Novel Performance Optimization of Large-Scale Discrete-Event Simulation on the Cell Broadband Engine

Qi Liu, Gabriel Wainer
Department of Systems & Computer Engineering
Carleton University, Ottawa, ON, Canada
{liuqi, gwainer}@sce.carleton.ca

Ligang Lu, Michael Perrone
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
{lul, mpp}@us.ibm.com

ABSTRACT

This paper presents a computing technique for efficient parallel simulation of large-scale discrete-event models on the IBM Cell Broadband Engine (CBE), which has one Power Processor Element (PPE) and eight Synergistic Processing Elements (SPE). Based on the general-purpose Discrete Event System Specification (DEVS), the technique tackles all performance bottlenecks, combining multi-dimensional parallelism and various optimizations. Preliminary experiments have produced very promising results, attaining speedups up to 134.34 and 41.23 over the baseline implementation on PPE and on Intel Core2 Duo E6400 processor respectively. The methods can also be applied to other multicore and shared-memory architectures. We conclude that the technique not only allows discrete-event simulation users to tap CBE potential without being distracted by multicore programming, but also provides insight on migration of legacy software to current and future multicore platforms.

KEYWORDS: Discrete-event simulation, multicore computing, DEVS formalism, Cell Broadband Engine

1. INTRODUCTION

Discrete-event simulation has been used to study complex systems in a broad array of domains. The Discrete Event System Specification (DEVS) [1], in particular, supports hierarchical construction of reusable models in a modular way. Numerous extensions to DEVS have been proposed in the literature. P-DEVS [2] extends DEVS to handle simultaneous events in the simulation. Cell-DEVS [3] defines n-dimensional cell spaces as discrete-event models where each cell is a basic DEVS model component. Both P-DEVS and Cell-DEVS have been implemented in CD++ [4], which is an object-oriented Modeling and Simulation (M&S) environment programmed in C++.

Parallel Discrete-Event Simulation (PDES) is widely accepted as a viable approach to efficient discrete-event simulation [5]. Traditionally, parallelism is achieved by partitioning a simulation onto multiple nodes of a cluster to exploit concurrent activities at different model components. While these coarse-grained parallelization strategies have achieved success in improving simulation performance, most of them neglect to integrate with other fine-grained parallelism available on multicore platforms.

As the monolithic approach to microprocessor design reaches a point of diminishing return, the industry is moving towards multicore Chip Multiprocessor (CMP) architectures. A latest example is the heterogeneous IBM Cell Broadband Engine (CBE) processor [6], which has a main Power Processor Element (PPE) and 8 specialized co-processors called Synergistic Processing Elements (SPEs). Each SPE can only directly access a fast, small, non-coherent local storage (LS) to execute the bulk of the workload in small chunks. Data sharing is achieved mainly through software-managed explicitly-addressed autonomous Direct Memory Access (DMA) transfers to and from the system main memory. In addition, the SPEs support 128-bit SIMD (Single Instruction, Multiple Data) operations that can be applied at different granularities. While the CBE processor offers a vast number of parallelization options at different levels, the asymmetric architecture of heterogeneous cores with explicit memory control requires innovative redesign of existing algorithms to achieve optimal execution efficiency.

As multicore computing becomes pervasive, there is a growing need for novel PDES algorithms targeting CMP platforms. To this end, we propose a computing technique that combines multi-dimensional parallelism in large-scale discrete-event simulations, while hiding the technical details of multicore programming. Based on the general-purpose DEVS and Cell-DEVS formalisms, the technique tackles all main performance bottlenecks in the simulation, incorporating various optimizations in a systematic way.

A realistic wildfire propagation model is used to illustrate the impact of each optimization step. Although not all of the proposed algorithms are fully implemented, the ones currently available have already produced very promising results, attaining simulation speedups up to 134.34 and 41.23 over the baseline implementation on the PPE and on the Intel Core2 Duo E6400 processor respectively.

In the following, Section 2 and 3 review the challenges and DEVS simulation in CD++. The fire simulation profile is analyzed in Section 4. Section 5 and 6 cover the optimization and parallelization strategies. The computing technique is proposed in Section 7. Section 8 shows the experimental results, and Section 9 concludes the paper.

