Execution Environment Support for Many Core Heterogeneous Accelerator Platforms

Vishakha Gupta
Georgia Institute of Technology
vishakha@cc.gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
sudha@ece.gatech.edu

Jose Duato
Universidad Politecnica de Valencia, Valencia, Spain
jduato@disca.upv.es

Abstract

We are seeing the advent of large scale, heterogeneous systems comprised of homogeneous general purpose cores intermingled with customized heterogeneous cores and interconnected to diverse memory hierarchies. The presence of accelerators requires support for new programming abstractions and run-time environments that can efficiently harvest platform resources comprised of general purpose and specialized processing cores, their diverse memory units and memory management support, and communication links that connect them. This paper describes an execution model and systems infrastructure for modeling and supporting multi-accelerator architectures in general and experiences with an implementation for interconnected network of Cell Broadband engine processors in particular. The primary contributions of this paper are i) a pooled accelerator execution model for orchestrating computations on and data movements across multiple accelerators, ii) an API for implementing the model effectively and iii) a distributed simulation environment for modeling multiple, communicating Cell/B.E. processors.

1. Introduction

The relentless progress of Moore's Law has periodically inspired major hardware and software innovations at specific points in time to keep performance growth at pace with transistor density. The industry has reached another such point where one can see some inevitable trends including performance scaling via replication of cores and the use of custom accelerators. The Cell/B.E. [8] is an example of such a technology inspired architecture. Along with a transition from highly pipelined, complex processors to simplified, more power-efficient execution platforms, general purpose cores are being enriched via additional asynchronous processing units to accelerate the execution of certain workloads, including GPUs, crypto units, and network processors. Currently, these units are externally connected to the multicore chip via fast interconnects and there are ongoing efforts to achieve tighter couplings (e.g., System on a Chip, SMP Coherency Busses, and AMD's Torrenza Intitiative). Therefore we can expect to see large scale, heterogeneous systems comprised of homogeneous general purpose cores intermingled with customized heterogeneous cores and connected to diverse memory hierarchies.

Regardless of the degree of coupling, the presence of such accelerators requires support for new programming abstractions and run-time environments that can efficiently harvest platform resources comprised of general purpose and specialized processing cores, their diverse memory units and memory management support, and communication links that connect them. Substantial efforts have gone into making it easier for applications to use accelerators, both by providing low-level runtime support and interfaces [17, 13] used by custom codes, and by offering higher level libraries [20, 12] for general use. Multiple working groups have developed interfaces and powerful libraries to reduce the complexity of programming accelerators [9]. But efforts to date have largely ignored the issues related to the support for multi-accelerator systems. *This paper describes an execution model and infrastructure for modeling and supporting multi-accelerator architectures - specifically architectures comprised of a large number of Cell Broadband engine processors.*

The primary contributions of this paper are i) a pooled accelerator execution model for orchestrating computations on and data movements across multiple accelerators, ii) an API for implementing effective movement of data and control information across multiple elements and iii) a distributed simulation environment for modeling multiple, communicating Cell/B.E. processors. In particular, this simulation infrastructure, the Multi-Cell Simulator (MCS) is an easily configurable simulation tool useful for exploring a wide range of issues of interest in multi-cell architectures including, development of parallel multi-cell applications, communication optimizations, interconnection network topologies, and allocation/management algorithms. The size of the multi-cell system that can be simulated and supported by the execution model is only limited by number of physical cores available to host parallel instances of the Cell simulator.

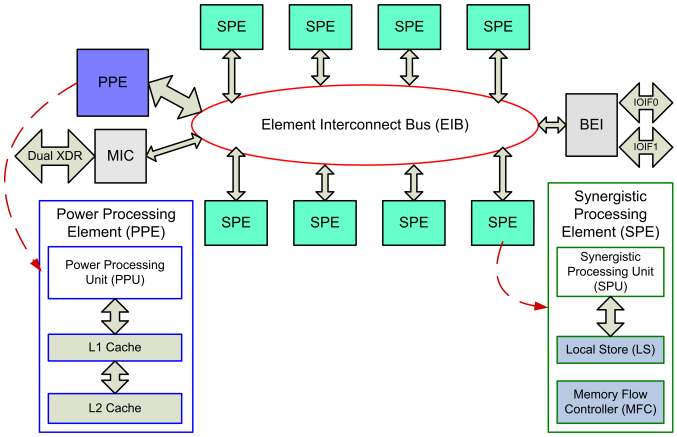

Figure 1. Cell Processor Architecture

Section 2 covers the Cell/B.E Processor. Section 3 talks about the execution model in general followed by a description of the execution model on the Cell/B.E. system in Section 4. The remainder of this paper describes the prototype that implements the execution model on a multi-cell simulator followed by experience with some applications in Sections 5 and 6. The paper concludes with a discussion of related work in Section 7, enhancements to the MCS functionality and directions for future work in Section 8.

