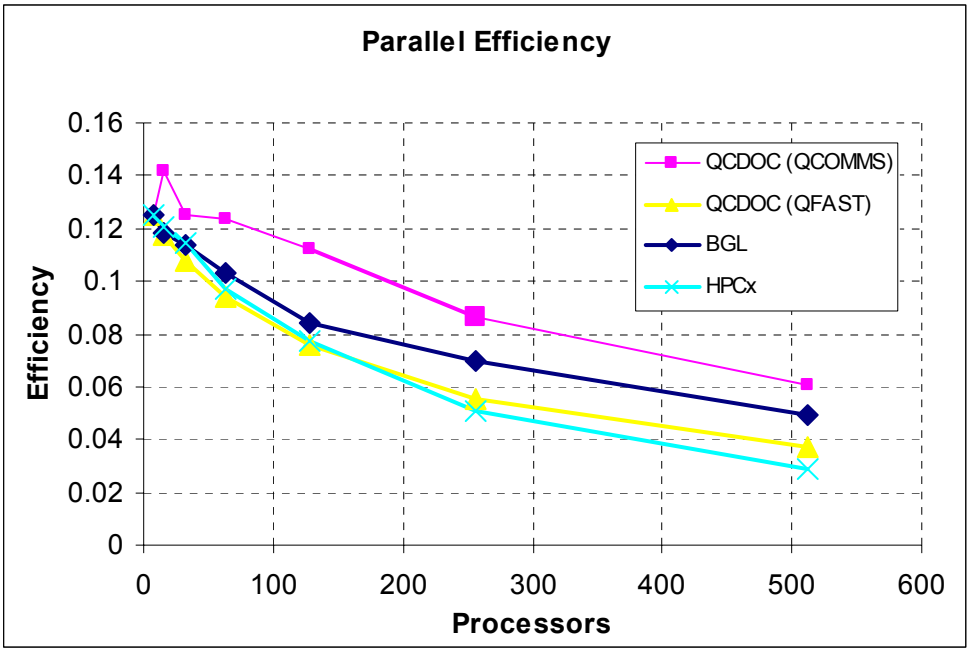

Figure 34 Parallel efficiency of image processing kernel

If we look (figure 35) at the speed-up in the Jacobi algorithm part of the image processing code, it matches with the overall kernel performance.

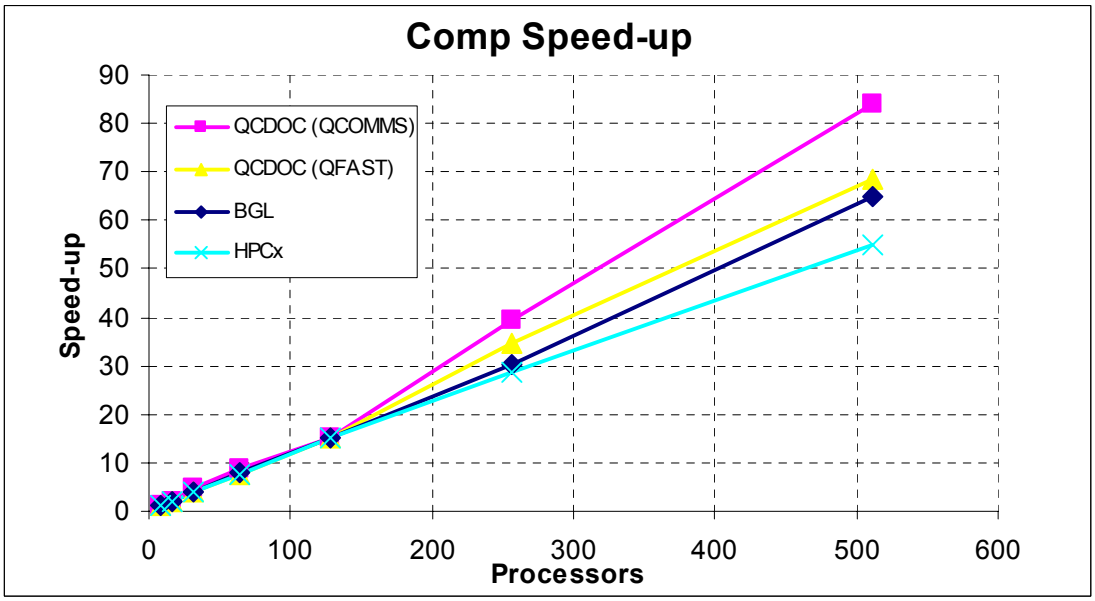

Figure 35 Computational speed-up of image processing kernel

By considering the Time x Processors graph (figure 35) at a logarithmic scale though we can better appreciate the overall performance of the code on the different architectures. This not only shows how well the code is scaling, but always gives a clearer view of how fast the kernel is actually running too.

