CellPilot: A Seamless Communication Solution for Hybrid Cell Clusters

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

Abstract—The CellPilot library provides a comprehensive inter-process communication solution for parallel programming in C on clusters comprised of Cell BE and other computers. It extends the process/channel approach of the existing Pilot library to cover processes running on Cell PPEs and SPEs. The same simple API is used to read and write messages on channels defined between pairs of processes regardless of location, while hiding communication details from the user. CellPilot uses MPI for inter-node communication, and the Cell SDK within a Cell node.

Keywords—Cell Broadband Engine; parallel programming; MPI; high-performance computing

I. INTRODUCTION

A high-performance cluster constructed of Cell Broadband Engine (Cell BE) [1] nodes, perhaps with other heterogeneous nodes, presents formidable difficulties for a programmer of ordinary skill and knowledge to utilize. While the popular message-passing library MPI [2] can handle inter-node communication on a hybrid cluster, MPI is not integrated with the communication libraries needed for intra-Cell use. The hardware resources include the relatively slow Cell Power Processor Elements (PPEs), the fast Synergistic Processor Elements (SPEs), and fast non-Cell cores, but splicing together a solution from MPI and the Cell’s own libraries that would facilitate deploying a cluster-wide application is daunting.

While it is true that not every HPC problem stands to benefit from the strengths of the Cell’s unique architecture, if the state of affairs exists such that an available Cell cluster is heavily underutilized while conventional clusters are crowded with users—precisely the situation in the HPC consortium associated with our university—it may be that the above challenges are at least partly to blame. Therefore, lowering the barriers for a would-be Cell cluster programmer to "get into the game" is a worthy goal.

In this paper, we present a work in progress called CellPilot, which gives HPC programmers an alternate communication model that applies seamlessly to all processor resources in a cluster of Cells and non-Cell nodes. CellPilot was first introduced in concept at HPCS 2010 (High Performance Computing Symposium) [3]. It is based on extending the Pilot library [4], originally designed as an easier method for novice scientific programmers to write parallel cluster applications in C and Fortran, onto the Cell, using Pilot’s same process/channel abstractions and its same economical, stdio-inspired API with easy-to-master fprintf/fscanf metaphor. With CellPilot, programmers design software in terms of processes that can be located on any PPEs, SPEs, or non-Cell nodes, and communications channels bound to process pairs. Programs are coded in terms of reading and writing on those channels, whereupon CellPilot transparently applies whichever communication mechanisms are required to transport the message, regardless of its endpoints. This gives the programmer a convenient unified view of all processor resources on the cluster, and a way to handle interprocess communication while avoiding low-level operations and multiple library APIs.

The Pilot library itself is a thin, transparent layer on top of standard MPI; that is, it uses MPI as the transport mechanism. CellPilot is an extension of Pilot that provides communication to and from SPE processes, with low-level communication carried out by means of IBM’s Cell Software Development Kit (SDK) functions.

This paper first presents the necessary background (Section II) on the Cell BE, its IBM-supplied communication libraries, the Pilot approach to cluster programming, and related work on Cell BE programming tools. Then the programming model provided by CellPilot is described in Section III, followed by the underlying implementation (Section IV) based on Pilot, MPI, and the Cell SDK. CellPilot is being used for several case studies through which performance data (Section V) can be obtained. Finally, the present status and availability of CellPilot is reported (Section VI), and the paper concludes.

II. BACKGROUND

A. Cell BE architecture

The Cell processor is made up of 9 RISC cores: One Power Processor Element (PPE) runs the operating system, typically Linux, and is the coordinator of the 8 Synergistic Processing Elements (SPEs) used for accelerated computing.

1The terms PPE and SPE seem to be used almost interchangeably with PPU (Power Processor Unit) and SPU (Synergistic Processing Unit), respectively. The reality is a bit more complicated, with the processor unit hardware being contained within the processing elements. For simplicity, this paper always refers to PPE and SPE.
tion with their SIMD instructions, each having a small local storage space of 256KB. These 9 cores are interconnected 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 handful of quad-core processors such as the Xeon, thus yielding a heterogeneous or hybrid cluster.

