HETEROGENEOUS NPACI-ROCKS/MPI/CUDA DISTRIBUTED MULTI-GPGPU APPLICATION FOR SEEKING COUNTEREXAMPLES TO BEAL’S CONJECTURE: MPI/CUDA INTEGRATION COMPONENT

A Project
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Sudhir Balasubramanian
Summer 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Project of Sudhir Balasubramanian:

Heterogeneous NPACI-Rocks/MPI/CUDA Distributed Multi-GPGPU Application for
Seeking Counterexamples to Beal’s Conjecture: MPI/CUDA Integration Component

________________________________________
William Root, Chair
Department of Computer Science

____________________________
Carl Eckberg
Department of Computer Science

____________________________
Robert Grone
Department of Mathematics and Statistics

11/12/2010
Approval Date
Copyright © 2011
by
Sudhir Balasubramanian
All Rights Reserved
ABSTRACT OF THE PROJECT

Heterogeneous NPACI-ROCKS/MPI/CUDA Distributed Multi-GPGPU Application for Seeking Counterexamples to Beal’s Conjecture: MPI/CUDA Integration Component

by

Sudhir Balasubramanian
Master of Science in Computer Science
San Diego State University, 2011

Beal’s Conjecture asserts that if \( Ax + By = Cz \) for integers \( A,B,C > 0 \) and integers \( x,y,z > 2 \), then \( A, B, \) and \( C \) share a common prime factor. While empirical computational studies by several researchers have established that Beal’s Conjecture holds for all \( A,B,C,x,y,z < 1000 \), the truth of the general conjecture remains unresolved. Extending the search for counterexamples to significantly greater values of the conjecture’s six integer parameters is a task ideally suited to the use of an SIMD parallel algorithm implemented on a GPGPU platform.

In 2009 researchers developed such an algorithm, implemented in the C programming language with CUDA extensions for execution on the NVidia GeForce 8400GS GPGPU. In the concluding remarks accompanying the report of their results, the researchers suggested that in the future the practical search range for their algorithm might be extended by implementing it as a distributed application across multiple GPGPUs.

This thesis project is one of a pair of related thesis projects aimed at developing a heterogeneous NPACI-Rocks/MPI/CUDA distributed multi-GPGPU application for seeking counterexamples to Beal’s Conjecture. In particular, this thesis project comprised the development and testing of the MPI-based communications and task-management layer and MPI/CUDA integration for the application on a commodity cluster host with NVidia GeForce 8400GS GPGPU-enabled compute nodes.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT ................................................................. iv</td>
</tr>
<tr>
<td>LIST OF FIGURES ......................................................... vii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS .................................................... viii</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1 INTRODUCTION .......................................................... 1</td>
</tr>
<tr>
<td>1.1 Choosing the Parallel Programming Model Implementation .......... 1</td>
</tr>
<tr>
<td>1.2 Data Parallel Model .................................................. 2</td>
</tr>
<tr>
<td>1.3 Shared Memory Programming Model ................................ 2</td>
</tr>
<tr>
<td>1.4 Message Passing Model ............................................. 2</td>
</tr>
<tr>
<td>2 MPI ARCHITECTURE ..................................................... 3</td>
</tr>
<tr>
<td>2.1 Introduction to MPI .................................................. 3</td>
</tr>
<tr>
<td>2.2 MPI Features ........................................................... 3</td>
</tr>
<tr>
<td>2.3 Popular MPI Implementations .................................... 4</td>
</tr>
<tr>
<td>2.4 MPICH2 Versus OpenMPI and LAM/MPI ............................ 4</td>
</tr>
<tr>
<td>3 MPI CONCEPTS ........................................................... 5</td>
</tr>
<tr>
<td>3.1 MPI Concepts ......................................................... 5</td>
</tr>
<tr>
<td>3.2 MPI Functions ......................................................... 6</td>
</tr>
<tr>
<td>3.3 Multi-Purpose Daemon(MPD) ...................................... 6</td>
</tr>
<tr>
<td>3.4 MPICH2 Process Manager .......................................... 6</td>
</tr>
<tr>
<td>3.5 MPICH2 Communication device .................................... 7</td>
</tr>
<tr>
<td>4 NPACI ROCKS/MPI INSTALLATION OVERVIEW .................... 8</td>
</tr>
<tr>
<td>5 MPI STARTUP METHODS ................................................. 9</td>
</tr>
<tr>
<td>5.1 Start MPD as the Root User ....................................... 9</td>
</tr>
<tr>
<td>5.2 Start MPD as a Non-Root User .................................... 9</td>
</tr>
<tr>
<td>6 TESTING MPD SETUP .................................................. 11</td>
</tr>
<tr>
<td>6.1 Testing Individual Nodes of the Cluster ......................... 11</td>
</tr>
<tr>
<td>6.2 Testing all the Nodes in the Cluster ............................. 12</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.1.</td>
<td>Cluster setup.</td>
<td>8</td>
</tr>
</tbody>
</table>
I would like to thank my thesis chair Mr. William Root for accepting me as a thesis student and guiding me in my work. I would also like to thank Dr. Carl Eckberg and Dr. Robert Grone for joining my thesis committee and helping me reach my goal of graduating with a master’s degree. Finally, I thank my family for supporting me in achieving my goal of earning a master’s degree.
CHAPTER 1

INTRODUCTION

Beal's Conjecture asserts that if $A^x + B^y = C^z$ for integers $A, B, C > 0$ and integers $x, y, z > 2$, then $A$, $B$, and $C$ share a common prime factor. Norvig [1] and others have shown that Beal’s Conjecture holds for all $A, B, C, x, y, z < 1000$, but the truth of the general conjecture remains unresolved. Extending the search for counterexamples to significantly greater values of the conjecture's six integer parameters is a task ideally suited to the use of an SIMD parallel algorithm implemented on a GPGPU platform. In 2009, J. Chauhan [2] (SDSU) developed such an algorithm, implemented in the C programming language with CUDA extensions for execution on the NVIDIA GeForce 8400GS GPGPU. In the concluding remarks of his thesis Chauhan suggested that future researchers might extend the practical search range for his algorithm by implementing it as a distributed application across multiple GPGPUs.