2. CHALLENGES

Existing PDES techniques usually adopt a coarse-grained parallelization strategy at the cluster level without paying much attention to other fine-grained parallelism available on multicore processors. Multi-grained parallelism has been recently exploited on CMP architectures in the context of scientific and multimedia applications [7]. New parallelization strategies for PDES are required to explore multi-dimensional parallelism at different levels on CBE.

Different programming models were proposed to improve programmability on CBE [6]. Although these abstract models provide excellent guidelines for designing new computing techniques, significant efforts are still needed to address PDES peculiarities.

Compiler-assisted vectorization is one way to facilitate software development on CBE [8]. Without understanding the application logic, nonetheless, this technique is still inadequate on its own for complex PDES systems involving highly irregular computation that must respect the causal dependency among individual events.

Several middleware frameworks have also been developed on top of CBE programming primitives [9]. However, some of them assume a strict data parallel model or adhere to pure C programming, while others provide only a minimal set of functionality of a standard library for specific applications, greatly hindering their applicability to complex object-oriented PDES systems.

Most existing CBE applications perform numerically-intensive regular computation on a large array of data following the well-known SIMD model [10]. CBE is also used to host M&S applications to offload compute-intensive functions to the SPEs [11]. It is, however, not straightforward to apply the methods employed in these applications in general-purpose PDES systems.

3. PARALLEL SIMULATION IN CD++

P-DEVS defines a model as a hierarchy of *atomic* (behavioral) and *coupled* (structural) components. The simulation is executed by several logical processes (LPs), which are specialized into *Simulators* and *Coordinators* [2]. A Simulator is paired with an atomic model to trigger the model behavior; while a Coordinator is attached to a coupled model to schedule events in the model hierarchy. Cell-DEVS [3] describes dynamic systems as cell spaces where each cell is a P-DEVS atomic model. Both P-DEVS and Cell-DEVS are realized in CD++ [4], which has been extended to support parallel simulation on distributed-memory clusters using a flat LP structure [12].

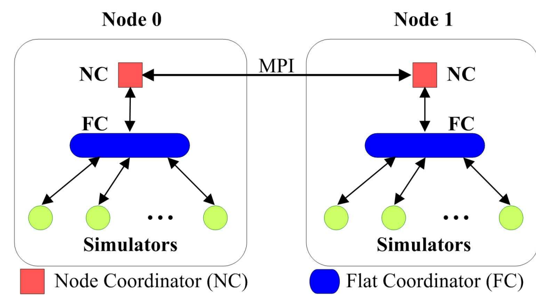

Figure 1. Flat LP Structure

Depicted in Figure 1, the sequential simulation on a node involves a *Node Coordinator* (NC), a *Flat Coordinator* (FC), and a set of *Simulators*. The NC is the local central controller and the endpoint of inter-node MPI messaging, whereas the FC synchronizes all child Simulators underneath. By eliminating the intermediate coordinators in the hierarchy, this flat LP structure significantly reduces communication overhead, while preserving the same hierarchical model definition [13]. As will be discussed later in Section 6 and 7, it also facilitates the exploitation of event-level and data-level parallelism on CBE.

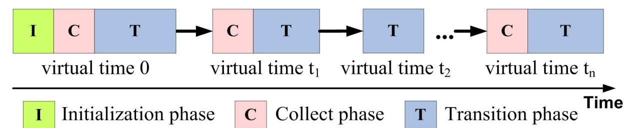

Figure 2. Multi-Phased Simulation Process

The LPs exchange messages that fall in two categories: *content messages* include the *external* (X) and *output* (Y) events that encode the model input and output data, while *control messages* include the *initialization* (I), *collect* (@), *internal* (*), and *done* (D) events that control the simulation flow. Detailed event-processing algorithms can be found in [12]. Figure 2 shows a high-level structured view of the sequential simulation on a node. At any virtual

time, the message flow between the LPs consists of an optional *collect phase* and a mandatory *transition phase*. The simulation starts with an *initialization phase* at virtual time 0. At the end of a transition phase, the NC advances the simulation to the next virtual time. This paper focuses on further parallelizing the sequential simulation on CBE so as to combine parallel simulation at cluster level with accelerated parallel simulation on each multicore node.