2. The Cell Broadband Engine Processor

Cell is a design solution based on the analysis of workloads in areas such as cryptography, graphics transform and lighting, physics, fast-Fourier transforms (FFT), matrix operations, and scientific workloads [8, 6, 5]. The cell processor is capable of delivering 204.8 GFlop/sec single precision and 14.6GFlops/sec double precision floating point performance [14]. It has an aggregate memory bandwidth of 25.6GB/s at 3.2GHz. Figure 1 shows the architecture of the Cell/B.E. processor. It is a heterogeneous chip multiprocessor that consists of a 64-bit dual-threaded Power core, referred to as the Power Processing Element (PPE), augmented with eight specialized high performance co-processors based on a novel single-instruction multiple-data (SIMD) architecture, referred to as Synergistic Processor Elements (SPEs). The SPEs and PPE are connected to each other and to the 512KB L2 cache via a coherent on-chip element interface bus (EIB) that consists of four sixteen-byte data rings with 64-bit tags.

PPE performs all the control tasks and the SPEs are responsible for taking care of the compute intensive tasks giving the Cell processor tremendous computing capabilities. These are simple RISC based, extremely power efficient cores with a 128 register file of 128bit registers and local store to which they can DMA data to and from the main memory. The PPE accesses main storage (the effective-address space) with load and store instructions, the contents of which may be cached. The SPEs, in contrast, access main storage with Direct Memory Access (DMA) commands that move data and instructions between main storage and a private local memory called local store(LS). Load-store data and other instruction-fetches for an SPE access its private LS rather than shared main storage, and the LS has no associated cache. PPE and SPEs can also communicate over mailboxes and event channels. The onboard memory interface controller (MIC) supports the Rambus XDR memory standard. The Broadband Engine Interface (BEI) Unit manages data transfers between the processor elements on the Element Interconnect Bus (EIB) and I/O devices.

Programming Cell - SPE Management Library The SPE Runtime Management Library (libspe) [13] is the standardized low-level application programming interface (API) that enables application access to the Cell/B.E. SPEs. This library provides an API that is neutral with respect to the underlying operating system and its methods to manage SPEs. In order to enable the use of multiple SPEs in parallel for an application, it supports functions like a) Creating N SPE contexts (logical representation of SPEs as seen by software), b) Loading the appropriate SPE executable, c) Running N threads for N SPE contexts, d) Waiting for threads to finish and then e) Destroying all N contexts, with the help of some standard thread package. Our API leverages these functionalities to provide a richer interface to one or multiple Cell/B.E.s.

3. An Execution Model for Heterogeneous Architectures

This section describes a general execution model for heterogeneous architectures, while a specific instantiation for the Cell/B.E. is described later in this paper. An execution model has two principal components: a resource view and a (often implied) programming model. The execution model binds program objects to hardware objects and specifies synchronization and communication requirements and in general how programs execute. We employ a stream programming model as illustrated in Figure 2. Variations of this model have been proposed elsewhere, and in general we adopt the principal features of models from [1, 4]. Applications are multithreaded implementations, where a thread orchestrates the execution of streaming kernels. Each kernel processes input data streams and produces output data streams, thereby encapsulating the computationally intensive components of the application. Threads execute on standard cores, while kernels are mapped to accelerators. Some restrictions may be placed on the form and structure of data types permitted in a single stream as well as the structure of computations permitted in a kernel. For example, pointers are generally not supported in case of FPGA accelerator. Kernels form independent compilation units that are linked with host core executables.

With respect to the resource component of the execution model, the system architecture is viewed as a a pool of ho-

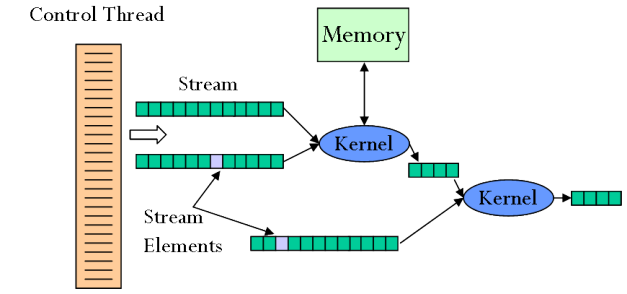

Figure 2. Stream programming model

mogeneous cores and heterogeneous accelerators. In general, compute accelerators such as GPUs, FPGAs, or Cell/B.E. SPEs are connected to general purpose host cores via high speed links such as AMD's Hypertransport, PCIe, or in the Cell/B.E. the EIB. Further the heterogeneous cores may have access to an inter-processor switched network through a host core or may have a direct interface to the network while they share access to a common memory space with the host. An *accelerator platform* is configured with one or more cores coupled to one or more accelerators. A platform may involve cores that span multiple chip and board boundaries with non-uniform inter-core communication latencies, for example, an accelerator platform may be configured with one PPE and 12 SPEs. The library API described in Section 4 provides the run-time communication infrastructure to implement this execution model.