This thesis project is one of a pair of related thesis projects aimed at developing a heterogeneous NPACI-Rocks[3]/MPI/ CUDA[4] distributed multi-GPGPU application for seeking counterexamples to Beal’s Conjecture. In particular, this thesis project comprised the development and testing of the MPI-based communications and task-management layer and MPI/CUDA integration for the application on a commodity cluster host with NVidia GeForce 8400GS GPGPU-enabled compute nodes.

1.1 CHOOSING THE PARALLEL PROGRAMMING MODEL IMPLEMENTATION

We propose to implement the algorithm to find counterexamples to Beal’s conjecture as a distributed parallel CUDA implementation on a heterogeneous NPACI-Rocks multi-GPGPU cluster using MPI-based communications for inter-process communication and task distribution, dividing the search range among all the nodes of the cluster.

Unlike sequential computing on single-processor computers, where the Von Neumann model serves as a universal programming model, parallel computing lacks a single universal programming model. Several parallel programming models [5] are in common use.
The most widely used ones used are the data parallel model, the shared memory programming model and the message passing model.

### 1.2 DATA PARALLEL MODEL

The data parallel model is easy to write and comprehend. A set of tasks perform the same operation collectively on the same data, however, each task works on a different subset of the data. Data distribution must be explicitly defined for all tasks before processing commences. The associated compilers generate code for supporting communication and synchronization between the various tasks. High Performance Fortran (HPF), an extension of Fortran 90, is an example of a language supporting data parallel programming.

### 1.3 SHARED MEMORY PROGRAMMING MODEL

In the shared memory programming model, often implemented as multithreaded programs, all tasks share a common address space which they read and write asynchronously. Manipulating shared data requires synchronization, and tasks coordinate explicitly by synchronization operations on shared variables – writing and reading flags, locks or semaphores. Programs using this model are easy to write since there is no notion of data ownership and hence no need to specify explicitly the communication of data between tasks. An important disadvantage is that it is not readily scalable, because of synchronization/coherency issues. OpenMP and POSIX Threads implement the shared memory programming model.

### 1.4 MESSAGE PASSING MODEL

In the message passing model, the computation is broken down into a set of tasks each with its own local memory. Multiple tasks reside either on the same machine or across a number of machines. Data is exchanged among the tasks by sending and receiving messages.

One implementation of the message passing model, Message Passing Interface (MPI) is an industry standard for the message passing model and is widely used on HPC (High Performance Computing) platforms. We chose MPI for this project because it was the best fit for the GPGPU commodity cluster. The data parallel model and the shared memory model will not scale effectively on the cluster.
CHAPTER 2

MPI ARCHITECTURE

2.1 INTRODUCTION TO MPI

The primary challenge in parallel programming is determining how to identify parallelizable portions of the solution algorithm so that different subsets of the solution computations can be executed concurrently on different machines and the coordination between the different machines to send back the final results.

MPI (Message Passing Interface) [6] is a standard library of functions used by processes to communicate with each other by sending/receiving messages among on parallel/distributed computers or clusters. MPI-2 is the most recent available MPI standard. MPI was specifically designed for programming SPMD (Single Program Multiple Data) and MIMD (Multiple Instruction Multiple Data) applications.

In the MPI parallel programming model, the same parallel code executes on all threads of all machines. Each thread is assigned a unique “rank” which enables it to differentiate itself from other threads and defines the portion of the problem’s computation for which that node is responsible. When computations are complete, all threads send their results to a master thread designated as the receiver.

2.2 MPI FEATURES

Some of the important features of MPI are:

1. Standardization: MPI follows a widely-supported formal standard and is supported on virtually all HPC platforms.
2. Portability: MPI is easy to port. There is no need to modify the source code when we have to port our application to a different platform that supports the MPI standards.
3. Availability: A variety of MPI implementations are available, both commercial vendor and open source implementations.
4. Functionality: Over 115 routines are defined in MPI-1 alone.
5. Stable and mature: The latest MPI version is MPI-2.
6. User interface: MPI’s user interface is efficient and simple.
2.3 Popular MPI Implementations

Popular MPI implementations include LAM/MPI, Open MPI & MPICH.

1. LAM/MPI (Local Area Multicomputer/Message Passing Interface) requires booting up an LAM environment by starting a daemon on a node, running the code, and then shutting down the daemon when completed. Its currently in a maintenance mode and not actively supported by the user community.

2. Open MPI is an almost fully open source MPI-2 implementation, combining technologies from several other implementations including LAM/MPI. Open MPI is developed and maintained by a consortium of academic, research, and industry partners.

3. MPICH is available for most flavors of Unix, Linux, Mac OS X and Microsoft Windows. The original implementation of MPICH is called MPICH1 which implements the MPI-1.1 standard. The latest implementation is MPICH2 which implements the MPI-2.0 standard.

2.4 MPICH2 Versus OpenMPI and LAM/MPI

Some of the major differences between MPICH2 and OpenMPI include:

1. OpenMPI can only be run on shared memory architectures, whereas MPICH2 runs on either shared or distributed memory architectures.