A key reason for the Cell’s high performance is the fact that each SPE can directly address only its own independent memory, thereby obviating the need for expensive and latency-producing cache coherency hardware. Each SPE’s local store is only 256KB for the application code and data. Programmers must pay special attention not to exceed this limit, and may need to divide up their application code accordingly, for which an overlay capability is available.

A by-product of this design is that programmers are compelled to move data explicitly between main memory and the SPES’ memories. The architecture provides a number of complex mechanisms to accomplish this, including DMA transfers and mailboxes, and some have stringent address alignment requirements. For optimal DMA performance, the data to be transferred should be aligned to a quad-word address.

SPE programs are launched when a PPE program creates a “context” associated with the desired executable (which was earlier embedded by a special linker into the PPE executable in the guise of initialized static data) and loads it onto an SPE under the control of a PPE POSIX thread that waits for the SPE program’s asynchronous completion. In this way, all SPEs can be kept busy computing in parallel.

Thus, learning to program an individual Cell node successfully can already be difficult enough without introducing the added complications of off-node communication. I/O between two different Cell nodes must be done via the PPE, thus for SPEs of different nodes to intercommunicate requires three hops involving two PPEs.

IBM provides libraries, described next, that at least allow a programmer to avoid coding assembly language instructions for data transfers.

B. Cell communication libraries

The Cell Software Development Kit (SDK) offers an API for creating, building, simulating and testing Cell applications. The SDK includes libraries in support of computation—vector and linear mathematics computation, Fast Fourier Transforms, access to SPE’s SIMD instructions at the C level—and data communication. One important library is the SPE Runtime Management Library (libspe2) which provides a low-level API for SPE management. Another is the Accelerated Library Framework (ALF), which offers a programming environment for data- and task-parallel applications.

The Data Communication and Synchronization library (DaCS), included in the Cell SDK, provides services to ease the development of applications on the Cell BE in terms of a hierarchical topology of processing elements. DaCS includes resource and process management, and data communication services. In the process hierarchy, the PPE is considered the Host Element (HE) and its associated SPEs are Accelerator Elements (AEs). There is limited support for collective operations, scatter and gather, between the PPE and a list of SPEs.

DaCS does have an extension, DaCS for Hybrid (DaCSH), that allows for off-node communication in a cluster. It effectively adds another layer to the process hierarchy, as illustrated in Figure 1. One non-Cell (x86-64) node is the HE for the cluster, and all Cell nodes (PPEs) are its AEs. These nodes are shown circled in the diagram. Each AE that resides on a PPE is also the HE of its own Cell, with its SPEs under its control as AEs. As will be seen below, this is somewhat similar to what CellPilot achieves using MPI.

However, there are limitations to the DaCS library. On the local level, direct communication between SPEs is not supported due to the strongly hierarchical model of DaCS, and DaCSH does not have the flexibility to encompass a cluster of arbitrary node architectures.

C. The Pilot approach

Pilot [4, 5] is a library for high-performance computing built on top of MPI, adding a layer of abstraction that follows the Communicating Sequential Processes (CSP) [6] process-channel paradigm. Pilot’s API has a very small set of functions. Its communication calls are modelled on C’s stdio syntax for fprintf and fscanf, specifying first the channel, then the data format and length, and finally the values or variables. For example, the PI_Write() call below sends an array of 1000 floating point numbers on the given channel:

```c
PI_CHANNEL *workerdata;
float data[1000];
...
PI_Write(workerdata,"%1000f",data);
```
It would be matched with a similar call to PI_Read() in the receiving process. The format is simply a convenient way to describe the data; it does not imply that the data is converted to text for transmission. And it need not be a string literal; it can be supplied by a variable.

Pilot applications execute in two distinct phases: The first is the configuration phase where the static application architecture, comprising processes and channels, is defined by calling PI_CreateProcess() and PI_CreateChannel(). The program can easily learn how many total Pilot processes can be created (=number of MPI processes specified by mpirun), which is necessary for writing scalable applications that utilize every available processor. Defining a process means pointing to a function for it to run during the execution phase. The same function body can be associated with multiple processes, and an index parameter can be passed so it can identify its own instance, very much in the same style as POSIX pthread_create(). This phase is concurrently executed by every MPI process in the cluster, resulting in the construction of equivalent internal tables on the various processors, regardless of their respective word length, data alignment, and endian properties. The execution phase commences as each process invokes its associated function, except for MPI rank #0, also known as PI_MAIN, which has no additional associated function and simply continues executing statements in the main() function.