Presuming a large system with 10s to 1000s of accelerators, we believe the accelerator platform abstraction simplifies application development, is compatible with many compilation chains, and forms an efficient execution environment for many applications. In this paper we present the design and implementation of the library for creating accelerator platforms for systems comprised of multiple Cell/B.E. processors.

4. Execution Model: Implementation on the Cell/B.E.

Consider a system comprised of multiple, interconnected Cell/B.E. processors which presents a pool of cores (PPEs and SPEs). A developer can specify an accelerator platform that consists of one or more PPEs and one or more SPEs forming a subset of the existing set of all PPEs and SPEs in the cluster. No explicit consideration is given to chip boundaries although the shared address space of the platform is no longer coherent. For example, one can configure an accelerator platform consisting of one PPE and 12 SPEs as shown in Figure 3. The platform API supports deployment of compute kernels on the SPEs, using an intermediate PPE transparently if necessary. We argue that the main value of this model is the ability to program large systems like those comprised of 4 PPEs and 128 SPEs and to be able to view

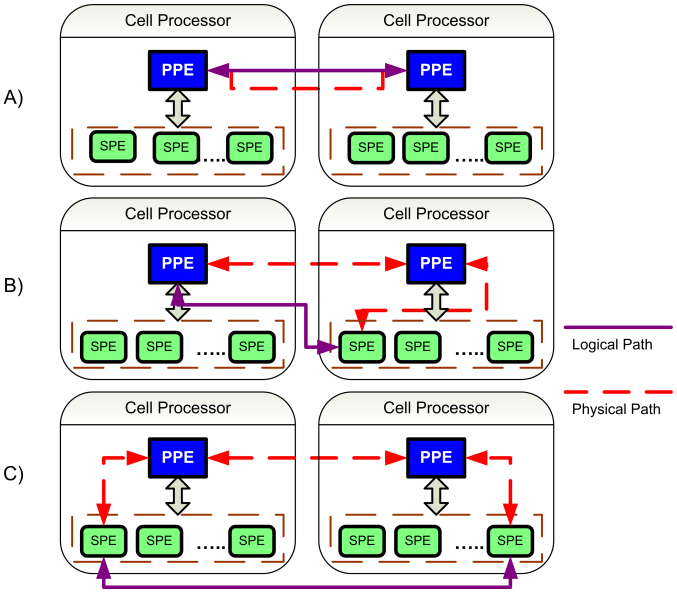

Figure 4. Logical communication paths exported by the API vs physical paths

this combination as a single accelerator platform rather than having to explicitly manage a cluster of distinct Cell/B.E.s.

In addition to the platform abstraction, another advantage offered by the library is a global name space for all the elements in the multi Cell system, thus making it easier for the programmer to address local as well as remote elements uniformly. The programmer is not required to configure a platform for element to element communication although he can choose to do so if the application calls for it. The basic inter-core communication mechanism supported in the Cell/B.E. is abstracted and transparently optimized by the API implementation. For example, data can be transferred a) between a PPE and an SPE on the same Cell via DMA, mailbox, channels or memory mapped I/O, b) between two PPEs on different Cell processors via communication calls provided by the platform API, c) between PPE and some remote SPE via communication through the remote PPE (transparently) and d) between two mutually remote SPEs. Figure 4 depicts the different logical and actual communication paths discussed above.

The API is structured around the following elements:

1. *Connection* - Connection-based communication is employed where connection establishment also serves to verify the presence and correctness of the communication endpoints.
2. *Stream and Packet* - The API supports both packet-based and stream-based communication.
3. *Element-Element and Group* - Both point-to-point and group communication semantics are supported. A group is a named set of PPEs and/or SPEs and the collective communication operations are patterned after MPI.

