CellPilot: Seamless communication within Cell BE and heterogeneous clusters

N Girard, J Carter, W B Gardner, G Grewal
School of Computer Science, University of Guelph, ON, Canada
E-mail: {ngirard, jcarter, gardnerw, ggrewal}@uoguelph.ca

Abstract. The Pilot library is targeted to novice scientific programmers within High Performance Computing. The CellPilot library extends the Pilot library to the Cell Broadband Engine processor and heterogeneous clusters. Using Pilot's process and channel abstractions, the CellPilot library can create a process on any of the processor types, both PPEs and SPEs, across the cluster. Communication is achieved by creating a channel between any two processes, and using the write/read channel functions in the participating processes. The CellPilot library uses MPI for the inter-node communication and the Cell SDK within a Cell node. All the architecture specific details of Cell communications are hidden from the user.

1. Introduction
The Cell BE processor by IBM, Sony and Toshiba offers tremendous speeds: the third fastest supercomputer is the Roadrunner, a Cell-based cluster located in Los Alamos, New Mexico, with a peak performance of over 1 petaflops [1]. The top 6 green supercomputers are Cell-based clusters, with 722.98 MFLOPS/W for first position [2].

However, Cell-based computing clusters available to the scientific research community can be underutilized due to the Cell's complex architecture and the difficulty programming for this environment. Programmers face small local storage on the accelerators, explicit address alignment for the Direct Memory Access (DMA) transfers and using several programming paradigms simultaneously in order to use the Cell cluster appropriately.

The Pilot library [3] is a new library for high-performance computing built on top of MPI, targeted to novice scientific programmers. The CellPilot library applies Pilot's concepts to the Cell BE processor and heterogeneous clusters. The goal of the CellPilot library is to have identical style communication, from a programmer's point-of-view, between any two types of processes within a Cell cluster. The CellPilot library hides the alignment of data for DMA transfers making it significantly easier to do communication on the cluster. Programmers will only need to work in a single paradigm when using the CellPilot library.

The following sections will give further details of the Cell BE processor, the Pilot library, the CellPilot library and the issues faced when communicating across a Cell cluster.

2. The Cell Broadband Engine processor
The Cell processor [4] is made up of 9 cores: one Power-PC Element (PPE) holds the operating system and is essentially the coordinator of the 8 Synergistic Processing Elements (SPEs), which are smaller processors used for accelerated computation and have a small local storage space of 256KB. These 9 cores are connected via a specialized high-bandwidth circular data bus called the Element
Interconnect Bus (EIB). Two Cell processors can be connected via their I/O elements to make a Cell blade. Typical high-performance Cell clusters are constructed using Cell blades and a few quad-core processors such as the Xeon processor. These types of clusters are hard to program due to their unconventional heterogeneous architectures.

2.1. The Cell Software Development Kit
The Cell Software Development Kit (SDK) offers an API for creating, building, simulating and testing Cell applications. The SDK includes many libraries to optimize the performance of vector and linear mathematics computation, Fast Fourier Transforms, and data communication.

2.2. The Data Communication and Synchronization library
The Data Communication and Synchronization library (DaCS), included in the Cell SDK, provides services to ease the development of applications on the Cell BE processor by providing a hierarchical topology of processing elements, resource and process management, and data communication services. However, there are limitations to the DaCS library. Communication between SPEs is not directly possible due to the hierarchical model of DaCS, and data transfers between nodes is not possible with the DaCS library without the use of an external library such as MPI.

2.3. Programming issues
Many of the Cell's inconveniences are due to its heterogeneous multi-core design. All data transfers within the Cell blade is done via DMA calls. For optimal transfer performance, the data to be transferred should be aligned to a quad-word. The communication between two different Cell nodes must be done via the PPE. Therefore three steps are needed for the communication between two SPEs of different nodes.

Another constraint related to coding for the Cell processor is the local storage room on the SPEs. Each SPE has only 256KB of local storage for the application code and data. Programmers must pay special attention not to exceed this limitation while dividing their application.