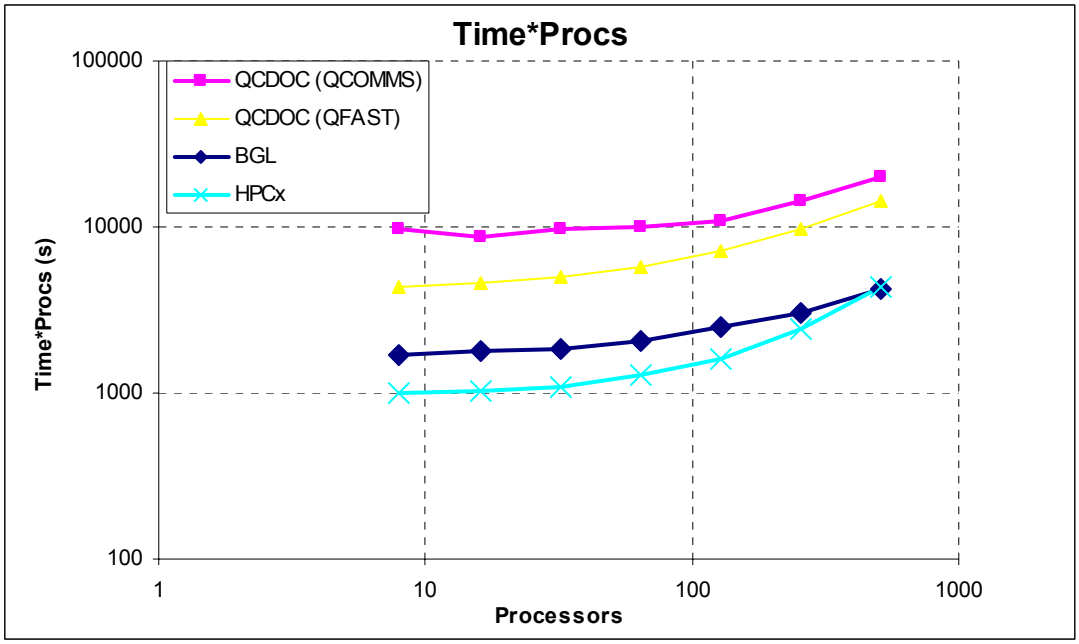

Figure 36 Time * Processors logarithmic plot for image processing kernel

Here the flatter curve indicates better scaling of the kernel. Now we can clearly see that the QCDOC runs are being outperformed by Bluegene and the HPCx system. Despite that QCDOC scales pretty well with the number of processors, this is mainly due to its mesh based communications network.

Using the DDR-SDRAM with QCDOC offers actually scaling better, but at the cost of a worse performance than if the fast EDRAM was used instead.

4.4. FFT Kernel

The input matrix considered for our comparison runs contains 1024x1024 complex elements.

We start by looking at (figure 37) the results of the run on QCDOC with two varying configurations. The first one involves 32 processors configured in a 4x8 layout and the second consists of 64 processors in an 8x8 layout.

Lattice Size	Processors (layout)	Scatter Time	1d-fft-row Time	Transpose Time	1d-fft-column Time	Gather Time	Total Time
1024x1024	32 (4x8)	0.888968	0.154458	23.08995	0.165683	0.920951	25.22001
1024x1024	64 (8x8)	0.882371	0.077472	24.015203	0.083274	0.925177	25.983497

Figure 37 FFT on QCDOC using qmemcpy based routines

The first thing to notice is that the dominant factor is the transpose time. That is the kernel performance is communications limited. As the number of processors increases, that time is also increasing.

This clearly shows that our AlltoAll() routines is not efficient enough. Whereas the scatter and gather routines seem to be performing better.