During the execution phase, processes may write to and read from channels, which results in MPI messages being sent and received “under the hood.” The program ends when all processes return from their respective functions, and PI_MAIN executes PI_StopMain(). They all synchronize on an internal barrier before exiting.

Pilot has a special way of providing access to a selected subset of MPI’s collective operations. During the configuration phase, a set of channels having a common endpoint can be designated as a bundle to be used for a specific purpose by calling PI_CreateBundle(). Bundle operations supported as of V1.2 are: broadcast, gather, and select. The first two have the usual MPI sense, while the select operation blocks until some channel in the bundle has data ready to read (so that a read on the channel would not block). The nomenclature is meant to suggest a Unix/POSIX “select” operation on a set of file descriptors. Non-blocking operations are also available: checking whether a channel or a bundle has data to read.

The key difference between MPI’s collective operations and Pilot’s is that MPI follows a pure SPMD convention. For example, if one process is broadcasting to 50 others, all 51 must execute MPI_Bcast(), which is arguably counterintuitive given that 50 are actually receiving. With Pilot, in contrast, only the broadcasting process calls PI_Broadcast() on the bundle; the 50 receivers each call PI_Read() on their respective channels. Thus, Pilot follows an MPMD convention akin to pthreads.

Launching a Pilot application on a heterogeneous cluster is straightforward: keeping in mind that a Pilot application is just an MPI job, the programmer constructs an mpirun command (or whatever their site requires) that designates which nodes will run which executables. The first node will take on the identity of MPI rank #0 or PI_MAIN, then the rest of the nodes will become the other Pilot processes. MPI will take care of any conversions required between datatype lengths, endianness, and character codes.

Pilot comes equipped with an integrated deadlock detection tool. This feature consumes one MPI process and is enabled simply by coding the option “-pisvc=d” on the command line (i.e., mpirun...-pisvc=d). Errors such as circular wait will cause the program to abort with a diagnostic message identifying the deadlocked processes.

Pilot is available for downloading from its website [7], which also contains documentation and tutorials. Pilot is copyright by the University of Guelph and is not open source, but it can be used for free by anyone without any licensing formalities.

Benefits of the Pilot approach to cluster programming include the elimination of categories of common parallel programming errors, such as one process mistakenly sending a message to another process that is unprepared to receive it, or coding errors with MPI rank or tag numbers. After the programmer configures an application’s process/channel architecture, Pilot enforces that architecture at run time, prohibiting communication from taking place except over predefined channels, and reporting API misuses by source file and line number. With the optional activation of built-in deadlock detection, Pilot can also diagnose conditions that would cause mysterious program hangs using MPI alone.

D. Related work

Several researchers have created languages and libraries designed to ease programming of the Cell’s complex architecture. However, only one of these is directly applicable to a cluster of Cells; that is, an HPC programmer could choose a technique to apply on a per-Cell level, and then arrange for inter-node communication via some other mechanism such as MPI.

First, partial versions of MPI have been programmed for use on a Cell. The core functions of MPI-1 have been implemented using a buffered approach for small messages [8] and a synchronous approach for large messages [9]. The Cell Messaging Layer (CML) is an implementation of a small subset of MPI that is usable on a single Cell processor or on Cell clusters [10]. CML assigns MPI ranks to all available SPEs, but not to PPEs, which are reserved for use by the library to carry out inter-Cell communication by means of conventional MPI. Available operations are
MPI_Send and MPI_Recv, and the collective operations MPI_Bcast, MPI_Reduce and MPI_Allreduce, which are designed hierarchically. The limited implementation of these libraries made them infeasible candidates for CellPilot to build upon, since Pilot itself uses more of MPI. A key difference is that with CellPilot PPEs can host processes just like any non-Cell node.

MPI Microtask [11] allows microtasks, essentially virtual SPEs written by the programmer, to be created dynamically at run time and assigned to a function. An MPI-style interface is used for communication. The MPI Microtask preprocessor decomposes the microtasks into basic tasks consisting of units of computation bounded by communication events, and groups together the basic tasks with strong dependencies. The preprocessor also pre-computes runtime parameters such as message buffer addresses which helps reduce the overhead in the runtime system. CellPilot does not require a preprocessor, but simply provides a higher-level library interface for communication.