3. The Pilot library
Pilot [5] is a new library for high-performance computing built on top of MPI, adding a layer of abstraction to the communication calls. Pilot's structure follows the Communicating Sequential Processes' (CSP, [6]) process-channel paradigm and it has a very small set of functions. The communication calls follow C's syntax for fprintf and fscanf, first specifying the channel, then the format and finally the values or variables.

Pilot involves two phases to help the novice programmer put together a working application. The first phase is the configuration phase where the application architecture, comprising processes and channels, is defined. The second phase, execution, is where all the calculations and communication are done. Pilot comes equipped with an integrated deadlock detection tool which can detect a variety of deadlock situations during the application's runtime.

4. CellPilot
The CellPilot library's aim is to extend the Pilot library for the global communication of heterogeneous clusters including the Cell BE processor. Cell Pilot uses a single communication abstraction for all communication types between two processors anywhere across the cluster. With the CellPilot library, data can be transferred from one end of a heterogeneous cluster to another with a single fprintf- or fscanf-style call by each of the sender and receiver processes, whether they be situated on different nodes or if the process is a PPE, an SPE or a non-Cell process.

With the Cell SDK and the DaCS library, the communication between two SPEs from different nodes involves code on both SPEs and on their corresponding parent PPEs. The CellPilot library follows the same conceptual model as Pilot, with minor changes to accommodate the Cell architecture. Only a single function is needed to communicate between any two processes across the cluster.
The Pilot paradigm helps programmers organize their parallel programs by first designing the process-channel diagram which should help reduce common communication errors.

4.1. SPE process creation and deployment
A new process type is introduced to the CellPilot library, the SPE process. The SPE process is similar to a regular process except that it will be run on a physical SPE processor, is created using a different function that specifies a "program handle" for the code that will be run, must be explicitly started, and has the SPE local store's memory size restriction. We require that the SPE processes be explicitly started to allow for SPEs to be reused by multiple processes under the control of their PPE process. SPE processes run to completion, after which an SPE is available for reuse. Starting more SPE processes than the currently available SPEs is an error. That is, CellPilot does not attempt to schedule or swap SPE processes.

Channels are still created using the same method as with Pilot and can be created between any two process types across the cluster whether they are an SPE, a PPE or a process of a non-Cell node. The channels are used for communication in the same way as is done using the Pilot library.

Thus there are effectively five types of channels, depending on the locations of their endpoints, as seen in Table 1. The local communication is done via DMA transfers and MPI is used for the communication between nodes.

| Channel Type | Processes involved          |
|--------------|-----------------------------|
| Type 1       | PPE/non-Cell Remote PPE/non-Cell |
| Type 2       | PPE Local SPE               |
| Type 3       | PPE/non-Cell Remote SPE     |
| Type 4       | SPE Local SPE               |
| Type 5       | SPE Remote SPE              |

Table 1. Channel types.

4.2. Co-Pilot process
The CellPilot library makes use of a second MPI process started on each Cell node. This process, called the Co-Pilot process, is a fully automated process that relays information to the SPEs and coordinates the intra-Cell and inter-node communication for the SPEs. The Co-Pilot process handles the start-up and completion of the SPE processes.

The Co-Pilot process coordinates the communication within the Cell node. The SPE processes signal the Co-Pilot process of a read or write on a channel involving a PPE or another SPE process of that node. The Co-Pilot process passes information such as the format, the location of the process and the buffer address to the other process.

With the CellPilot library, communication between a Cell node and a non-Cell node within a heterogeneous cluster is possible with a read or write over a channel. The Co-Pilot process formats the variables and calls the appropriate MPI functions to communicate with another Co-Pilot thread of another Cell node or a process of a non-Cell node. The Co-Pilot process allows the seamless communication across a heterogeneous cluster.