2. OpenMPI is most appropriate for problems in which the primary parallelization paradigm is loop parallelization, whereas MPICH2 is appropriate for a wider variety of problems.

3. OpenMPI requires a compiler that explicitly supports OpenMPI; MPICH2 has no such restriction.

4. MPICH2 programs can be harder to debug as compared to OpenMPI programs.

Another major difference between MPICH2 and LAM/MPI: unlike LAM/MPI, in MPICH2 the mpd (multi-purpose daemon) daemon can be started as root user, and user profiles can be configured to have programs attach to and run as the root daemon; therefore there is no need to start up and tear down an environment before running a parallel program.

Considering all of the above comparisons, we chose MPICH2 for this project.
CHAPTER 3

MPI CONCEPTS

3.1 MPI CONCEPTS

There are 4 basic components of a MPI message passing implementation:

1. Communicator: A set of processes that are allowed to communicate between themselves. All processes are by default initially enrolled in a universe called MPI_COMM_WORLD, and each process is given a unique rank, a number from 0 to p-1 where p is the number of processes. Conventionally a process with rank=0 is usually designated the “master process.”

2. Rank: The rank is an integer defined within the context of a communicator and is used by the programmer to specify the sources and destinations of messages. Rank is often used in conditional expressions that control program flow.

3. Messages sending and receiving via buffers: sending and receiving messages are the two fundamental operations in MPI. One process executes a Send command, and another process executes a Receive command, to send a message unit from the first process to the second.

4. Messages: A “message” is the basic unit of communication, and consists of two parts: a data content part, and an “envelope” part that specifies routing from the sending process to the receiving process.

   The data content part has three parameters: start-buffer (address of the send-buffer), number of elements in the send-buffer, and data type of each send-buffer element.

   The envelope component also has three parameters: the rank of the destination/receiving process, a message tag which allows the message to be distinguished from other messages within the context of a communicator, and the communicator “handle.”

```
A Message Unit => startbuf,count,datatype, dest,tag,comm
                  \            | /          \       |       /
                \---DATA--/       ENVELOPE
```

---END OF PAGE---
3.2 MPI FUNCTIONS

There are 6 basic MPI functions which can be used to write parallel programs

1. MPI_Init : Initialize MPI execution environment
2. MPI_Comm_size : Find out how many processes there are in the communicator
3. MPI_Comm_rank : Find out which process I am in the communicator
4. MPI_Send : Send a message from one process to another
5. MPI_Recv : Receive a message from one process to another
6. MPI_Finalize : Terminate MPI

The destination and source specified in a send-and-receive operation always refer to the rank of the corresponding process in the group identified by a communicator.

3.3 MULTI-PURPOSE DAEMON(MPD)

The mpd is essentially a combination of daemons and associated auxiliary programs residing on the machines hosting the MPI program. These daemons collectively form a process management system for executing parallel jobs including MPICH jobs. Since even a single mpd on a single machine can form a self-loop, each machine must be configured in such a way that all mpds can connect to each other and pass messages via sockets.

3.4 MPICH2 PROCESS MANAGER

MPICH2 is designed to work with multiple process managers. The default is starting the MPICH2 jobs using the MPD process manager using the mpiexec command.

An interface called PMI (process management interface) isolates the MPICH2 library code from the process manager.

A process manager can also be explicitly chosen at configure time by adding --with-pm=mpd to the configure arguments (default is mpd)

Four process managers are distributed with MPICH2:

1. mpd (multi-purpose daemon) : This is the default process manager.
2. smpd : It can be used for both Linux and Windows and is the only process manager that supports the Windows version of MPICH2.
3. gforker : This is a simple process manager that creates all processes on a single machine. It is useful for both debugging on shared memory multiprocessors.
4. hydra(new) : This is a process manager, new in MPICH2 v1.3, that natively uses the existing daemons on the system such as ssh, slurm, pbs
3.5 MPICH2 Communication Device

MPICH2 can be built with many different communication devices allowing the programmer to tune the implementation for different communication fabrics. The default is “ch3”.

The communication methods can be specified by providing the name of the method after a colon in the --with-device configure option.

The communication methods are:

1. ch3:nemesis is a high performance method which uses shared-memory to send messages between processes on the same node and the network for processes between nodes. The supported networks are sockets and Myrinet-MX.
2. ch3:sock uses sockets for communications between processes. For platforms that are not supported by nemesis, the ch3:sock method can be used.
3. ch3:shm uses shared memory and only works within a single SMP
4. ch3:ssm uses sockets between nodes and shared memory within a node
CHAPTER 4

NPACI ROCKS/MPI INSTALLATION OVERVIEW

NPACI Rocks 5.3 Cluster installs CentOS as the cluster operating system on all the cluster nodes. As part of the Rocks Cluster installation, MPI-2 and Open MPI are installed when the HPC roll is selected. All MPICH2 binaries on CentOS are under /opt/mpich2-gnu/bin.

Our Cluster hardware setup (Figure 4.1) consisted of 3 Dell OptiPlex GX270 servers each with 1 Intel P4 processor 800MHz, 1GB RAM and 1 HDD of 72 GB. One server functioned as a ‘frontend’ server which is the master node and the other 2 servers served as the ‘compute’ nodes. The domain names of the server were set to the defaults ie ‘.local’ ie frontend.local, compute-0-0.local and compute-0-1.local.

![Cluster setup diagram](image)

**Figure 4.1. Cluster setup.**

Both the compute nodes have 1 NVIDIA GeForce 8400 GS with 512MB DDR2 inbuilt memory. The frontend server does not have a GPU card.