The Charm++ Offload API [12] allows Charm++ work requests, which are blocks of work that do not have any data dependencies with any other executing work requests, to be off-loaded to the SPEs. The Offload API can be run independently from Charm++ as it is simply an API that allows C/C++ programs to be executed on the SPEs. CellPilot uses a different approach to spawning blocks of work: They are packaged as C functions that become executing processes in the program’s configuration phase, with CellPilot automatically handling the off-loading of processes to SPEs.

The MultiCore Framework from Mercury [13] consists of a manager and multiple workers that communicate via “tiles” over channels. A distribution object, containing the entire data set and parameters of a portion of the algorithm, is used to create tile channels. Tiles are buffers that contain a small part of the data and are sent to the workers via the channels. Workers connect to a tile channel, receive a tile, perform computation on the data within the tile, replace the data with the results, and return the tile to the manager via the channel. CellPilot also depends on the channel abstraction, but does not provide a data organizational construct like tiles.

Cell Superscalar (CellSs) [14, 15] has the objective of providing a simple and flexible programming model by attempting automatic parallelism. The programmer needs to annotate functions that can be run independently in parallel. A source-to-source compiler converts the annotated code into a Cell-compatible program by exploiting function parallelism. A runtime library builds a dependency graph that exposes data dependencies among the annotated functions, does the task scheduling and handles the data transfers between the processors. CellPilot does not attempt automatic parallelization. The programmer is entirely responsible for the application’s parallelism.

The Cell processor was designed to accommodate different programming paradigms. The streaming model, useful for gaming and high-definition television, is supported by Multicore Streaming Layer (MSL) [16]. MSL is a general runtime framework that employs a dataflow graph—where each node is an actor that performs some computation, and edges express dependencies between input and output streams associated with those actors—for automatically determining pipeline or data parallelism within streaming applications. Since all data communication within CellPilot is blocking, it is not a good candidate for the stream programming model.

Coconut (COde CONstructing User Tool) [17] parallelizes a program written in Haskell for the Cell processor by using a graph to represent the program's data and control flow. The tool allows the programmer to manipulate the graph in order to create a high-performance schedule of the work to be done on the SPEs. Coconut uses formal methodologies to ensure that the parallel version of a program is equivalent to its serial version and to verify that the schedule created by the programmer is valid, i.e., independent of any eventual execution order. CellPilot programmers do not require any knowledge of formal methods. Instead, it was designed to inherently embody formal principles of CSP in its process-channel model.

Cellflow [18] is a programming toolkit offering both off-line and on-line facilities. Among the off-line facilities are a task allocator and scheduler that uses a Constraint Programming approach for optimizing allocation and scheduling using a task graph representing the application and hardware resources. A customizable application template allows programmers to identify task dependencies in their application and easily create a task graph. The on-line support includes a software library and high-level APIs which manage the communication and synchronization of the tasks using data queues, which are stored in a task table formed at start time; counters, which keep track of free slots in the data queue; and a series of semaphores, which are used for synchronization by signalling data transfers completion. In CellPilot, the SPEs are manually scheduled by the programmer during the execution phase and the CellPilot library does not attempt to optimize SPE scheduling.

BlockLib [19] is a Cell abstraction built atop the NestStep platform, which is a bulk-synchronous parallel (BSP) address space language supporting nested parallelism, which makes the user code more portable between NestStep enabled systems. BlockLib provides skeleton functions for basic computation patterns and automates much of the memory management and parallelization. The library uses a small macro language to allow users to access SIMD optimizations without having to do hand optimization. Synchronization and inter-SPE communication are available outside of the BSP model. Synchronization is achieved using signals, and all transfers are double buffered and 128-
byte aligned where possible. Synchronization is implicit within CellPilot as all communication is blocking, and there is no explicit way to synchronize processes such as a barrier operation.