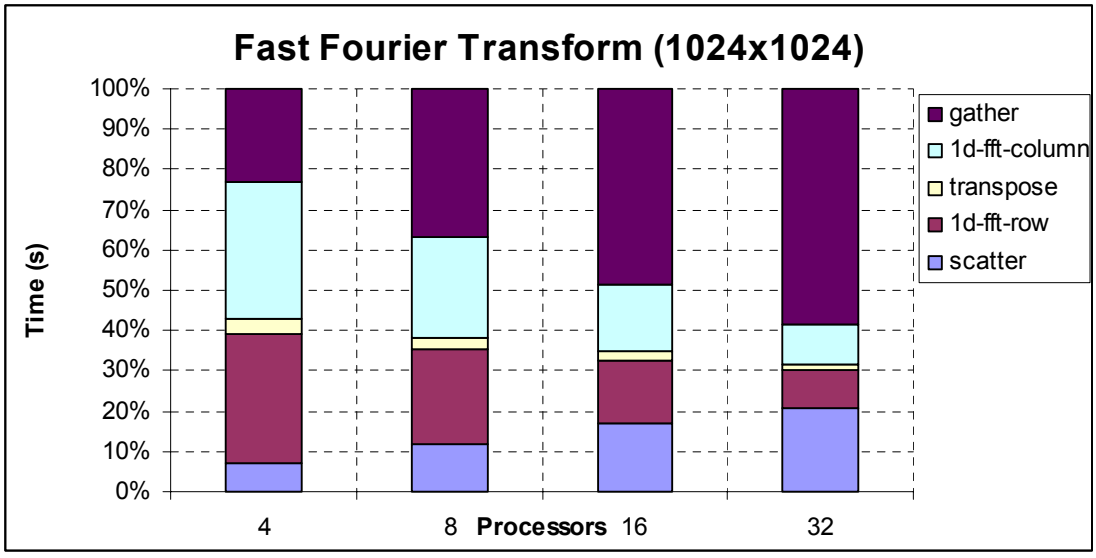

Figure 38 FFT on BlueGene

If we look at the performance of the FFT code on Bluegene (figure 38), we see an increase in the time taken by the gather routine as the number of processors increases.

Similarly to QCDOC though, the serial part of the code performing the one-dimensional FFTs is being halved as the number of processors double. Scaling performance is being gained but at the price of increasing time spent in collective data movement communications.

In brief, the serial performance on the QCDOC runs performance scaled reasonably well whereas the transpose did not.

5. Conclusions

5.1. Summary of our Contributions

In this work we have achieved our goal of porting different computational kernels to QCDOC. This has allowed us to become more familiar with porting parallel codes to QCDOC, specifically porting MPI codes to QMP.

This work has shed the light on the issues involved in the porting process, such as the unavailability of general collective communication routines on QCDOC. We therefore made a first attempt to add the required collective communication routines using the convenient `qmemcpy` routine because it was available and easy to use.

We also looked at the memory placement techniques used on QCDOC. This showed us that codes can benefit greatly from running on QCDOC by exploiting the fast EDRAM memory available on the system.

Lastly although the Gather and Scatter routines which are not performance critical performed well, we noted that the performance critical AlltoAll routine does needs to be improved in order to allow a greater number of scientific applications to take advantage of QCDOC.

5.2. Limitations and Future Directions

The limited time did not allow us to look further into optimizing the AlltoAll communication on QCDOC. The fact that there is no standard optimized AlltoAll communication on QCDOC makes the system less attractive for a large number of scientific applications that require it. Probably the next milestone should focus on making optimized collective communication routines available.

6. Appendix

6.1. Code Fragments

6.1.1 Timing Routine

```
/* timing function */
double
     get_current_time() {
     struct timeval tv;
     gettimeofday(&tv, 0);
     return (double)tv.tv_sec + tv.tv_usec(1.e-6);
}
```

6.1.2 Basic SCU Communication Example

```
#include <qalloc.h>
#define BUFFERSIZE 80

int main(){
  SCUDirArgIR send, receive;
  char *mybuffer1,*mybuffer2;
  DefaultSetup();

  /* allocate and fill communication buffers */
  mybuffer1 = (char *) qalloc(QNONCACHE|QCOMMS|QFAST,BUFFERSIZE);
  mybuffer2 = (char *) qalloc(QNONCACHE|QCOMMS|QFAST,BUFFERSIZE);
  sprintf(mybuffer1,"hello world from processor %d",UniqueID());
  sprintf(mybuffer2,"incoming message will go here");
  printf("mybuffer1: %s\nmybuffer2: %s\n",mybuffer1,mybuffer2);

  /* transfer buffer1 to buffer2 down the W direction */
  send.Init(mybuffer1,SCU_WM,SCU_SEND,BUFFERSIZE,1,8);
  receive.Init(mybuffer2,SCU_WP,SCU_REC,BUFFERSIZE,1,8);
  send.StartTrans();
  receive.StartTrans();
  send.TransComplete();
  receive.TransComplete();
  printf("mybuffer1: %s\nmybuffer2: %s\n",mybuffer1,mybuffer2);

  qfree(mybuffer1);
  qfree(mybuffer2);
  return 0;
}
```