5. Communication scenario
The most complicated communication pattern on a Cell-based cluster is the communication between the SPEs of two different nodes. To do this, the SPEs communicate through their parent PPEs and use MPI for the communication between the two different nodes. Figure 1 illustrates this communication process.
5.1. Explanation
In this communication scenario, the programmer wishes to send data from SPE0 of Node 0 to SPE0 of Node 1, as seen in Figure 1 above. Due to the hierarchical nature of the Cell, the data needs to be transferred through the PPEs of each corresponding node. Therefore, three separate transfer steps are coded: from SPE0 of node 0 to the PPE of node 0, from the PPE of node 0 to the PPE of node 1 using MPI, then from the PPE of node 1 to SPE0 of node 1.

5.2. Code comparison
The appendix shows the code for the communication of the scenario in figure 1 using the Cell SDK, the DaCS library and the CellPilot library. The Cell SDK and the DaCS library have roughly the same amount of code across four processes for the communication between two SPEs of different nodes. The CellPilot library only requires one line of code on each of the SPEs for the communication, hiding the details of the three separate transfers.

The communication set-up (sharing of memory location between PPE and SPE) for the Cell SDK and DaCS library has to be done each time these two processes need to communicate. The CellPilot library’s set-up is only done once (the creation of a channel object) at the beginning of the program during the configuration phase.

Table 2 shows the number of lines of code for a Cell SDK, DaCS and CellPilot implementation of a program where an array of 10 integers is sent from the PPE of node 0 to the SPE of node 0, then from SPE of node 0 to SPE of node 1, then from SPE of node 1 to PPE of node 1.

| Program Implementation | Lines of code |
|------------------------|---------------|
| Cell SDK               | 186           |
| DaCS                   | 114           |
| CellPilot              | 80            |

Table 2. Number of lines of code for a simple program implemented with the Cell SDK, DaCS library and CellPilot library.

6. Conclusion and future work
It can be seen from the contrasting code samples that Cell communication using CellPilot is much simpler to program than either the Cell SDK or the DaCS library. The Pilot paradigm helps programmers organize their parallel programs and avoid common communication errors. A programmer that is familiar with the Pilot library should not have difficulty using the CellPilot library. CellPilot is currently under development. Future work includes implementing Pilot’s collective communication functions and deadlock detection on the Cell.
7. References
[1] http://www.top500.org, June 2010.
[2] http://www.green500.org, November 2009.
[3] Carter J, Gardner W B, Grewal G, 2010 The Pilot Approach to Cluster Programming in C Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium, Workshops and Phd Forum (Atlanta) 8 pp.
[4] CBE programming tutorial ver 3.1, http://www.ibm.com/developerworks/power/cell/, 2008.
[5] http://carmel.cis.uoguelph.ca/pilot.
[6] Hoare C A R, 1978 Communicating Sequential Processes Communications of the ACM 21 (8) 666-677.

Appendix A. Cell SDK code
1  --- Sender SPE ---
2  unsigned int add = (unsigned int) spu_read_in_mbox();
3  int Array[100] __attribute__ ((__aligned__ (128))); //put data here
4
5
6
7  mfc_put(Array, add, sizeof(int)*100, 10, 0, 0); // size in bits
8  mfc_write_tag_mask(1<<10);
9  mfc_read_tag_status_all(); //wait for transfer to complete
10  spu_write_out_mbox((unsigned int)1); //notify PPE
11
12
13
14  --- Sender PPE ---
15  int Array[100] __attribute__ ((__aligned__ (128)));
16  int retVal;
17  unsigned int *add, data, temp;
18  spe_context_ptr_t ctx; //holds the SPE info
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
add = &data;
retVal = spe_in_mbox_write(ctx, add, 1, SPE_MBOX_ANY_NONBLOCKING);
while(retVal!=1); //retry until success
while(spe_out_mbox_status(ctx)); //wait for a message from SPE
spe_out_mbox_read(ctx, &temp, 1);

--- Receiver SPE ---
#include about 22 lines of code is needed to start the SPE process
unsigned int add = (unsigned int)spu_read_in_mbox();
int Array[100] __attribute__ ((__aligned__ (128)));
...
mfc_get(&Array, add, sizeof(int)*100, 31, 0, 0);
mfc_write_tag_mask(1<<31);
mfc_read_tag_status_all();
spu_write_out_mbox((unsigned int)1); //notify PPE