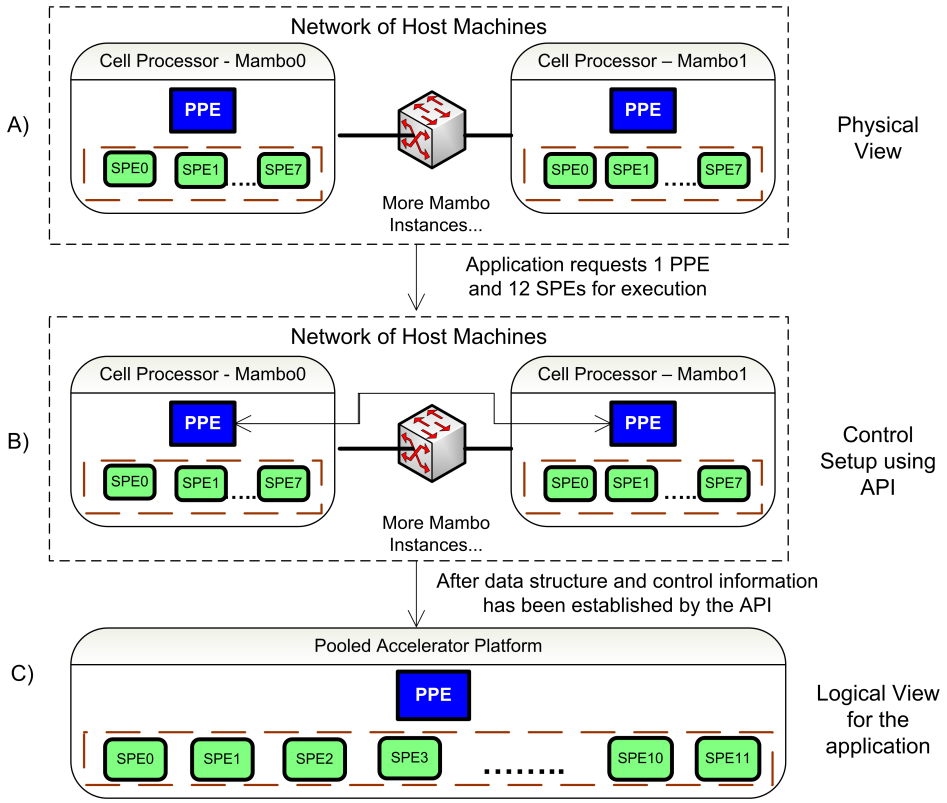

Figure 3. Deriving a logical accelerator platform from the physical infrastructure

4.1 API Components

The function classes forming the library API can be classified into the following general categories:

- Platform Management - These functions are responsible for creating platforms and servicing intra-platform communication requirements. Figure 3 shows the sequence of events for configuring a platform of accelerators within a network of Cell/B.E. processors. The implementation seeks to minimize fragmentation - i.e., use the minimum number of Cell/B.E. processors (the current version just looks at the round trip times and does not consider interconnect topology). Once the required platform has been created, the programmer can work with the SPEs independent of the physical connectivity and perform operations like loading executables, sending mailbox messages, querying SPE state etc. As of the current implementation, the programmer still needs to be aware of correct data partitioning so that memory gets allocated on nodes as required. To make this simpler, there is a call to query the rank of local elements within a platform.
- Group Management - Groups for collective communication may comprise of only PPEs, SPEs or contain both types of cores. The implementation of groups allocates members based on network proximity. Alternatively the developer may specify members of a group. If the group consists only of SPEs, there are still PPEs involved in the implementation of group communication in order to perform the actual transfer between SPEs located on distinct Cell/B.E. chips.
- Data Communication - In addition to source/destination specification, communication functions also enable specification of the data transfer patterns. As mentioned earlier in this section, data transfers can be unicast or multicast (group) and may have additional associated semantics. For example, transfers may be blocking/non-blocking or in-order/out-of-order. The API implementation itself is multi-threaded and the degree of concurrency can be specified as a configuration parameter.
- Memory Management - When transferring large buffers locally or remotely there are requirements such as memory alignment and page sizes. The memory management module serves to hide these complexities from the programmer who can just specify the size of buffer, alignment and optionally provide sharing information.
- Timing and Synchronization - Although not complete yet, this module is designed to provide timing information to all instances and synchronization services such as barriers taking into account the SPE primitives. In particular, this will be valuable for cycle accurate parallel simulation of multicell systems - the current parallel simulator

is only functionally accurate due to the use of functional models of the inter-Cell communication.