StarPU [20] is a high-level framework that abstracts heterogeneous processors such as Cell BE and GPGPU (general-purpose computing on graphics processing units), allowing applications to be written without needing to worry about the underlying hardware specific details. CellPilot extends the Pilot library by adding SPE functionality. Therefore it is only applicable to Cell systems. The Pilot library can still be used, with the exclusion of the SPE features, on MPI-enabled clusters.

In terms of programming a single Cell computer, it can be said that CellPilot is not as ambitious as some of the above tools. The initiative for identifying code to run on SPEs is left in the programmer’s hands, and CellPilot’s basic contributions are easy methods to launch SPE processes and to handle interprocess communication. However, all of the above techniques, along with their greater ambition also have their significant learning curves, whereas with CellPilot, the programmer is coding in C as usual. Then, in the cluster context, CellPilot’s special contribution of cluster-wide communication and coordination applies.

III. CELLPILOT PROGRAMMER’S MODEL

Simply put, the design objective of CellPilot was to allow the Cell’s SPEs to participate as “equal citizens” as sites for processes and for channel-based communication in the Pilot programming model. If a programmer has already learned how to use Pilot on a conventional cluster, learning a couple more API functions for the SPE is a small matter.

During the configuration phase, which is executing on the various nodes’ PPEs, each function that is intended to be run as an SPE process is defined with a PI_CreateSPE() call. The difference from the usual PI_CreateProcess() is that SPE processes are not automatically launched in the execution phase, as are all other Pilot processes. Rather, they must be explicitly launched by the process “in charge of” that Cell node (i.e., the PPE process) during its own execution phase. This is completely in keeping with the idea that SPEs have limited memory and may need to be loaded and reloaded with codes to perform small computations that are part of the overall application. It establishes a kind of process hierarchy, in that each set of SPE processes is controlled by its local PPE-based “parent” process. This is similar to the hierarchy of DaCSH, except that channels can directly route communication between processes at any level, as shown in Figure 2. The various “types” of channels, which are transparent to the programmer, are explained in Section B.

In this way, all Pilot processes are globally defined throughout the application and are candidates for binding to channels. Channel variables are typically global to the program, so that they can be initialized in main() and referenced in process functions. SPE processes can use the __ea “effective address” attribute to refer to globals in main memory, and the compiler will generate the necessary linkage code to resolve the address at run time. Thus, SPE processes can refer to channels and arrays of channels symbolically. Furthermore, only their address is utilized on the SPE, not the contents of the PI_CHANNEL data structure in main memory, which would require much more overhead to extract.

IV. CELLPILOT IMPLEMENTATION

The two key implementation issues concern the control of SPE-resident processes and the carrying out of channel communication across every combination of local vs. remote PPE or SPE, and off-node processes. These are described in the sections below.

A. SPE processes

Just like regular Pilot processes, which are equivalent to MPI ranks, SPE processes are defined during the program’s configuration phase by calling PI_CreateSPE(). Instead of pointing to a C function to run (on the PPE), SPE processes are associated with an spe_program_handle_t (SDK typedef), which is an external symbol associated by the linker with some SPE object code that is embedded into the PPE executable file in the guise of static data. (This typedef is referred to by the macro PI_SPE_FUNC so that CellPilot configuration code can also compile on non-Cell nodes.) Such processes remain dormant until PI_StartSPE() is called by the “parent” PPE Pilot process. At that time, CellPilot takes care of spawning a pthread to load the code into an SPE by means of SDK functions, and waiting (in the background) for its completion.

Prior to that, the programmer will have coded and compiled the SPE processes. Two cellpilot.h macros, PI_SPE_PROCESS(int, void*) and PI_SPE_END (see example in Figure 4), are used to bracket an SPE pro-
cess function. The first one hides the code that transfers the arguments provided by PI_StartSPE(...,int,void*). Such arguments are especially useful when starting multiple instances of the same process function in data parallel programming, e.g., to give each instance a different index number or other parameter. The second macro hides the code that terminates the SPE program.