4. FIRE SIMULATION PROFILE

M&S has long been used to study wildfire phenomena. Cell-DEVS supports discrete-event simulation of wildfire, improving model accuracy and execution efficiency [14]. The CD++ environment allows for defining Cell-DEVS models using a built-in specification language, a feature that is especially valuable on CMP platforms since general users can focus on their modeling issues without being distracted by multicore programming details.

```
type : cell                    dim : (1024,1024)
delay: inertial                border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)
neighbors : (0,0) (0,1) (1,-1) (1,0) (1,1)
localtransition : FireBehavior

[FireBehavior]
...
rule : {(1,0)+(15.24/5.1069)} {(15.24/5.1069)*60000}
                               {(0,0)=0 and 0<(1,0)}
rule : {(0,-1)+(15.24/5.1069)} {(15.24/5.1069)*60000}
                               {(0,0)=0 and 0<(0,-1)}
...
```

Figure 3. Fire Model Definition in CD++ [14]

The fire model introduced in [14] is used in this paper for performance evaluation. This model defines a 1024 by 1024 cell space (over one million cells) to simulate a wildfire scenario based on predetermined spread rates. Figure 3 shows a skeleton of the fire model definition in CD++. The model behavior is specified using a set of transition rules, which are evaluated sequentially by active cells, also known as *imminent cells*, at each virtual time to determine their future states. Each rule has three expressions separated by spaces, defining a *postcondition*, a *delay*, and a *precondition* for the rule. A rule is fired when its *precondition* is evaluated to true; and the rule's *postcondition* will define the cell's next state, which is sent to the neighboring cells after a period calculated from the *delay* expression [4]. A full interpretation of the fire model can be found in [14]. Internally, these rules are represented as syntax trees in the main memory.

The original CD++ was ported to the PPE using the flat LP structure, resulting in a baseline version. The fire model was executed on an IBM BladeCenter QS22 with two PowerXCell 8i processors. Table 1 gives the fire simulation profile with the baseline CD++ on one PPE.

Table 1. Baseline Simulation Profile on PPE

Msg. Type	Components				
	Simulator	FC	NC	BT	Other
(I)	3.06	0.90	–	–	–
(*)	515.69	16.96	–		
(@)	8.13	55816.60	–		
(X)	11.94	0	–		
(Y)	–	94.41	–		
(D)	–	112215.00	3.25		
Sum (s)	**538.82**	**168143.87**	**3.25**	**181.57**	**134.58**
Total (s)	**169002.10**				

It is clear that the main bottleneck resides at the FC, consuming more than 99% of the total execution time. The table also shows that the FC spends most of the time on processing (@) and (D) events, during which two functions are called respectively to synchronize the Simulators at each virtual time: 1) *findImminents* finds the imminent Simulator IDs; and 2) *findMinTime* computes the next global minimum state change time among the Simulators. This synchronization task becomes time-consuming in large-scale simulations as a large amount of timing data needs to be processed. Another bottleneck resides at the Simulators, especially during the execution of (*) events where the transition rules are evaluated. It is relatively minor since the fire model uses simplified rules to approximate the real system. More complex rules would be required to obtain more precise approximation, increasing the computation intensity at the Simulators.

5. OPTIMIZATION STRATEGIES

5.1. Optimizing FC Synchronization Task

The two synchronization functions are invoked regularly by the FC: *findImminents* is called at the beginning of a collect phase, whereas *findMinTime* is called at the end of each collect and transition phases. A closer examination shows that it is unnecessary to compute the next state change time in the collect phases as these transient phases do not advance virtual time at all, which allows us to safely reduce the number of invocations of *findMinTime*. This optimization is only possible by directly exploiting the multi-phased abstraction of the simulation process, a concept that has not yet been taken into account in the original CD++ and in many other DEVS simulators alike.