All cluster nodes are connected via a single network interface to a 5-port 10/100 Mbps Ethernet switch ports and connected to a monitor via a 4-port Belkin KVM switch.
CHAPTER 5

MPI STARTUP METHODS

MPD daemon can be started under the profile of a non-root user who can start and use their own individual mpd rings, or they can be started by the root user.

One of the advantages of starting mpd daemon as root user is that it can support multiple users simultaneously.

Our implementation had the mpd daemons started under the root user profile. We created a non-root user called ‘cudauser’ on all cluster nodes with the same profile and all MPI/CUDA programs were run under the ‘cudauser’ environment.

5.1 START MPD AS THE ROOT USER

The mpdroot program is installed by default in the /opt/mpich2/gnu/bin directory with setuid-root permissions. The configuration file /etc/mpd.conf should contain one line

secretword=<secretword>

where <secretword> is your password

To use the root’s ring, set environment variable MPD_USE_ROOT_MPD in the current environment or in the .mpd.conf under the user home directory to a value of 1 as shown below:

MPD_USE_ROOT_MPD=1

5.2 START MPD AS A NON-ROOT USER

For non-root users to start and use their own rings, create a file called .mpd.conf in the user’s home directory who needs to start the mpd daemon.

Set the permissions of the .mpd.conf file so that only that user can read and write this file as shown below:

chmod 600 .mpd.conf
Enter a secret word (password) into the .mpd.conf file:

```
secretword=mysecretword
```

The mpd daemon looks in the user home directory in the .mpd.conf file containing the line

```
secretword=<secretword>
```

Add /opt/mpich2/gnu/bin to the PATH environment variable in the .bash_profile file

```
export PATH= /opt/mpich2/gnu/bin:$PATH
```

All these steps are done just once, not every time we wish to compile/run an MPI job.
CHAPTER 6

TESTING MPD SETUP

Testing the MPD setup involves 2 general steps, First testing the individual nodes of the cluster as separate entities and Secondly testing the entire cluster with its nodes as an single entity. All the tests were run under the ‘cudauser’ profile.

6.1 TESTING INDIVIDUAL NODES OF THE CLUSTER

Individual nodes of the cluster can be tested using the mpdcheck and mpdtrace commands

1. Using mpdcheck command

On each node of the cluster, run mpdcheck as a server in one window. MPD uses client-server communications to perform work

For example, running the below command on frontend server would generate an output:

$ mpdcheck -s

server listening at INADDR_ANY on: frontend.local <port>

Running mpdcheck as client on frontend server in another window to check if client and server can communicate with each other generate a similar output:

$ mpdcheck -c frontend.local <same port as above>

The server output would look like:

server has conn on <socket._socketobject object at 0x40200f2c> from
(‘10.1.1.1,<some_port_number>)

server successfully recvd msg from client:
hello_from_client_to_server
The client will print this output:

client successfully recvd ack from server: ack_from_server_to_client

Repeat the above steps for all the nodes in the cluster.

2. Using mpdtrace command

We will bring up a ring of one mpd on one node say on the frontend server, test it and the bring the ring down using the following commands below.

```bash
mpd &  # start mpd in the background

mpdtrace  # displays hostname of the machine

mpdtrace -l  # displays hostname and port of the machine

eg

frontend.local_46147 (10.1.1.1)

mpdallexit  # shut down mpd daemon
```

Run a non-MPI program using the daemon on the same node.

```bash
mpd &
mpiexec -n 1 /bin/hostname  # print local server hostname
mpdallexit
```

The above steps should be repeated for all the nodes in the cluster. We should only proceed when the tests run successful on all nodes in the cluster.

### 6.2 Testing all the Nodes in the Cluster

As the root user, to bring up a ring of mpd’s on a set of cluster nodes, we create a file in root home directory named mpd.hosts and put the fully qualified hostnames of all cluster nodes in it:

```bash
frontend.local
compute-0-0.local
```
compute-0-1.local

From the master node ie frontend.local server, we ping all the cluster nodes to verify that they are alive and reachable.

We should check that we are able to ssh to each node from itself and from each node to each other node in the cluster.

We use the mpdboot command to start the daemons on all the cluster nodes defined in the file mpd.hosts:

```
mpdboot -n <number of mpd deaemons to start>  -f  mpd.hosts
```

mpdboot command will only start one mpd per machine even if the machine name appears in the hosts file multiple time

Cluster nodes can be tested using the mpdcheck, mpdtrace, mpdring commands

1. Using mpdcheck command

We run the mpdcheck command which will show if the frontend is having trouble discovering information about compute-0-0 and compute-0-1.

```
#mpdcheck -f mpd.hosts -v
obtaining hostname via gethostname and getfqdn
gethostname gives  frontend.local
getfqdn gives  frontend.local
checking out unqualified hostname; make sure is not "localhost", etc.
checking out qualified hostname; make sure is not "localhost", etc.
obtain IP addrs via qualified and unqualified hostnames; make sure other than 127.0.0.1
gethostbyname_ex:  ('frontend.local', ['frontend'], ['10.1.1.1'])
gethostbyname_ex:  ('frontend.local', ['frontend'], ['10.1.1.1'])
checking that IP addrs resolve to same host
now do some gethostbyaddr and gethostbyname_ex for machines in hosts
```
We can also run mpdcheck to test the ssh support between frontend and the 2 compute nodes as below:

```bash
mpdcheck -f mpd.hosts -ssh
```

