Accelerating Large-scale DEVS-based Simulation on the Cell Processor

Qi Liu and Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University Centre on Visualization and Simulation (V-Sim)
Carleton University, Ottawa, ON, Canada K1S 5B6
{liuqi, gwainer}@sce.carleton.ca

Keywords: Discrete event simulation, parallel simulation, multicore computing, DEVS, Cell-DEVS, Cell processor

Abstract

This paper presents a new technique for efficient parallel simulation of large-scale DEVS-based models on the IBM Cell processor, which has one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). By taking a performance-centered approach, the technique allows for exploitation of multi-dimensional parallelism to overcome the bottlenecks in the simulation process. We illustrate the underlying design methodology with detailed simulation profiles. Our preliminary experiments have already produced promising results, accelerating the baseline PPE-only simulation of a fire model and a flood model by a factor of up to 70.6 and 83.32 respectively. The technique not only enables DEVS users to harness the potential of the Cell processor without being distracted by the technical complexity of multicore programming, but also provides insights on migration of legacy software to current and future multicore platforms.

1. INTRODUCTION

The physical limitations of heat dissipation, memory latency, and gate density are pushing the microprocessor industry towards multicore Chip Multiprocessor (CMP) designs. One latest example is the IBM Cell processor [1], which exhibits enormous potential for scientific computing [2] and has been used in the Roadrunner project to build a petascale supercomputer for the Los Alamos National Laboratory [3]. The Cell processor adopts a heterogeneous CMP architecture with nine independent cores: one dual-threaded Power Processing Element (PPE) and eight specialized co-processors called Synergistic Processing Elements (SPEs). The PPE uses a conventional cache hierarchy to access system main memory and provides top-level thread control for a parallel application, whereas each SPE can only directly access a small, non-coherent, on-chip 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. In addition, the cores can also communicate 32-bit messages with each other via the on-chip interconnect bus channels such as mailboxes and signals. Furthermore, the SPEs support both scalar and 128-bit SIMD (Single Instruction, Multiple Data) computations, which can be applied at 2, 4, 8, and 16-way granularities. All these features make the Cell processor an attractive platform to study new computing paradigms for high-performance scientific applications on the emerging CMP architectures. On the flip side, the asymmetric design of heterogeneous cores with explicit memory control requires careful reconsideration of existing algorithms from a data-flow perspective to attain optimal execution performance.

The Discrete Event System Specification (DEVS) formalism [4] provides a sound theoretical foundation for describing discrete-event systems. Numerous extensions to DEVS have been proposed in the literature. P-DEVS [5] improves the mechanism for handling simultaneous events. Cell-DEVS [6] allows for defining n-dimensional cell spaces as discrete-event models where each cell is a basic DEVS model component. Both P-DEVS and Cell-DEVS are implemented in CD++ [7], an object-oriented modeling and simulation (M&S) environment that has been used to solve a variety of sophisticated problems (e.g., [8][9][10]).

With the growing size and complexity of the system, the simulation is increasingly time-consuming. Parallel Discrete-Event Simulation (PDES) is widely used to speed up discrete-event systems [11]. Traditionally, PDES exploits concurrent activities at different model components by partitioning the simulation onto multiple nodes of a cluster. Although this coarse-grained parallelization strategy has achieved success in improving performance, other types of fine-grained parallelism available on multicore processors (e.g., thread-level, event-level, and data-level parallelism) remain untapped in most existing algorithms.

As multicore computing becomes pervasive, there is an acute need for bridging the gap between PDES algorithms for conventional clusters and those for CMP platforms. In this work, we seek to narrow this gap by exploring new forms of fine-grained parallelism and proposing a novel technique that combines multi-dimensional parallelism coherently in large-scale DEVS simulations, while hiding the technical details of multicore programming from users. By taking a performance-centered approach, the technique addresses all major bottlenecks in the simulation process. The underlying design methodology is illustrated with detailed simulation profiles of two example Cell-DEVS models (each has a cell space of over one million cells), which simulate wildfire propagation and flooding scenarios

respectively. Although the proposed technique has not yet been fully implemented thus far, the algorithms currently available have produced very promising results, attaining overall speedups up to 70.6 and 83.32 in the fire and flood simulations respectively over the baseline implementation on PPE. We believe that the technique not only exposes the potential of Cell processor to DEVS users, but also offers valuable insights for other application developers who intend to port existing legacy software to CMP platforms.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 recaps DEVS simulation in CD++. Section 4 analyzes the performance bottlenecks. Section 5 covers the parallelization strategies, and Section 6 proposes the computing technique. The experimental results are discussed in Section 7. Section 8 concludes the paper.