Table 2. Optimized FC Synchronization Task

Msg. Type	Components				
	Simulator	FC	NC	BT	Other
(I)	3.07	0.91	–	–	–
(*)	497.40	14.38	–		
(@)	7.66	55044.50	–		
(X)	11.79	0	–		
(Y)	–	93.52	–		
(D)	–	55526.50	2.32		
Sum (s)	**519.92**	**110679.81**	**2.32**	**180.65**	**121.89**
Total (s)	**111504.59**				

Table 2 shows the performance improvement. With the number of *findMinTime* invocations reduced roughly by half in the simulation, the time required for processing (D) events at the FC is decreased by 50.5% as a result.

5.2. Preparing Simulation Data for CBE

To expose the simulation data in a CBE-friendly way, the following strategies are applied.

1. The Simulator IDs are allocated continuously from 0 to (N-1), where N is the total number of Simulators.

2. The FC uses an integer ***Time Array*** **(TA)** to hold the timing data of all child Simulators, where the array indexes serve as Simulator IDs. It also uses an integer ***Imminent ID Array*** **(IA)** to hold the imminent Simulator IDs found in *findImminents*. Both arrays are 128-byte aligned for efficient DMA transfer.

3. Each type of CD++ events is encoded in 32 bytes, and a flat 128-byte aligned ***event buffer*** is used to pass events between the FC and Simulators. Each buffer entry has an adjustable size of 32 events, allowing multiple events to be sent to an SPE with one DMA transfer. This approach also removes most of the events from the central Future Event List (FEL), minimizing event queue operation cost.

4. The state data included in the Simulator-atomic pairs are repacked in a flat 128-byte aligned ***state buffer***, where each buffer entry has an adjustable size of 512 bytes.

5. The syntax trees derived from the transition rules are converted to a sequence of floating-point values in postfix format and stored in a flat 128-byte aligned ***rule buffer*** for efficient rule evaluation on the SPEs.

Table 3 gives the simulation profile with the optimized CD++, which runs 13.79 times faster than the baseline version. The biggest improvement comes from the FC whose data-intensive synchronization task benefits the most from the enhanced data locality. The bootstrap time (BT) is reduced by 50% as most of the simulation data are now allocated in batches with big arrays. Using the event buffer also accelerates Simulator event execution and FEL operations ("Other") by 7.1% and 23.81% respectively.

Table 3. Optimized Simulation Profile on PPE

Msg. Type	Components				
	Simulator	FC	NC	BT	Other
(I)	2.58	0.76	–	–	–
(*)	491.68	12.60	–		
(@)	6.24	5650.61	–		
(X)	–	0	–		
(Y)	–	76.41	–		
(D)	–	5821.37	1.62		
Sum (s)	500.51	11561.75	1.62	89.10	102.53
Total (s)	12255.51				

The FC synchronization task and the Simulator event execution remain the two dominant bottlenecks, referred to as *FC Synchronization Kernel* (FSK) and *Simulator Event-processing Kernel* (SEK), covering 93.61% and 4.08% of the total execution time respectively.

6. PARALLELIZATION STRATEGIES

Figure 4 illustrates the parallelization strategy for the FSK.