6.1.3 Jacobi Update

```
/* compute jacobi */
void
compute_jacobi(int Mp, int Np, double ** old, double ** edge, double ** newb) {

  int i, j;

  for (i = 1; i < Mp + 1; i++)
    for (j = 1; j < Np + 1; j++)
      newb[i][j] = 0.25 * (old[i - 1][j] + old[i + 1][j] + old[i][j - 1]
            + old[i][j + 1] - edge[i][j]);
}
```

6.1.4 Parallel Read Routine

```
qmp_datread_offset(char *filename, void *vx, int M, int N, int nx, int
ny, int myrank, int ndims, int myrow, int mycolumn) {
  FILE *fp;
  int nxt, nyt, i, j=0, t;
  double *x = (double *) vx;

  if (NULL == (fp = fopen(filename,"r" ))) {
    perror(filename);
    QMP_fprintf(stderr, "datread: cannot open \"%s\"\n", filename);
    QMP_fprintf(stderr, "check M,N values in imagempi.h\n", filename);
    QMP_abort(1);
    exit(-1);
  }

  /* read in header data */
  QMP_barrier();

  fscanf(fp,"%d %d",&nxt,&nyt);

  if (M != nxt || N != nyt) {
    QMP_fprintf(stderr,"datread:  size  mismatch,  (nx,ny)  =  (%d,%d)
expected (%d,%d)\n",
     nxt, nyt, nx, ny);
    exit(-1);
   }
  QMP_barrier();


  int top=myrow*ny,
    bottom=(myrow+1)*ny,
    left=mycolumn * nx,
    right=(mycolumn+1) * nx;

  /* read data into buffer */
  QMP_barrier();
  for (j=0; j<N; j++) {
    for (i=0; i<M; i++) {
      fscanf(fp,"%d", &t);

      if(i<right && i>=left && j< bottom && j>=top)
        x[(j-top) + ny*(i-left)] = t;
    }
  }
   QMP_barrier();

  fclose(fp);
}
```

6.1.5 Random Number Generator Routine

```
/* fill buffer with random data of type double */
void
rndm_data_fill(void *vx, int nx, int ny, int rank) {

  int i,j;
  double *x = (double *) vx;

  printf("filling buffers with %dx%d of random data\n", nx, ny);

  /* seed based on current time */
  srand(static_cast<unsigned>(time(0)*rank-(rank%4)));

  /* fill array with random data of type double */
  for (i=0;i<nx * ny;i++)
     x[i] = (double) (rand()%240);

}
```

6.1.6 Image Processing Kernel Main Loop

```
/* loop over iterations to recreate the image from the edges */
  for (delta = 10.0 * TOL, iter = 1; iter <= MAXITER && delta > TOL; iter++) {

  comm_time-=get_current_time();
    do_left_right_swap(mm, mh, Mp, Np, old, mycolumn, columns);
    do_up_down_swap(mm, mh, Mp, Np, old, myrow, rows);
  comm_time+=get_current_time();

  comp_time-=get_current_time();
    compute_jacobi(Mp, Np, old, edge, newb);
  comp_time+=get_current_time();

    /* compute and update delta */
  delta_time-=get_current_time();
    delta = update_delta(M, N, iter, rank, Mp, Np, old, newb,delta);
  delta_time+=get_current_time();

    /* set old array equal to newb, excluding halos *
     * copy old array back to buf, excluding halos */
  buf_update_time-=get_current_time();
    for (i = 1; i < Mp + 1; i++)
      for (j = 1; j < Np + 1; j++) {
        old[i][j] = newb[i][j];
        buf[i - 1][j - 1] = old[i][j];
      }
  buf_update_time+=get_current_time();

  }
  time_end = get_current_time();
```

6.2. Timetable

Tentative Schedule	Objectives	Tasks	Resources
3 weeks	Implement Image Analysis Code	Port C/MPI Code to C/QMP on QCDOC	C/MPI Code, QMP Standard, and QCDOC 64 node Partition
1 week	Update Documentation and Produce Interim Report	Write User Manual, Write User Report	Experimental Data and Literature Review
3 weeks	Implement another kernel	-	-
2 weeks	Compare Results with Other Systems**	Port Kernel to Other Platforms and Analyse Performance/Issues	QCDOC, BlueSky, HPCx, Lomond
3 weeks	Produce Thesis Paper	Write Thesis Paper	Experimental Data And Literature Review

6.3. Risk Analysis

Risk	Likelihood	Impact	Mitigation
Running overtime in porting a kernel	60%	Key project goals are not satisfied	Scrap one of the kernels
Running late in gathering data for analysis	40%	Write up is delayed because of lack of results	Using meetings and feedback to drive regular results findings
Communication Routines are not highly effective	60%	Limited impact from thesis in presenting new routines	Focus on porting the kernels. Communication routines are additional goals.