4.2 An Example

Platform creation and removal is similar to group management, however they operate in a distributed manner with minimal communication of information across nodes. For example, when a programmer calls *mcs_platform_create*, he can specify the number of PPEs and SPEs and even provide a list of their identifiers. The call verifies whether the system comprises of the required number of elements. This information is requested from all nodes by the master through exchange of messages. The specified number of elements sent to the Master node by slaves are committed on all Cell/B.E.s until the Master node informs them of the actual number of elements they are expected to contribute to the platform being formed. Participating nodes then save this information. Once this function returns successfully, the *mcs_platform_spe_context_create* call allocates spe contexts only for local spes on each participating node per platform on a distributed basis. The programmer can now invoke *mcs_platform_spe_program_load* to load the SPE image on all the SPEs without looping. Other calls for platform management including the mailbox send and receive calls work in a similar fashion, thus reducing the overall communication overhead in the system and reducing the number of calls a programmer typically has to make to create traditional Cell/B.E. applications.

Data Communication The communication primitives are inspired by and are closely patterned after the MPI library [16] for message passing systems. The library exports an accelerator independent interface providing a consistent view of a pool of accelerators for parallel application programs covering basic functionality such as initialization, addressing, group communication, and transparency across chip and board boundaries. The list of function calls in the Appendix shows some of the functions forming the API.

5. Prototype Environment

We have built a parallel cell simulator to advocate the approach described in the paper. In this multi Cell Simulator (MCS), the host side simulator configuration and boot module have been implemented with components as shown in Figure 5. The Host Side setup components are implemented in order to boot up the multiple Mambo instances using configuration parameters as specified by the user or provided as default and execute the selected parallel application. The API components are implementations of those described in Section 4. The library has been structured to distinguish between (local) fast path and remote communication. The implementation of the sample API function calls enlisted in the Appendix checks for correctness of the arguments, deciphers whether the call pertains to local components or remote components and then calls the correct local or interconnect functions. Error codes indicating reason(s) for failure are returned if any of the functions fail thus making it easier to debug massively parallel applications.

We have implemented the API and tested it on a cluster of multiple Cell/B.E. simulators [10] (or Mambo as it is commonly referred as) which can serve the role of actual Cell processors. The MCS software stack is shown in Figure 5. Applications utilize the MCS library and execute within a Linux environment booted on IBM's Mambo execution environment. The communication between Mambo and the host machine is carried through the virtual TUN device [15]. This software stack is replicated on all the machines participating in the cluster that host the simulator instances. The application needs to link against the library to be able to use it. We rely on the TAP forwarding capabilities of the Linux kernel to enable communication between elements on different simulators on the same host or on different hosts.

The library is written in C and can be compiled using gcc. The first version provides most of the calls except the group based communication and synchronization functionality [7]. We use the pthreads API for implementing threads in the library. As of the present implementation, the network communication uses the socket API and data is transferred over Ethernet.

6. Evaluation

Testbed and Benchmarks: We have tested our benchmarks on varying number of simulator instances ranging between two and eight. Each of the simulator instance currently simulates 1 PPE and 8 SPEs. The operating system used on the host side as well as on the simulator is Fedora Core 6 because the version of IBM SDK we have used had been tested on Fedora Core 6 and it provides the same system image to boot on the simulator as well. There is a host side script which asks the programmer for configuration information such as the number of simulators required, the type of communication protocol (TCPIP/Infiniband etc) to use and other such parameters and then boots all the simulator instances. These simulators can be booted on one host or multiple hosts The process of creating routing tables relevant to simulator network has not been automated yet and it is left to the programmer and the operating system (because of TUN/TAP forwarding) to make sure that one virtual tap interface can communicate with another. We have used IBM SDK2.1 [11] for providing the Mambo simulator and the SPE management calls.

The following benchmarks have been used to test the multi Cell simulator:

Chained Matrix Multiplication : Matrix chain multiplication [19] is an optimization problem that can be solved using dynamic programming. We have many options because matrix multiplication is associative. However, the order in which we parenthesize the product affects the number of