2. Using mpdtrace command

Running ‘mpdtrace –l’ on frontend node will output the hosts & ports of all the cluster nodes where MPD daemons are running in the form <host>_<port>:

eg:

```bash
$ mpdtrace –l
```

```bash
frontend.local_46147 (10.1.1.1)
compute-0-1.local_48109 (10.1.255.253)
compute-0-0.local_55729 (10.1.255.254)
```

3. Using mpdring command

We can check how long it takes for a message to circle the ring of mpd’s by running the below command:

```bash
mpdringtest <number of times to go around the ring>
```

eg:

```bash
# mpdringtest 1000
```
time for 1000 loops = 0.84285902977 seconds

# mpdringtest 10000

time for 10000 loops = 7.90956902504 seconds

We can test that the ring can run a multiprocess job by running the mpiexec command:

mpiexec -l -n 30 /bin/hostname

4. NPACI Rocks Test script

NPACI Rocks installation provides a test script which can be run to test the mpd setup.
Create a file on frontend under the cudauser home directory named machines, and put the
hostnames of the compute nodes in:

compute-0-0
compute-0-1

Compile a test program using the MPICH environment:

$ cd $HOME
$ mkdir mpich-test
$ cd mpich-test
$ cp /opt/mpi-tests/src/mpi-ring.c .
$ /opt/mpich/gnu/bin/mpicc -o mpi-ring mpi-ring.c –lm

Now launch the job from the frontend:

$ ssh-agent $SHELL
$ ssh-add
$ /opt/mpich/gnu/bin/mpirun -nolocal -np 2 -machinefile \\
   $HOME/machines $HOME/mpich-test/mpi-ring

We are ready to run MPI jobs if all above steps are successful
CHAPTER 7  

MPI TROUBLESHOOTING  

7.1 MPD TROUBLESHOOTING STEPS  

We can do the below as further troubleshooting steps if needed:  

i. Check if one or more of your hosts are having trouble communicating with the other nodes due to firewall issues:  

   `iptables -L`  

ii. Check `/etc/hosts.deny` file to see if the sshd daemon is blocked (mpdboot uses ssh)  

iii. Check if free ports are available via `cat proc/sys/net/ipv4/ip_local_port_range`  

iv. If we are unable to start the daemons via mpdboot, then we start the daemon manually on the frontend.local machine via:  

   ```  
   mpd &  
   mpdtrace -l  
   ```  

   Now we run the below on every cluster node:  

   ```  
   mpd -h <hostname> -p <port> &  
   ```  

   where the hostname is ‘frontend.local’ and port belong to the original mpd that you started on ‘frontend.local’  

From each machine after starting the mpd, we can do mpdtrace to see which machines are in the ring so far.  

7.2 IMPORTANT MPD COMMANDS NEEDED FOR TROUBLESHOOTING  

Some of the important commands include:  

- `mpd` start an mpd daemon  
- `mpdtrace` show all mpd’s in ring  
- `mpdboot` start a ring of daemons all at once
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpdringtest</td>
<td>test how long it takes for a message to circle the ring</td>
</tr>
<tr>
<td>mpdalexit</td>
<td>take down all daemons in ring</td>
</tr>
<tr>
<td>mpdcleanup</td>
<td>repair local Unix socket if ring crashed badly</td>
</tr>
<tr>
<td>mpdlistjobs</td>
<td>list processes of jobs (-a or --all: all jobs for all users)</td>
</tr>
<tr>
<td>mpdkilljob</td>
<td>kill all processes of a single job</td>
</tr>
<tr>
<td>mpdsigjob</td>
<td>deliver a specific signal to the application processes of a job</td>
</tr>
<tr>
<td>mpiexec</td>
<td>start a parallel job</td>
</tr>
</tbody>
</table>
CHAPTER 8

IMPLEMENTATION

8.1 HIGH LEVEL ALGORITHM

MPI_Init starts MPI and initializes the MPI Execution environment
MPI_Comm_rank gets the rank of each process id (rank is a number between 0 and nprocs-1)
MPI_Comm_size returns the size of the group associated with a communicator.

If the process is the Master process (rank=0)

- Total number of workers (nWorkers) = total number of communicator processes - 1
- Parse startRange & endRange values passed to the program into variables
- Read the GPU file to find out the total number of GPUs in the cluster
- Compute sub-range per GPU device ie (endRange-startRange+1)/nGPU’s
  and assign each GPU a sub-range
- Divide workers among the GPU if workers > GPU devices else each GPU
  at max will have 1 worker
- Split the sub-range further for each GPU based on the initial sub-range and
  no of workers assigned to the GPU
- Initialize the GPU queue array (2xnGPU dimensional array)
  first row of array holds total number of worker processes assigned to
  each GPU (worker pool of process per GPU)
  second row of array holds the number of worker processes currently used
  from the worker pool for that GPU
  each column of array represents a GPU for a single GPU system
  populate first row of the array. If nWorkers is not evenly divisible
  among all GPU's, 1st GPU gets extra workers

Start assigning ranges to all workers starting with startRange as initial starting
number
if current worker is the final worker,
    end range = endRange of the range passed to program
else
    end range = start of range + sub-range allocated to the worker
send start range id & end range id to each worker via MPI_Send routine

Receive status from worker. Each worker process will print the values if Counter
eamples to Beal Conjecture are found

If the process is not a Master process(rank>0)
    Receive the start range id and end range id from the Master process
    Find free worker by scanning each GPU queue i.e. if total worker process assigned <
total
    worker allocated, assign current range to worker,
    increment assigned worker count by 1
    Attach slave to GPU, Records device as the device on which the active thread executes
    the device code
    Call external function in beal.cu file that searches for Counterexamples to Beal's
    conjectures happens with the CUDA Code
    If counterexample found, print the x,y,z,A,B,C values
    If completed, send worker status back to the master