2. RELATED WORK

As mentioned earlier, existing PDES algorithms usually take a coarse-grained parallelization strategy at the cluster level without paying much attention to other forms of fine-grained parallelism available on modern multicore platforms. Multigrain parallelism has been studied in the context of scientific and multimedia applications [12][13]. New techniques are required to exploit multi-dimensional parallelism in large-scale PDES on the Cell processor.

Although different programming models and strategies were investigated to improve programmability on the Cell processor [1][14], significant efforts are still needed to specialize and integrate them to address PDES peculiarities.

Compiler-assisted vectorization is one way to facilitate software development on Cell [15][16]. Without a deep understanding of high-level application logic, this technique is still inadequate on its own for complex PDES systems involving irregular computation that must respect the causal dependency among individual events.

Several middleware frameworks have been developed on top of the Cell programming primitives [17][18]. Nonetheless, some of them adopt a strict data parallel model or adhere to pure C programming, while others tailor the functionality of a standard library for specific applications. These limitations greatly hinder their applicability to complex object-oriented PDES systems.

Most applications developed on the Cell processor perform numerically-intensive computation on a large array of data, a SIMD-oriented model that has proven well-suited for exploiting data-level parallelism [19]. Recently, M&S applications have also been implemented on Cell to offload compute-intensive functions to the SPEs [20]. It is, however, not straightforward to apply the methods used in these applications in general-purpose PDES systems.

3. P-DEVS AND CELL-DEVS SIMULATION IN CD++

The P-DEVS formalism supports the construction of hierarchical and modular models. It describes a model as a hierarchy of *atomic* (behavioral) and *coupled* (structural) components. A P-DEVS atomic model uses a bag of inputs to support the execution of multiple simultaneous events, combining the functionality of multiple external transitions into a single one [5]. The simulation is driven by a set of logical processes (LPs), which are specialized into *Simulators* and *Coordinators*. A Simulator is in charge of triggering an atomic model's behavior, while a Coordinator is responsible for scheduling events in the model hierarchy. The Cell-DEVS formalism describes dynamic systems as n-dimensional cell spaces where each cell is a P-DEVS atomic model, allowing for efficient asynchronous execution and providing a natural mechanism for defining temporal and spatial relations between model components.

Both P-DEVS and Cell-DEVS are implemented in CD++ [7]. To enable users to focus on their modeling issues without being distracted by the details of simulation implementation, CD++ provides a built-in specification language to code the behavior of Cell-DEVS models with a set of transition rules. Internally, these transition rules are represented as syntax trees, which are loaded into main memory during simulation bootstrap for evaluation by the cells. This extra level of abstraction is especially valuable on multicore platforms since it hides the complexity of multicore programming, allowing users to gain performance with minimum knowledge of the execution environment.

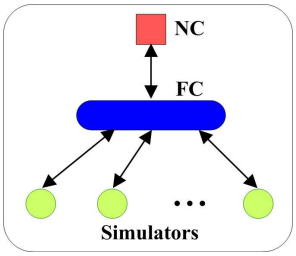

Figure 1. Flat LP structure

Recently, CD++ has been extended to support parallel simulation on cluster systems based on a flat LP structure [21]. By eliminating the intermediate Coordinators in the hierarchy, this flat LP structure not only reduces communication overhead, but more importantly exposes the maximum degree of event-level parallelism and facilitates the exploitation of data-level parallelism during simulation synchronization, as will be discussed in Section 5. 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 central controller on the host node, whereas the FC synchronizes all child Simulators.

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. The (I) events initialize the LPs at the beginning of the simulation. The (@) and (*) events trigger the output and

DEVS state transition functions respectively in the atomic models, and the (D) events carry the model timing information for simulation synchronization.

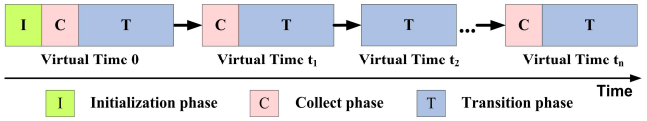

Figure 2. Multi-phased simulation process

Figure 2 shows a structured view of the simulation process [21]. At any virtual time, there is a mandatory *transition phase* and an optional *collect phase*. The simulation starts with an *initialization phase* at virtual time 0. At the end of a transition phase, the NC determines the next simulation time and sends link messages to the FC, advancing the simulation on the node. Based on these concepts, this paper focuses on parallelizing the sequential simulation on the Cell processor.