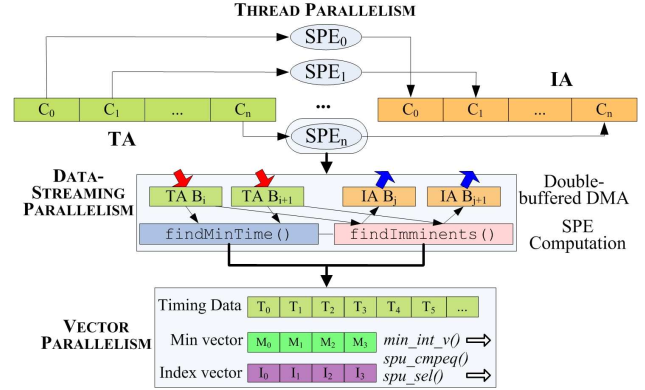

Figure 4. Parallelization Strategy for FSK

Both TA and IA contain independent data, an ideal case for exploiting data-level parallelism. The arrays are divided into multiple chunks, each of which is handled by an SPE to realize thread-level parallelism across the SPEs. On each SPE, data-streaming parallelism is utilized to process the data as a stream of blocks with regular sizes, hiding memory latency with double-buffered DMA. The synchronization functions are implemented using SPE SIMD intrinsics to achieve vector parallelism. Function *findMinTime* uses a 128-bit, 4-way integer Min Vector to scan the timing data in the current chunk. The Min Vector is then compared horizontally to obtain the local chunk-wise minimum, which is in turn sent to the PPE. Similarly, *findImminents* uses a 128-bit Index Vector to keep track of 4 indexes corresponding to the TA entries in the current chunk. The imminent Simulator IDs are sifted through the Index Vector using the PPE-determined global minimum time that has been replicated in the Min Vector. Moreover, the scanning process can be further accelerated by using multiple Min and Index Vectors as simultaneous logical threads to explore loop-level parallelism.

Unlike the LP-oriented parallelization strategy adopted in most PDES techniques [5], the SEK directly exploits the inherent event-level parallelism in the simulation. Figure 5 depicts a step-by-step view of message flow in the simulation phases, showing that two types of parallelism can be utilized without violating causal consistency.

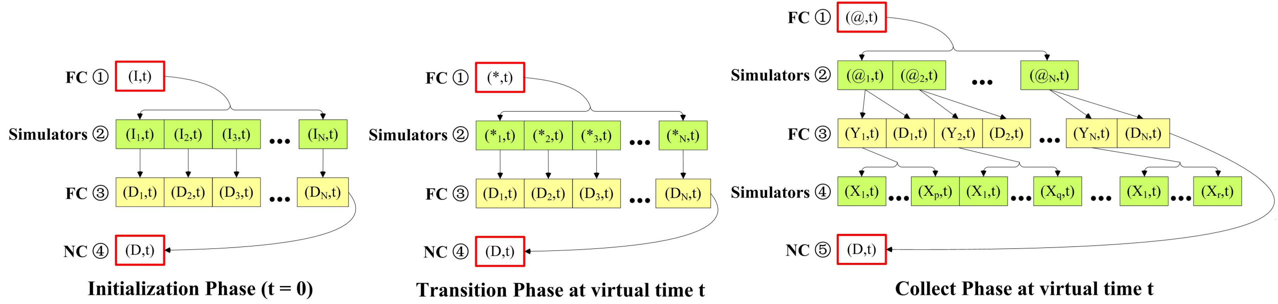

Figure 5. Event-Level Parallelism

1. **Event-embarrassing parallelism** exists between the *independent* events executed *within* each step (shaded). As there is no causal dependency between them, these events can be executed concurrently in an arbitrary order.

2. **Event-streaming parallelism** exists between the *causally-dependant* events executed in *consecutive* steps. As the output events from the preceding step serve as the inputs to the step that follows, these events can be executed concurrently in a pipelined manner.

The first and last steps in each phase provide the *fork* and *join* points for simulation synchronization. The flat LP structure minimizes the number of synchronization points, exposing the maximum degree of event-level parallelism.

To parallelize the SEK, the Simulator event-processing algorithms are implemented using SPE SIMD intrinsics whenever possible, and the performance is enhanced with loop unrolling, branch hints, and proper data alignment. Thread-level parallelism is applied across the SPEs where each SPE hosts an instance of the SEK, which processes a stream of events scheduled on the PPE. The PPE event-scheduling algorithm dispatches one or more independent events targeting the same Simulator to an SEK at a time to achieve event-embarrassing parallelism, while event-streaming parallelism is realized by executing causally-dependent events between the SPEs and the PPE in a two-stage pipeline. The simulation data are transferred with double-buffered DMA to tap data-streaming parallelism.