MPI_Finalize Waits for all processes to reach the function, closes TCP connections & cleans
up

8.2 ANNOTATED CODE WALKTHROUGH

#include "/opt/mpich2/gnu/include/mpi.h"
#include "/usr/local/cuda/include/cuda_runtime.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>

/***************************************************************************
***************
Name of function  : run_beal
Input Paramters   : start_worker_range : starting number of the worker range
end_worker_range   : end number of the worker range
Output Parameters : None
Description       : Finding a counter example to BEAL's Conjecture
***************************************************************************/

void run_beal();

#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2

/***************************************************************************
**********************************
Name of function  : main
Input Paramters   : argc : 2 parameters
argv : start_range : starting number of the range
end_range   : starting number of the range
Output Parameters : None
Description       : Start and End of the range is passed to the main function via command line.
Number of processes to be spawned is also passed via command prompt per
node.
Main function starts MPI and initializes the MPI Execution environment.
Based on the number of GPU's and number of workers/slave processes,
Master process(rankid=0) splits the range into sub-ranges per gpu. The sub-ranges are further subdivided into ranges based on the number of workers/slaves(rankid>0) assigned to the gpu.

to compute the beal conjecture

***************************************************************************
**********************************

int main(int argc, char *argv[])
{
    int rankid=0, numworkers=0, numtasks=0, dest=0, initial_range=0, final_range=0;
    int number_of_gpu_devices=0, device_number=0, number_of_gpu_cores=0;
    int startRange=0, endRange=0, range_size_per_gpu=0;
    int split_numworker_per_gpu=0, numworker_per_gpu=0,
    range_split_within_gpu_per_worker=0;
    int remaining_workers=0, start_worker_range=0, end_worker_range=0;
    int source=0, mtype=0, worker_status=0, assigned_gpu=0;
    int send_buf_count=1;
    int i=0, j=0, k=0;
    MPI_Status status;
    struct cudaDeviceProp device_properties;
    char hostname[1024];
    char gpufile[]="gpufile";
    char line[100];
    char *p;
    FILE *file;

    /*
     Starts MPI & Initialize the MPI environment. This routine must be called before any
other MPI routine is called

On exit from this routine, all processes will have a copy of the argument list
Forms the MPI_COMM_WORLD communicator and opens necessary TCP connection.

A communicator is a list of all connections between nodes

*/

MPI_Init(&argc, &argv);

//get the rank of each process id. rank is a number between 0 and numtasks-1
MPI_Comm_rank(MPI_COMM_WORLD, &rankid);

/*
Returns the size of the group associated with a communicator.
This function indicates the number of processes involved in a communicator
*/
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks < 2 ) {
printf("Need at least two MPI tasks. Quitting...
");
exit(1);
}

///////////
//Process with rankid=0 is designated as the MASTER
///////////
if (rankid == MASTER)
{
    hostname[1023]=\0;
    gethostname(hostname, 1023);
//Total number of workers = Total number of communicator processes -1
numworkers = numtasks-1;

printf("n=")
printf(\nl AM MASTER with rank=%d started on %s with %d

tasks, numworkers=%d
rankid, hostname,
 numtasks, numworkers);

startRange=atoi(argv[1]);  /* Giving input from command line */
endRange=atoi(argv[2]);    /* Giving input from command line */

//To find the total number of GPU devices for a node
//cudaGetDeviceCount(&number_of_gpu_devices);
//cudaGetDeviceProperties(&device_properties,0);
//number_of_gpu_cores=8 * device_properties.multiProcessorCount;
//printf("n no of cuda devices=%d, no of cuda cores= %d",
 number_of_gpu_devices,number_of_gpu_cores);
//printf("n
==\\n"):

//Read GPUfile to find out total number of GPU's in the cluster
file=fopen(gpufile,"r");
if (file != NULL)
{
 while(fgets(line,sizeof line, file) != NULL)
 {

//read gpufile
p=strtok(line,":");
p=strtok(NULL,":");
//printf("\n %s",p);
number_of_gpu_devices=number_of_gpu_devices+atoi(p);
}
close(file);
}
printf("\nTotal Number of GPU devices = %d \n",number_of_gpu_devices);

//Compute sub-range per GPU device. Each GPU device will be assigned a subrange to work on
range_size_per_gpu=ceil((endRange-startRange+1)/number_of_gpu_devices);

//Distribute workers among the GPU if workers > GPU devices else each GPU can at max have 1 worker
if (numworkers >= number_of_gpu_devices)
    numworker_per_gpu=(numworkers/number_of_gpu_devices);
else
    numworker_per_gpu=1;

//Split the sub-range further for each GPU based on the initial sub-range and no of workers assigned to the GPU
range_split_within_gpu_per_worker=range_size_per_gpu/numworker_per_gpu;

//print results if Master
printf("\n
range_size_per_gpu=%d,numworker_per_gpu=%d,range_split_within_gpu_per_worker=%d\n",
range_size_per_gpu,numworker_per_gpu,range_split_within_gpu_per_worker);
//Initializing the GPU queue array, 2xnGPU dimensional array
//First row holds Total number of worker processes assigned to each GPU (worker pool of process per GPU)
//Second row holds the current number of worker processes already used from the worker pool for that GPU
//Each Column represents a GPU, 1 GPU for a single GPU system

remaining_workers=numworkers;
for(i=0;i<=number_of_gpu_devices-1;i++)
{
    //printf("n within for lkookl rankid = %d",rankid);
    gpu_2_worker[0][i]=numworker_per_gpu; //Assign worker pool to GPU(i)
    gpu_2_worker[1][i]=0; //Current number of worker processes used for GPU(i)
    remaining_workers=remaining_workers-numworker_per_gpu;
}