B. CellPilot channels

Of the existing SDK libraries, DaCS is closest to what CellPilot wants to offer on the local node level, and can be a partial replacement for MPI on the cluster level. At least in part, CellPilot could have used DaCS in a similar manner to how Pilot uses MPI. However, investigation determined that DaCS (a) was not a comprehensive solution—in particular, it did not address SPE-to-SPE communication nor support collective operations across the cluster; and (b) linking it into every SPE program would simply take up precious space from the 256K limit. So we decided to use only the basic functions in libspe2 to implement CellPilot, while retaining Pilot’s relationship with MPI, and thus avoid DaCS altogether. Similarly, the programming model supported by ALF was judged to be too restrictive to be compatible with the Pilot paradigm, so ALF was not used in CellPilot either.

Based on the possible locations of channel endpoints, Table I lists all types of channels that must be catered for.

<table>
<thead>
<tr>
<th>Type</th>
<th>Process Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PPE or non-Cell</td>
</tr>
<tr>
<td>2</td>
<td>PPE</td>
</tr>
<tr>
<td>3</td>
<td>PPE or non-Cell</td>
</tr>
<tr>
<td>4</td>
<td>SPE</td>
</tr>
<tr>
<td>5</td>
<td>SPE</td>
</tr>
</tbody>
</table>

Table I. CellPilot Channel Types

Type 1 transfers are between regular Pilot processes, and are handled by Pilot in the usual way, via MPI send/receive.

The other types require assistance from a PPE process. In order to avoid interfering with the PPE’s Pilot process (which may be occupied with its own computation or channel communication), a second MPI process, known as the Co-Pilot process, is created on each Cell node to provide services for these four channel types. Since Cell blades have two PPEs and each PPE has dual hardware threads, an added Co-Pilot process utilizes a computing resource that might otherwise go idle.

For type 2 transfers, the PPE process treats this as an MPI send/receive between itself and the Co-Pilot process. On the SPE side, mailbox messaging is used to send a read/write request to the Co-Pilot process. When Co-Pilot learns the address of the SPE process’s local memory buffer, it translates that into a main memory effective address (since SPE memory is mapped into main memory, from the PPE’s standpoint). Co-Pilot then uses that address in its own MPI call. The result is that the message transfers directly between the PPE’s buffer and the SPE’s local memory. Completion of the transfer is signalled to the SPE process by means of its mailbox. Note that this technique does not need recourse to DMA transfers.

Type 3 transfers are handled the same as type 2, with the PPE or non-Cell Pilot process contacting the remote SPE’s Co-Pilot process via MPI.

For type 5 transfers, both SPE processes send their buffer addresses to their respective Co-Pilot processes, which then make the transfer between themselves via MPI.

Type 4 transfers do not involve MPI. Both SPE processes send their buffer addresses to the Co-Pilot process and wait for transfer completion confirmation. Whichever address arrives first is stored, then the Co-Pilot process polls for requests until the second SPE’s request arrives. Co-Pilot calculates the effective addresses corresponding to the two SPE buffers, transfers the data using memcpy, and then notifies each SPE process of completion via their respective mailboxes.

Initially, we attempted to use pthreads to provide the Co-Pilot service. However, it seems that MPI is liable to be configured without support for threading. Rather than insisting on the system administrator’s reconfiguring and reinstalling MPI, we decided to make CellPilot use processes so that it will work with MPI_THREAD_SINGLE support.

C. Sample code

By way of example, the simple code in Figures 3 (PPE program) and 4 (SPE programs) runs on two Cell nodes. The PPE Pilot program is executed via mpirun. Each PPE process in turn starts one SPE process on its node (line 27 in main and line 10 in recvFunc), one of which writes an array of 100 integers to the other (line 43 writes; line 53 reads, with “*” illustrating the syntax for argument-supplied length). This is an example of using a Type 5 channel (Figure 2) which requires relaying through two PPEs. The channel is created on line 24 during the configuration phase, and the PI_CHANNEL variable is referenced in the SPE programs on lines 35 and 48.

A longer example (too bulky to print here), involves three channel transfers: from one SPE process to its parent PPE process, from there to another node’s PPE process, and from there to its SPE process. That example took 80 lines to code using CellPilot. Recoding this example using the Cell SDK required 186 lines, and called functions such as mfc_put, mfc_write_tag_mask, mfc_read_tag_status, spu_write_out_mbox, spe_in_mbox_status, and so on.
Recoding using DaCS required less code at 114 lines, and called dacs_remote_mem_create, dacs_remote_mem_query, dacs_put, dacs_wait, dacs_remote_mem_release, and so on. The simplicity of the CellPilot approach is shown by its small code size.