7. COMPUTING TECHNIQUE

Figure 6 shows an overview of the computing technique. During bootstrap, the PPE main thread spawns a helper thread, which in turn creates a set of SPE threads (one on each SPE). The SPEs are divided into two groups: one for the FSKs and the other for the SEKs. More SPEs should be reserved for the FSKs in large-scale simulations with moderate model complexity, while more SPEs are needed to speed up the SEKs for medium-sized models with complex behavior. The two PPE threads communicate with each other through the shared FEL, which only holds the events passed between the NC and the FC. With this computing technique, the Simulators are turned into *virtual LPs* in the sense that all of them share a limited group of SEKs, and the mapping of imminent Simulators to the SEKs is determined dynamically at each virtual time throughout the simulation.

The FSKs are invoked in a RPC (Remote Procedure Call) style. Each synchronization function is assigned a unique integer ID, which is sent to the FSKs through the inbound mailbox channel. The local chunk-wise minimum times obtained in *findMinTime* are then returned to the PPE to be merged into the global minimum. Note that, before calling *findImminents*, the current global minimum is compared with the previously obtained local minimums to ensure that only those FSKs that actually found the global minimum value are involved in the computation.

The SEKs are orchestrated as follows. After processing events in the FEL at the beginning of a simulation phase, the FC writes the generated events for the Simulators directly into the event buffer based on the Simulator IDs. The index of a modified buffer entry (Job ID) is inserted into a pending job queue, thus mapping a Simulator to a chosen SPE. As the events executed by the SEKs at any virtual time have similar compute intensity, simple yet effective policies (e.g., round-robin or shortest-queue-first) can be used to achieve fine-grained load-balancing among the SEKs. The FC then notifies the SEKs about the number of pending jobs via mailbox channel. As a result, the SEKs fetch the job IDs in groups, double-buffering the data of the next job while executing the current one. After each job execution, the updated state and output events are transferred back to the main memory buffers using double-buffered DMA. The SEKs send status signals to the PPE periodically during the execution, allowing the FC to process the output events in parallel. Finally, the FC sends events to the NC via the FEL at the end of the current phase. In the simulation, the PPE main thread handles file I/O and/or inter-node messaging, overlapping computation and communication to enhance performance.

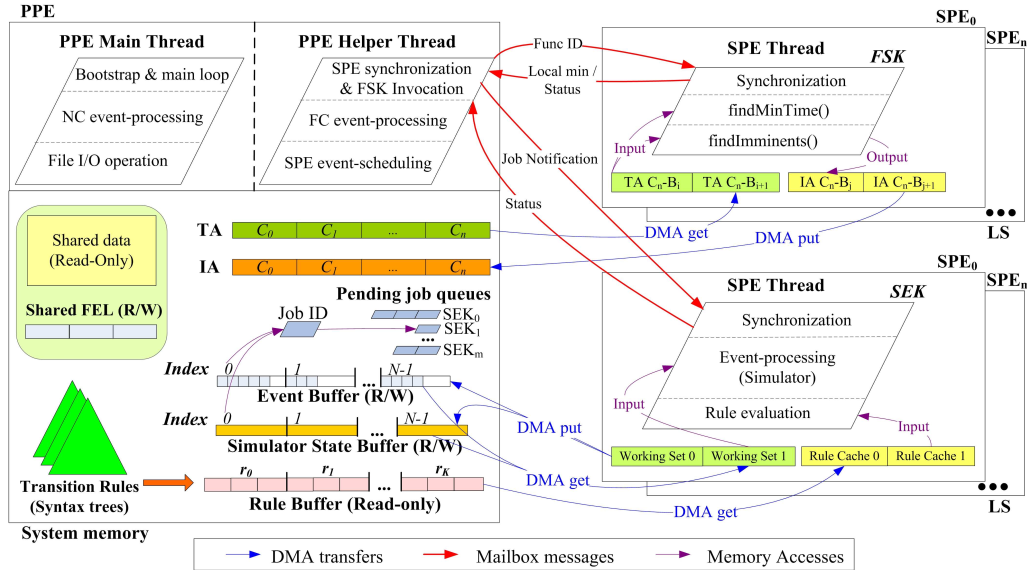

Figure 6. An Overview of the Computing Technique

8. EXPERIMENTAL RESULTS

This section analyzes the performance of the FSK, while the SEK is still under implementation. Figure 7 shows the speedups of the FSK itself over the optimized PPE version. Both functions attained super-linear speedups due to SIMD vectorization and reduced memory latency with double buffering. Function *findImminents* performed better for two reasons: 1) a FSK is called in *findImminents* only if it finds the global minimum, while all the FSKs are called in *findMinTime*; 2) *findMinTime* is called in place by the FC, whereas *findImminents* is called in advance by the NC once the next simulation time is determined, thus overlapping the computation at the PPE and the SPEs. Overall, the FSK achieved a speedup of 25.04 on 16 SPEs.