//if numworkers is not evenly divisible among all the GPU-Rs, first GPU will get extra workers
if (remaining_workers>0)
{
    gpu_2_worker[0][0]=gpu_2_worker[0][0]+remaining_workers;
}

start_worker_range=0;
end_worker_range=0;
//Start assigning ranges to all the workers, start with startRange as the initial starting number
start_worker_range=startRange;
mtype=FROM_MASTER;

for (dest=1;dest<=numworkers;dest++)
{
    if (dest==numworkers) //if final worker, end range = endRange of the range passed on cmd prompt
        end_worker_range=endRange;
    else
        //end of range allocated to the worker = start of range + sub-range allocated to the worker
        end_worker_range=start_worker_range+range_split_within_gpu_per_worker-1;

    //&start_worker_range => initial address of send buffer
    //send_buf_count => number of elements in send buffer
    //MPI_INT => datatype of each send buffer element (handle)
    //dest => rank of destination /worker, tag => message tag
    //comm => communicator (handle) MPI_COMM_WORLD
    //Send start range id to each worker via MPI_Send routine
    MPI_Send(&start_worker_range,send_buf_count,MPI_INT,dest,mtype,MPI_COMM_WORLD);
}

//Send end range id to each worker via MPI_Send routine
MPI_Send(&end_worker_range,send_buf_count,MPI_INT,dest,mtype,MPI_COMM_WORLD);
start_worker_range=end_worker_range+1;
}

mtype=FROM_WORKER;
for (j=1;j<=numworkers;j++)
{
    source=j;
    //worker_status  => initial address of receive buffer
    //1              => maximum number of elements in receive buffer
    //MPI_INT        => datatype of each receive buffer element (handle)
    //source         => rank of source (integer)
    //tag            => message tag (integer)
    //MPI_COMM_WORLD => communicator (handle)
    //status         => status object
    //Receive status from worker. Each worker process will print the values if
counterexamples found

MPI_Recv(&worker_status,1,MPI_INT,source,mtype,MPI_COMM_WORLD,&status);
    printf("MASTER Received results from Worker with rank=%d\n",source);
}
} //I am master

///////////////
//I am slave, worker part
///////////////
if (rankid > MASTER)
{
    hostname[1023]="\0";
    gethostname(hostname, 1023);
printf("\n========================================================
====================")
printf("\n----> I AM WORKER with rank=%d started on %s\n",rankid,hostname);

mtype=FROM_MASTER;
//Receive start and end range from Master

MPI_Recv(&start_worker_range,1,MPI_INT,MASTER,mtype,MPI_COMM_WORLD,&status);

MPI_Recv(&end_worker_range,1,MPI_INT,MASTER,mtype,MPI_COMM_WORLD,&status);

//find an unassigned worker from GPU worker queue array. Find a unassigned worker by scanning
//each GPU queue, if unassigned /free worker available , assign current range to worker, increment
//assigned worker count by 1
k=0;
assigned_gpu=0;

while(k<=number_of_gpu_devices-1) { //scan all GPU worker queue
    if (gpu_2_worker[1][k] < gpu_2_worker[0][k]) //if total worker process assigned < total worker allocated
    {
        gpu_2_worker[1][k]=gpu_2_worker[1][k]+1; //assign another worker
        //process to the current worker range
assigned_gpu=k;  //get GPU device id
break;
}
k=k+1;  //check other GPU worker queue to find free slots

//attach slave to gpu, Records device as the device on which the active thread executes the device code
//printf("nAssign Worker=%d to GPU=%d on host=%s\n",rankid,assigned_gpu,hostname);
cudaSetDevice(assigned_gpu);

/*This is the external function that is called in the beal.cu file in order to evaluate the truth of Beal’s Conjecture for a given range of parameter values*/

printf("n=");  
printf("nWorker with rank=%d running CounterExample for Beal Conjecture , start_range=%d,end_range=%d\n",rankid,start_worker_range,end_worker_range);
//Search for Counterexamples to Beal–Rs conjectures happens with the Cuda Code  
//If counterexample found, print the x,y,z,A,B,C values  
run_beal(start_worker_range,end_worker_range);

worker_status=0;  //Worker Status

//send ack to master  
mtype=FROM_WORKER;
8.3 MPI-CUDA CODE COMPILATION AND EXECUTION

The file beal.cu contains the CUDA code used for finding counter examples to Beal’s conjecture. The file mpi.c contains the MPI code which invokes the cuda code and facilitates the communication between all the processes in the cluster. All files are staged and compiled under the cudauser home directory on the frontend server before they are synched with the other cluster nodes.
1. The environment variables for MPI and CUDA are set in the .bash_profile file of the cudauser for all cluster nodes including the frontend server. All MPI/CUDA scripts are placed under cudauser home directory on all nodes.

   export PATH= /opt/mpich2/gnu/bin:/usr/local/cuda/bin:$PATH
   export LD_LIBRARY_PATH=/usr/local/cuda/lib

Compile CUDA code using nvcc compiler. The option –arch=sm_11 needs to be specified otherwise an error ‘atomicadd undefined’ will be returned.

   nvcc -arch=sm_11 -c beal.cu

2. Compile the MPI code and link it with cuda object code:

   mpicc -o mpicuda mpi.c beal.o -lcudart -L /usr/local/cuda/lib -I \   /usr/local/cuda/include

3. Copy the MPI & CUDA program code on the frontend server to all the cluster nodes running the beal conjecture program via the scp command.