Appendix B. DaCS code
--- Sender SPE ---
uint64_t size;
int Array[100];
dacs_wid_t waitid;
dacs_remote_mem_t location;
...
//accept memory location
dacs_remote_mem_accept(DACS_DE_PARENT, DACS_PID_PARENT, &location);
//get memory size
dacs_remote_mem_query(location, DACS_REMOTE_MEM_SIZE, &size);
dacs_put(location, 0, &Array, size, waitid, DACS_ORDER_ATTR_NONE,
   DACS_BYTE_SWAP_DISABLE);
dacs_wait(waitid); //wait for transfer to complete
dacs_remote_mem_release(&location);

--- Sender PPE ---
de_id_t spe_de;
dacs_process_id_t spe_pid;
//2 more lines of code to start the SPE process
int Array[100];
dacs_remote_mem_t location;
...
dacs_remote_mem_create(&Array, sizeof(int)*100, DACS_WRITE_ONLY,
   &location); //create memory region
dacs_remote_mem_share(spe_de, spe_pid, location);
dacs_remote_mem_destroy(&location);
MPI_Send(&Array, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);

--- Receiver PPE ---
de_id_t spe_de;
dacs_process_id_t spe_pid;
//2 more lines of code to start the SPE process
int Array[100];
dacs_remote_mem_t location;
MPI_Status status;

MPI_Recv(&Array,100,MPI_INT,0,0,MPI_COMM_WORLD,&status);
dacs_remote_mem_create(&Array, sizeof(int)*100, DACS_READ_ONLY, &location); //create memory region
dacs_remote_mem_share(spe_de, spe_pid, location);
dacs_remote_mem_destroy(&location);

--- Receiver SPE ---
uint64_t size;
int Array[100];
dacs_remote_mem_t location;
dacs_wid_t waitid;

//accept memory location
dacs_remote_mem_accept(DACS_DE_PARENT, DACS_PID_PARENT, &location);
dacs_remote_mem_query(location, DACS_REMOTE_MEM_SIZE, &size);
dacs_get(&Array, location, 0, size, waitid, DACS_ORDER_ATTR_NONE, DACS_BYTE_SWAP_DISABLE);
dacs_wait(waitid); //wait for transfer to complete
dacs_remote_mem_release(&location);

Appendix C. CellPilot code
#include <stdio.h>
#include <pilot.h>

extern spe_program_handle_t spe_send, spe_recv;
PI_CHANNEL *betweenSPEs

--- Receiver PPE function ---
int recvFunc(int arg, void *ptr){
    PI_RunSPE(recvSPE, 0, NULL);
    return 0;
}

--- Sender PPE function ---
// (configuration phase also done by Receiver PPE)
int main(int argc, char *argv[]){
    int N = PI_Configure(&argc, &argv);
    PI_PROCESS *recvPPE = PI_CreateProcess(recvFunc, 0, NULL);
    PI_PROCESS *sendSPE = PI_CreateProcess(spe_send, PI_MAIN, 0);
    PI_PROCESS *recvSPE = PI_CreateProcess(spe_recv, recvPPE, 0);
    betweenSPEs = PI_CreateChannel(sendSPE, recvSPE);
    PI_StartAll();
    PI_RunSPE(sendSPE, 0, NULL);
    PI_StopMain(0);
    return 0;
}
--- Sender SPE ---
//contents of spe_send.c
PI_CHANNEL_SPE betweenSPEs
PI_SPE_PROCESS(int arg1, void *arg2)
    int Array[100], i;
    for(i=0; i<100; i++)
        Array[i]=i;
    PI_Write(betweenSPEs, "%100d", Array);
PI_SPE_END

--- Receiver SPE ---
//contents of spe_recv.c
PI_CHANNEL_SPE betweenSPEs
PI_SPE_PROCESS(int arg1, void *arg2)
    int Array[100], i;
    PI_Read(betweenSPEs, "%*d", 100, &Array);
    for(i=0; i<100; i++)
        printf("%d ", Array[i]);
    printf("\n");
PI_SPE_END