V. CELLPILOT PERFORMANCE

The main metrics of interest to CellPilot users are the amount of SPE’s 256K memory consumed by the library, and the latency of channel communication.

The CellPilot object file, cellpilot.o, takes up 10336 bytes of SPE storage (per the Linux size command). In comparison, the DaCS SPE library, libdacs.a, is 36600 bytes. With the CellPilot approach, a good deal of the logic is carried out on the PPE side by the Co-Pilot process where memory is abundant. Nonetheless, a pure SDK approach would be smaller, since CellPilot includes general purpose code for interpreting PI_Read and PI_Write format strings and carrying out mailbox messaging.

Timing results are given in Table II and graphed in Figure 5. For each bar, the lower solid portion represents the 1-byte message time, and the upper hashed portion the 1600-byte time. In addition, the throughput for the array case is graphed in Figure 6. Not surprisingly, Co-Pilot was found to add overhead compared to hand-coded transfers. Our
current analysis is that all SPE-connected channel types are paying some overhead for the Co-Pilot process. Moreover, type 2 uses MPI for the local PPE-to-Co-Pilot transfer, which could be a fast shared-memory copy, but nonetheless involves MPI processing in order to match the treatment of type 3 channels. Type 5 involves two Co-Pilot processes on separate Cell nodes. Given these insights, it is likely that Co-Pilot processing can be sped up in the future.

VI. STATUS AND FUTURE WORK

Although the CellPilot approach is new, and more testing is required, our initial results are encouraging. Currently, we are forging forward with various case studies for CellPilot, including the parallelization and implementation of scatter search, a well-known meta-heuristic that has been successfully applied to a variety of NP-hard problems, pri-
CellPilot will be made available for student projects in upcoming University of Guelph parallel programming classes. It will be available for free public downloading from the Pilot website [7].

As for future work, the main features of Pilot that currently do not apply to CellPilot are collective operations and deadlock checking. Pilot provides a subset of collective functions that are available to PPE and non-Cell processes, but CellPilot does not yet support collective operations among SPEs, much less involving a mixture of SPE and other processes. It may also be possible to optimize the operation of the Co-Pilot process and reduce its overhead to be more competitive with low-level, hand-coded methods.

VII. CONCLUSION

The two key innovations furnished by CellPilot are (1) the Co-Pilot PPE process, which effectively allows SPE processes to participate in MPI as first-class citizens; coupled with (2) the extension of Pilot’s “friendly face for MPI” to both PPE and SPE programs. The first innovation enables the delegation of all message handling, buffering, and synchronization to MPI which already has the proven code to do those things. The second means that programmers are spared from dealing with the complexities of both MPI and the Cell SDK. This was accomplished by adding only two function calls to the Pilot API (PI_CreateSPE and PI_RunSPE).

CellPilot also has at least two strengths in its implementation: (1) The bulk of SPE messaging logic has been off-loaded onto the Co-Pilot PPE process, thereby conserving scarce SPE memory; and (2) the fact that SPE memory can be mapped into PPE effective addresses is used to set up direct transfers for MPI and intra-Cell messages, thus avoiding extra buffering and copying.

In summary, CellPilot provides the following services to a Cell cluster programmer: (1) easy starting (with parameters) and stopping of SPE programs; (2) all kinds of communication between PPE, SPE, and non-Cell processes using a single, uniform API, while hiding the complications of DMA transfers, signals, mailboxes, alignment issues, and network transfers. The programmer still has to cope with the limited memory available on the SPEs.

Using CellPilot, a scientific programmer is able to design and implement a cluster application based on the process/channel model, and utilize the processor resources of a Cell cluster without delving into low-level communication details. By having a single, unified programming model, one major difficulty—that of interprocess communication and synchronization—can be overcome.

ACKNOWLEDGMENT

This research was supported by NSERC (Natural Science and Engineering Research Council) of Canada. We gratefully acknowledge the use of the Cell cluster hosted by SHARCNET (Shared Hierarchical Academic Research Computing Network), and consultations with R. Enenkel (IBM Canada) and M. Perrone (IBM USA).

REFERENCES