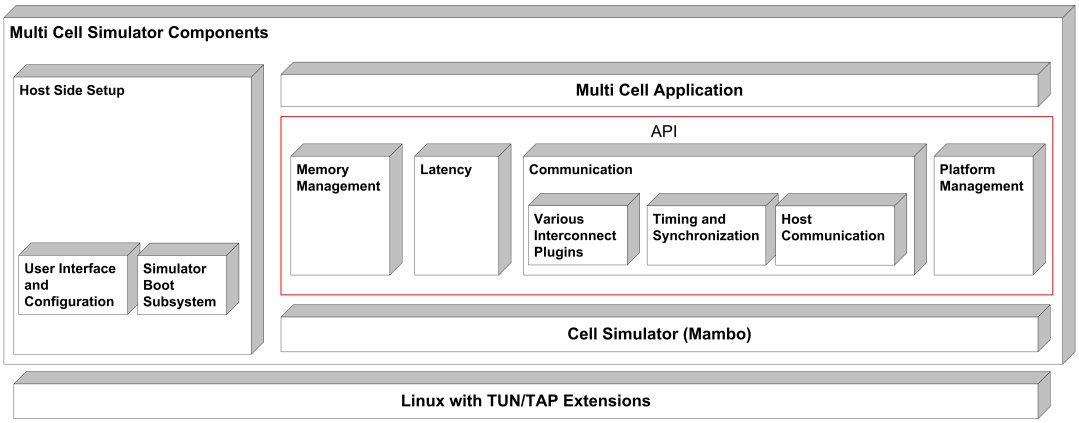

Figure 5. Software components for Multi Cell Simulation

simple arithmetic operations needed to compute the product, or the efficiency. Consider the example where we need to multiply four matrices A, B, C and D on two nodes. The optimum ordering in this case could be ((AB)C)D. But this would imply less parallelism as compared to (AB)(CD). We employ this simple technique and present this problem as a proof of concept for our communication API.

We deploy two to eight simulators to multiply 4 to 16 matrices of size 256x256 using single precision floating point arithmetic. The matrix multiplication benchmark present in the Cell SDK has been modified to include the required changes. All Cell processors (represented by the Mambo instances) generate the input data for pairs of matrices being multiplied in this example. The processors with odd rank send their output to the even rank which in turn send the intermediate output to the higher in rank processor. Table 1 shows the overhead imposed by the library against the amount of data being transmitted by the application for different data sets.

# Matrices	# Cells	Overhead (Bytes)	Matrix data (KB)
4	2	28	256
8	4	84	768
16	8	196	1792

Table 1. Communication overhead introduced by the API

Black Scholes : The Black-Scholes option pricing formula prices European call or put options on a stock that does not pay a dividend or make other distributions. The formula assumes that the underlying stock price follows a geometric Brownian motion with constant volatility. It is historically significant as the original option pricing formula published by Black and Scholes in their landmark paper [2].

Our implementation creates a platform using the platform API, splits the entire data among the required number of SPEs and triggers the SPEs to calculate the option prices using the Black Scholes equation. We use this benchmark as a proof of concept for the platform API that transparently moves data across chip boundaries. Platform creation results in exchange of information between the master and slaves. While creating the platform, the slaves first inform the master of the number of SPEs (PPEs can be multiplexed but SPEs have to be available for use) and the master then has to respond with platform related information such as the rank of a particular slave in the platform, the number of SPEs contributed from that Cell. This results in 88Byte network messages per slave independent of the application and is proportional to the number of slaves present in the system.

Discussion of Results The library initialize function results in exchange of messages between slaves and master (in this case, Mambo instance with rank 0) informing the master about certain parameters like number of usable spes, total ppes etc. The master also measures round trip time (RTT) to slaves while exchanging these messages in order to keep track of the topologically ”closer” slaves. This flow of information results in 112Bytes of network messages from each slave to the master, thus increasing linearly with the number of slaves.

As seen from the two benchmarks presented, the overhead introduced by the library is negligible. We intend to measure runtime information and execution time overhead introduced by the library once we incorporate the cycle accurate network simulator with MCS. But from a functional point of view, the advantage for the programmer is the increased data set size made possible due to increase in the number of Cell processors that can now be tested using this assembly of simulators. One instance of Mambo can at most boot 2 PPEs and 16 SPEs if configured in the blade mode but with MCS, we can increase the number to as many as desired. The only limitation imposed is by the physical hosts and their network connectivity. Once we have a setup with multiple Cell blades, we intend to test the API and the parallel benchmarks so that we get an estimate of their timing behavior. The low overhead introduced by the library is more

than compensated for by the simplicity of a global address space for all accelerators and the platform abstraction.

7. Related Work

IBM has released a generic acceleration programming framework ALF [9] and intends to make it a standard for accelerator programming. While designing a massively parallel application that can run threads on the heterogeneous cores available on the same chip, as well as distribute data across multiple such chips, the issues to be handled are two fold - a) Level 1 distribution of data and control across processors and b) Level 2 distribution of data and tasks so that work can be divided among the compute efficient on-chip cores. The API presented in this paper is primarily intended for the first part and by providing the platform abstraction, makes it easy for the programmer to focus on the actual data division logic among the compute-efficient cores required by the second part. ALF on the other hand is designed to help the programmer think about how an application can be split into parallel tasks and what data could form input to these tasks. So ALF could actually replace the SPE library currently being used by our API in order to provide a richer set of functionality for the programmer.