Figure 8 shows the FSK impact on the overall simulation speedups over the PPE-optimized CD++. Super-linear speedups were attained on up to 7 SPEs. The speedup grows a bit slower after that for two reasons: 1) an increasing number of SPEs leads to higher orchestration overhead; 2) frequent DMA contention and channel stalls occur when all the FSKs transfer data at the same time.

Figure 9 gives the overall simulation time attained on both CBE and Intel E6400. On CBE, the simulation time was reduced from over 3 hours with the PPE-optimized CD++ to just 20 minutes with FSK on 16 SPEs. Comparing to the baseline and optimized CD++, the simulation achieved speedups up to 134.34 and 9.74 on CBE and up to 41.23 and 1.92 on E6400 respectively. The heavy-iron E6400 has a much larger cache (64KB L1, 2MB L2) than the PPE (32KB L1, 512KB L2), allowing the E6400 to handle the FSK's intensive memory I/O at much lower cache miss rates – a main reason for the performance difference between the two optimized versions.

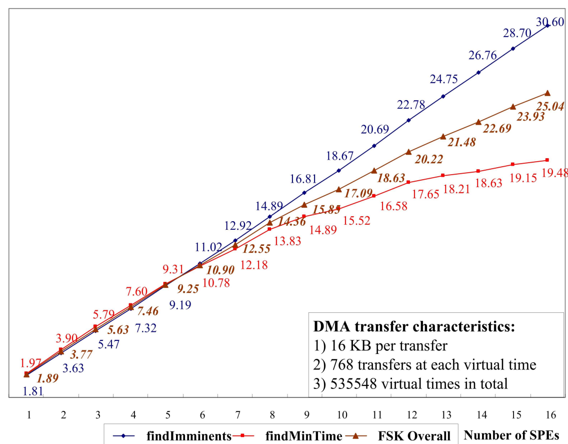

Figure 7. FSK Speedups over PPE Version

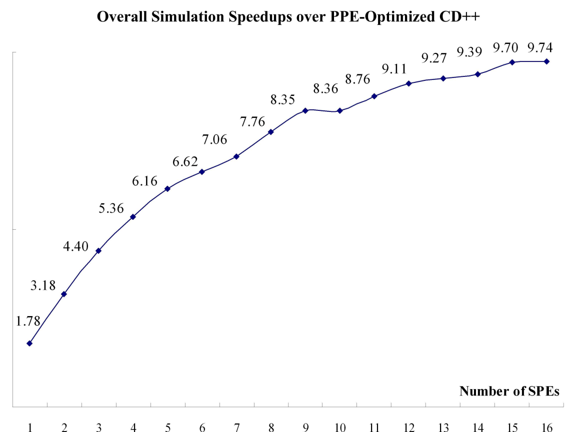

Figure 8. FSK Impact on Overall Simulation

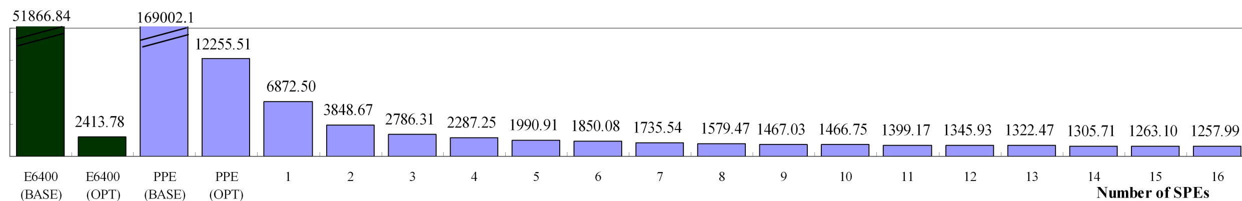

Figure 9. Total Execution Time Attained on IBM CBE and Intel E6400

9. CONCLUSION / FUTURE WORK

This paper presents a computing technique for efficient parallel simulation of large-scale discrete-event models on the CBE platform. Based on the DEVS formalism, the technique tackles all major performance bottlenecks by combining multi-dimensional parallelism and various optimizations in the simulation. Moreover, the methods presented in this work can also be readily applied to other CMP and shared-memory multiprocessors. Promising results have been produced in our experiments, attaining speedups up to 134.34 and 41.23 over the baseline implementation on the PPE and on the Intel E6400 processor respectively. The technique not only allows a broad community of discrete-event simulation users to harness CBE potential without being distracted by multicore programming, but also provides insight on porting legacy software to current and future multicore platforms. We are implementing the SEK in CD++ and integrating cluster-based parallel simulation with CBE-accelerated parallel simulation on hybrid systems.

ACKNOWLEDGEMENTS