   scp * compute-0-1:/export/home/cudauser/code/beal_using_mpicuda
   scp * compute-0-0:/export/home/cudauser/code/beal_using_mpicuda

4. Check that the machinefile and the GPU file have the below entries:

   $ cat machinefile:
   frontend:1 ➞ Master process with rank=0 starts on frontend
   compute-0-0:10 ➞ <hostname>:<total no of process to start on that host>, rank=1-10
   compute-0-1:10 ➞ process with rank=11-20 will be started here

   $ cat gpufile:
   compute-0-0:1 ➞ <hostname>:<total no of gpu’s on the system>
   compute-0-1:1 ➞ compute nodes have 1 GPU each, frontend does not
have a GPU

Run the mpiexec command on the frontend server to start the program on all the compute
nodes as specified in the machine file:

```bash
mpiexec -machinefile machinefile -np <no_of_proc> mpicuda \ 
<start_range> <end_range>
```

where

- np is the total number of MPI processes spawned for all cluster nodes
- start_range and end_range are the search ranges for Beal Counterexample
CHAPTER 9

RESULTS

Searching for possible counterexamples to Beal’s Conjecture given the start range and end range as 3 and 50 with number of workers as 10 yields the below output.

I AM MASTER with rank=0 started on frontend.local with 11 tasks, numworkers=10
Total Number of GPU devices = 2
range_size_per_gpu=24, numworker_per_gpu=5, range_split_within_gpu_per_worker=4

I AM WORKER with rank=1 running CounterExample for Beal Conjecture, start_range=3, end_range=6
I AM WORKER with rank=2 running CounterExample for Beal Conjecture,
Worker with rank=5 running CounterExample for Beal Conjecture,
start_range=19,end_range=22

Worker with rank=4 running CounterExample for Beal Conjecture,
start_range=15,end_range=18

Worker with rank=3 running CounterExample for Beal Conjecture,
start_range=11,end_range=14
Note: Launching kernel with Grid dimensions 3 by (3*3).

Worker with rank=6 running CounterExample for Beal Conjecture,
start_range=23,end_range=26
Grid Size = 3 x 9
Block Size = 10 x 10 x 1

Worker with rank=7 running CounterExample for Beal Conjecture,
start_range=27,end_range=30
Done. Time elapsed = 1
Worker with rank=1 sending results to Master
Worker with rank=1 completed task

MASTER Received results from Worker with rank=1
Note: Launching kernel with Grid dimensions 3 by (3*3).
Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Note: Launching kernel with Grid dimensions 3 by (3*3).

Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Note: Launching kernel with Grid dimensions 3 by (3*3).

Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Done. Time elapsed = 1
Worker with rank=6 sending results to Master
Worker with rank=6 completed task

=================================

Worker with rank=8 running CounterExample for Beal Conjecture, start_range=31,end_range=34
Note: Launching kernel with Grid dimensions 3 by (3*3).

Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Done. Time elapsed = 1
Worker with rank=7 sending results to Master
Worker with rank=7 completed task

=================================

Worker with rank=9 running CounterExample for Beal Conjecture, start_range=35,end_range=38
Note: Launching kernel with Grid dimensions 3 by (3*3).

Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Worker with rank=10 running CounterExample for Beal Conjecture,
start_range=39,end_range=50
Done. Time elapsed = 2
Worker with rank=3 sending results to Master
Worker with rank=3 completed task
=================================================================
=====
Note: Launching kernel with Grid dimensions 3 by (3*3).
Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Note: Launching kernel with Grid dimensions 3 by (3*3).
Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Done. Time elapsed = 3
Worker with rank=5 sending results to Master
Worker with rank=5 completed task
=================================================================
=====
Note: Launching kernel with Grid dimensions 3 by (3*3).
Grid Size = 3 x 9
Block Size = 10 x 10 x 1
Done. Time elapsed = 3
Worker with rank=2 sending results to Master
Worker with rank=2 completed task
=================================================================
=====
MASTER Received results from Worker with rank=2
MASTER Received results from Worker with rank=3
MASTER Received results from Worker with rank=4
MASTER Received results from Worker with rank=5
MASTER Received results from Worker with rank=6
MASTER Received results from Worker with rank=7
Done. Time elapsed = 3
Worker with rank=4 sending results to Master
Worker with rank=4 completed task

MASTER Received results from Worker with rank=8
MASTER Received results from Worker with rank=9
Done. Time elapsed = 1
Worker with rank=8 sending results to Master
Worker with rank=8 completed task

Note: Launching kernel with Grid dimensions 11 by (11*11).
Grid Size = 11 x 121
Block Size = 10 x 10 x 1
MASTER Received results from Worker with rank=10
MASTER with rank=0 Exiting MPI code
Done. Time elapsed = 2
Worker with rank=10 sending results to Master
Worker with rank=10 completed task
CHAPTER 10

CONCLUSION AND FUTURE ENHANCEMENTS

10.1 CONCLUSION

We have demonstrated that the search for counterexamples to Beal’s Conjecture can be successfully implemented as a parallel/distributed CUDA C application hosted on a heterogeneous NPACI-Rocks cluster with GPGPU-capable compute nodes and an MPI-based communications and task-management layer.

10.2 FUTURE ENHANCEMENTS

1. Currently the total number of GPU’s in the cluster is calculated using a gpufile parsed from within the program. Functionality could be added to send a message to a process on each compute node to execute cudaGetDeviceCount and return the GPU count for that node.

2. Implement non-blocking MPI_Isend and MPI_Irecv functions. Deadlocks are avoided but we could implement code that uses an MPI_Wait call in conjunction with each send and receive operation.
REFERENCES