The other class of related work is in the area of cluster management systems like Conga [18]. Conga provides management functions useful more from an administrator's perspective than a programmer's perspective whereas MCS has been built as a tool to experiment with heterogeous multicore and multi processor environments. The API we propose differs from the multi-processor programming environments such as MPI [16] and Open MPI [3] with respect to the overall goal we are trying to achieve. As we have mentioned earlier, the API is targeted to reduce complexity while maintaining functionality of heterogeneous multi-processor systems.

8. Conclusion and Future Work

This paper has documented preliminary work on a accelerator platform API that is proposed to enable creation and management of compute clusters comprised of homogeneous cores augmented with heterogeneous accelerators. The first implementation of the API has been on the Cell/B.E. facilitated by a multi-cell simulator(MCS) that can model execution for systems of interconnected Cell/B.Es.

Several avenues for future work are immediately apparent. We plan to port the API to multi-cell blade systems to assist application development. This will involve porting the communication layer to alternative interconnects, in this case to Infiniband, in addition to ethernet. While the group communication semantics have been defined, the implementation and testing of all of the calls has not been completed. We also have an ongoing effort for the development of a cycle accurate parallel simulator for interconnection networks. We plan to integrate that with the MCS to enable experimentation with interconnection architectures for multi-cell systems. We anticipate augmenting the Cell/B.E. experience with a port of the API to a compute cluster comprised of GPUs or FPGAs. Finally, we observe that the MCS may find great value in experimenting with multicell applications.

References

[1] DAS, A., DALLY, W. J., AND MATTSON, P. Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2006), ACM, pp. 33–42.

[2] FISCHER, B., AND SCHOLES, M. The pricing of options and corporate liabilities. In *Journal of Political Economy* (1973), pp. 81:637–659.

[3] GABRIEL, E., FAGG, G. E., ET AL. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[4] GOKHALE, M. B., STONE, J. M., ARNOLD, J., AND KALINOWSKI, M. Stream-oriented fpga computing in the streams-c high level language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines* (Washington, DC, USA, 2000), IEEE Computer Society, p. 49.

[5] GSCHWIND, M. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers* (New York, NY, USA, 2006), ACM Press, pp. 1–8.

[6] GSCHWIND, M., HOFSTEE, P., ET AL. Synergistic processing in cell's multicore architecture. *IEEE Micro 26*, 2 (March 2006).

[7] GUPTA, V., AND YALAMANCHILI, S. MCS API documentation. `http://www.cc.gatech.edu/~vishakha/projects.php#MCS`.

[8] HOFSTEE, H. P. Power efficient processor architecture and the cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 258–262.

[9] IBM CORPORATION. Accelerated library framework programming guide. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD0`.

[10] IBM CORPORATION. Full-system simulator user's guide. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/B494BF3165274F67002573530070049B`.

[11] IBM CORPORATION. IBM cell broadband engine software development kit. `http://www.alphaworks.ibm.com/tech/cellsw`.

[12] IBM CORPORATION. SIMD math library specification for cell broadband engine architecture. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/6BFB9899CEA5456800257360001938B3`.

[13] IBM CORPORATION. SPE management library. Part of Cell Broadband Engine SDK Documentation.

[14] KISTLER, M., PERRONE, M., AND PETRINI, F. Cell multiprocessor communication network: Built for speed. In *IEEE Micro* (May/June 2006), IEEE Computer Society.

[15] KRASNYANSKY, M. Universal tun/tap device driver. `http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt`.

[16] MPI FORUM. MPI: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[17] NVIDIA CORPORATION. NVIDIA CUDA revolutionary gpu computing. `http://developer.nvidia.com/object/cuda.html#documentation`.

[18] REDHAT ENTERPRISE LINUX. Conga - a management platform for cluster and storage systems. `http://sourceware.org/cluster/conga/spec/`.

[19] WIKIPEDIA. Matrix chain multiplication. `http://en.wikipedia.org/wiki/Chain_matrix_multiplication`.