This work was supported in part by NSERC, the MITACS Accelerate Ontario Program, Canada, and by the IBM T. J. Watson Research Center, NY. We would like to thank Dr. D. P. Scarpazza and Dr. L. K. Liu from IBM Watson Center for their insightful feedback.

REFERENCES

[1] B. P. Zeigler, H. Praehofer, T. G. Kim, THEORY OF MODELING AND SIMULATION, 2nd Edition, Academic Press, London, 2000.

[2] A. C. Chow, B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism". Proceedings of Winter Simulation Conference, Lake Buena Vista, FL, pp. 716-722, 1994.

[3] G. Wainer, N. Giambiasi, "N-dimensional Cell-DEVS models". *Discrete Event Dynamic Systems*, Vol. 12, No. 2, pp. 135-157, 2002.

[4] G. Wainer. "CD++: a toolkit to develop DEVS models". *Software: Practice and Experience*, Vol. 32, No. 13, pp. 1261-1306, 2002.

[5] R. M. Fujimoto, PARALLEL AND DISTRIBUTED SIMULATION SYSTEMS, John Wiley & Sons, New York, 2000.

[6] J. A. Khale, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy, "Introducing to the Cell multiprocessor". *IBM Journal of Research and Development*, Vol. 49, No. 4/5, pp. 589-604, 2005.

[7] F. Blagojevic, X. Feng, K. W. Cameron, D. S. Nikolopoulos, "Modeling multigrain parallelism on heterogeneous multi-core processors: A case study of the Cell BE". *High Performance Embedded Architectures and Compilers*, LNCS 4917, pp. 38-52, Springer, Berlin, 2008.

[8] T. J. Knight, J. Y. Park, M. Ren, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, P. Hanrahan, "Compilation for explicitly managed memory hierarchies". Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Jose, CA, pp. 226-236, 2007.

[9] M. D. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform". Proceedings of GSPx Multicore Applications Conference, Santa Clara, CA, 2006.

[10] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, "Scientific computing kernels on the Cell processor". *International Journal of Parallel Programming*, Vol. 35, No. 3, pp. 263-298, 2007.

[11] V. Agarwal, L. K. Liu, D. A. Bader, "Financial modeling on the Cell Broadband Engine". Proceedings of IEEE International Symposium on Parallel and Distributed Processing, Miami, FL, pp. 1-12, 2008.

[12] Q. Liu, G. Wainer, "Parallel environment for DEVS and Cell-DEVS models". *SIMULATION*, Vol. 83, No. 6, pp.449-471, 2007.

[13] E. Glinsky, G. Wainer, "New parallel simulation techniques of DEVS and Cell-DEVS in CD++". Proceedings of Annual Simulation Symposium, Huntsville, AL, pp. 244-251, 2006.

[14] G. Wainer. "Applying Cell-DEVS methodology for modeling the environment". *SIMULATION*, Vol. 82, No. 10, pp. 635-660, 2006.