6.4. List of Acronyms

DMA: Direct Memory Access
DRAM: Dynamic Random Access Memory
EDRAM: Embedded DRAM
EPCC: Edinburgh Parallel Computing Centre
FFT: Fast Fourier Transform
FLOP: Floating Point Operation
GCC: Gnu C Compiler
GNU: Gnu is Not Unix
HPC: High Performance Computing
JTAG: Joint Test Action Group
MIMD: Multiple Instruction Multiple Data
MPI: Message Passing Interface
QCD: Quantum Chromo Dynamics
QCDOC: Quantum Chromodynamics On a Chip
QDP: QCD Data Parallel
QMP: QCD Message Passing
QOS: QCDOC Operating System
RISC: Reduced Instruction Set Architecture
SCU: Serial Communications Unit

7. Bibliography

[1] *Overview of the QCDSP and QCDOC computers*, P. Boyle et Al. IBM Research and Development Journal Vol. 49, No. 2/3 March/May 2005 (http://www.research.ibm.com/journal/rd/492/boyle.pdf)

[2] Michael Creutz, qmemcpy memory copy routine source code, (http://thy.phy.bnl.gov/~creutz/qcdoc/qmemcpy/include/qmemcpyoc.C)

[3] The HPCx System Website, (http://www.hpcx.ac.uk/services/hardware)

[4] The University of Edinburgh Blue Gene system website, (http://www.epcc.ed.ac.uk/computing/services/BlueGene)

[5] *MPI: A Message-Passing Interface Standard*, Version 1.1, The Message Passing Interface Forum, June 12, 1995.

[6] *Quantum Fields on a Lattice*, István Montvay and Gernot Münster, Cambridge University Press 1994

[7] *Lattice gauge theories -- an introduction*, H.J. Rothe, World Scientific, Singapore, New Jersey, London, Hong Kong (1992).

[8] *QCDSP Machines: Design, Performance and Cost*, D. Chen et Al. ACM/IEEE SC97, 1998

[9] QMP: LQCD Message Passing API (Version 2.0) Oct 29, 2004 (http://www.lqcd.org/qmp/QMP-2-0-Introduction.html)

[10] *Message passing on the QCDSP supercomputer*, Michael Creutz, Nuclear Physics Proceedings Supplement, volume 83, 2000, hep-lat/9908024

[11] *An Introduction to Using the QCDSP Computer*, Balint Joó, Robert Mawhinney, (http://quark.phy.bnl.gov/www/docs/user_guide_v0.ps.Z)

[12] *QCDOC Quick Start Guide*, Balint Joó, (http://www.ph.ed.ac.uk/~bj/QCDOC_quick_start.html)

[13] *Using QCDOC*, Michael Creutz, (http://quark.phy.bnl.gov/~creutz/qcdoc/using.txt)

[14] *An Introduction to the QCDOC Computer*, Chulwoo Jung, (http://quark.phy.bnl.gov/~creutz/qcdoc/qos-1.ps)

[15] Joachim Hein, Mark Bull, *Capability Computing, Achieving Scalability on over 1000 Processors*, HPCx Technical Report HPCxTR0301, 2003, (http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0301.pdf)

[16] ImageMagick website, (http://www.imagemagick.org/script/index.php)

[17] STREAM benchmark website, (http://www.cs.virginia.edu/stream/)

[18] IBM Engineering and Scientific Subroutine (ESSL) Library Book Online, (http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp?topic=/com.ibm.cluster.essl.doc/esslbooks.html)

[19] Two-dimensional fast Fourier Transform C code, George Gusciora, (http://www.llnl.gov/computing/tutorials/mpi/samples/C/mpi_2dfft.c)

[20] *"File I/O from multi processor jobs"*, Joachim Hein, HPCx Technical Report HPCxTR0306, 2003, (http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0306.pdf)