[20] WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

A. Example Functions from MCS API

Following is a list of important functions forming the MCS API. Detailed description, an exhaustive list of these functions and state of their implementation can be found at [7]

```
Library Initialize and Cleanup:
INT mcs_init(BOOL need_platform, BOOL
    need_slaves);
INT mcs_destroy();

Platform:
INT mcs_platform_init();
PLATFORM_ID mcs_platform_create(ELEMENT_ID
    *ppes, ELEMENT_ID *spes, INT num_ppes, INT
    num_spes);
INT mcs_platform_cpu_info_get(PLATFORM_ID plid,
    UINT info_requested, INT cpu_node);
INT mcs_platform_get_rank(PLATFORM_ID plid, UINT
    info_requested);
INT mcs_platform_spe_context_create(PLATFORM_ID
    plid, ppu_pthread_data_t *datas, UINT flags,
    spe_gang_context_ptr_t gang);
INT mcs_platform_program_load(PLATFORM_ID plid,
    spe_program_handle_t *spu_exec);
INT mcs_platform_thread_create(PLATFORM_ID plid,
    void * (*start_fn)(void *), void
    *thread_pkg);
INT mcs_platform_in_mbox_send(PLATFORM_ID plid,
    UINT **mbox_data, INT count, UINT behavior);
INT mcs_platform_out_mbox_recv(PLATFORM_ID plid,
    UINT **mbox_data, INT count);
INT mcs_platform_spe_wait(PLATFORM_ID plid, void
    **value_ptr);
INT mcs_platform_context_destroy(PLATFORM_ID
    plid);
INT mcs_platform_destroy(PLATFORM_ID plid);

Group Management (in progress):
GROUP_ID mcs_create_group(ELEMENT_ID *ppes, UINT
    num_ppes, ELEMENT_ID *spes, UINT num_spes);
INT mcs_modify_group(GROUP_ID grp_id, INT
    oper_type, UINT num_ppes, UINT num_spes);
INT mcs_destroy_group(GROUP_ID grp_id);
INT mcs_get_group_size(GROUP_ID grp_id, INT
    *num_spe, INT *num_ppe);
INT mcs_get_group_info(GROUP_ID grp_id,
    group_info_t grp_info);

Data Communication:
CONNECTION_ID mcs_connect_sender(ELEMENT_ID
    receiver, INT recv_port, ELEMENT_ID sender,
    INT send_port, INT conn_type, INT timeout);
CONNECTION_ID mcs_connect_receiver(ELEMENT_ID
    receiver, INT recv_port, INT conn_type);
INT mcs_send(CONNECTION_ID conn_id, void *buff,
    UINT size, INT data_type, INT timeout, UINT
    flags);
INT mcs_stream_send(CONNECTION_ID conn_id, void
    *info, INT data_type, INT timeout, UINT
    flags);
INT mcs_recv(CONNECTION_ID conn_id, void *buff,
    UINT size, INT timeout, UINT flags);
INT mcs_stream_recv(CONNECTION_ID conn_id, void
    *info, INT timeout, UINT flags);
INT mcs_track_conn(CONNECTION_ID conn_id,
    conn_info_t *conn_info);
conn_info_t *mcs_query_data(CONNECTION_ID c_id);
INT mcs_track_group_conn(GROUP_ID group_id,
    group_info_t *group_info);
INT mcs_wait(CONNECTION_ID conn_id);
INT mcs_close(CONNECTION_ID conn_id);
INT mcs_connect_sender_group(GROUP_ID receivers,
    INT recv_port, ELEMENT_ID sender, INT
    base_send_port, INT conn_type, INT timeout);
INT mcs_group_send(GROUP_ID grp_id, void *buff,
    UINT size, INT data_type, INT timeout, UINT
    flags);
INT mcs_group_stream_send(GROUP_ID grp_id, void
    *buff, UINT size, UINT split_size, INT
    data_type, INT timeout, UINT flags);
INT mcs_group_recv(GROUP_ID grp_id, void **buff,
    UINT size, INT timeout, UINT flags);
INT mcs_group_stream_recv(GROUP_ID grp_id, void
    **buff, UINT size, INT timeout, UINT flags);
INT mcs_group_wait(GROUP_ID group_id);
INT mcs_group_close(GROUP_ID group_id);

Memory Management:
void *mcs_alloc(size_t size, void *share_info);
void *mcs_alloc_align(size_t size, INT
    alignment, void *share_info);
void *mcs_platform_alloc_align(PLATFORM_ID plid,
    size_t chunk_size, INT alignment);
INT mcs_dealloc(void *area);
INT mcs_dealloc_align(void *area);

Timing and Synchronization (in progress):
TIME_T mcs_measure_rtt(UINT src_sim_id, UINT
    dst_sim_id);
TIME_T mcs_get_host_time();
TIME_T mcs_get_app_time();
INT mcs_barrier(GROUP_ID grp_id);
INT mcs_get_group_time(GROUP_ID grp_id, TIME_T
    *mean, TIME_T *std_dev);
INT mcs_sync_group_time(GROUP_ID grp_id, TIME_T
    *mean, TIME_T *std_dev);
```