4. PERFORMANCE BOTTLENECKS

In large-scale, high-resolution simulation of complex systems, the simulation performance is primarily dominated by the *size* and *complexity* of the model. The former determines the synchronization overhead required for the FC to process the timing information of the Simulators, while the latter decides the computation intensity needed for generating model behavior (rule evaluation in the case of Cell-DEVS models). To show the impact of these two factors, the original CD++ was ported to the PPE core using the flat LP structure, resulting in a baseline implementation.

M&S has been used in environmental sciences to simulate wildfire propagation and flooding phenomena for several years. In this work, the fire model introduced in [9] and the flood model available at [22] are used as examples for performance testing. Both models define a 1024 by 1024 cell space (over one million cells) to simulate wildfire propagation and flooding scenarios over 50 virtual hours based on a set of environmental parameters. The experiments were carried out on an IBM BladeCenter® QS22 with 3.2 GHz IBM PowerXCell 8i processors.

Table 1. Baseline simulation profiles on the PPE core

Fire Model							
	(I)	(*)	(@)	(X)	(Y)	(D)	Sum (s)
Simulator	3.07	**497.4**	7.66	11.79	–	–	519.92
FC	0.91	14.38	**55044.5**	0	93.52	**55526.5**	110679.81
NC	–	–	–	–	–	2.32	2.32
Bootstrap	–						180.65
Other	–						121.89
Total Execution Time (s)							**111504.59**
Flood Model							
	(I)	(*)	(@)	(X)	(Y)	(D)	Sum (s)
Simulator	3.09	**144.1**	4.64	4.89	–	–	156.73
FC	0.93	3.40	**30413.2**	0	34.48	**30690.7**	61142.7
NC	–	–	–	–	–	1.22	1.22
Bootstrap	–						77.62
Other	–						54.25
Total Execution Time (s)							**61432.53**

Table 1 gives the resulting simulation profiles obtained on the PPE. As we can see, the FC constitutes the main bottleneck, consuming 99.3% and 99.5% of the total execution time in the fire and flood simulations respectively. A closer look at the message-wise decomposition shows that the FC spends most of its time on processing (@) and (D) messages during which the child Simulators are synchronized at each virtual time. This bottleneck is prominent in large-scale simulations since the FC needs to process a large amount of timing data in two synchronization functions. Function `findMinTime` is called during the processing of (D) messages to compute the next minimum state change time among the Simulators, while `findImminents` is called during the execution of (@) messages to find the imminent Simulator IDs.

Another bottleneck resides at the Simulators, especially the execution of (*) messages where the transition rules are evaluated. This bottleneck appears to be minor in the tested models because they use simplified rules to approximate the dynamic behavior of the real systems. More complex rules would be required to obtain more precise approximation, leading to higher computational cost at the Simulators.

In the following, we present new parallelization strategies for the FC synchronization task and the execution of all types of events targeting the Simulators, referred to as the *FC Synchronization Kernel* (FSK) and the *Simulator Event-processing Kernel* (SEK), on the Cell processor.

5. PARALLELIZARTION STRATEGIES

5.1. Strategy for the FSK

The FSK consists of the two synchronization functions. Originally, the FC uses a standard C++ map (<SimulatorID, VirtualTime>) to keep track of the next state change time scheduled by the Simulators. Before parallelizing the FSK on Cell, this timing data needs to be prepared in a way that allows us to reconsider the FC synchronization task from a data-flow perspective, as described below.

- A new ID allocation scheme is used to allocate positive IDs for the atomic models and their associated Simulators continuously from 0 to (N-1), where N is the total number of Simulators created in the simulation. On the contrary, the coupled models and the Coordinators use negative IDs.
- The FC uses an integer ***Time Array*** (TA) to hold the timing data of all child Simulators, where the array indexes serve as the Simulator IDs. This approach offers three main benefits. First, it reduces the amount of data by half when compared to the original map structure. Secondly, it allows us to align the data on cache-line (128 bytes) boundaries and to pass the effective address of TA to the SPEs for efficient DMA transfer. Thirdly, it enhances data locality with reduced memory contention and cache miss even when the computation is carried out on the PPE alone. Owning to the flat LP structure, the timing data can be concentrated in a single array, greatly facilitating the parallelization of the FSK.

- The FC also uses a 128-byte aligned integer ***Imminent ID Array*** (IA) to hold the IDs of imminent Simulators found in `findImminents`. Since only a fraction of the Simulators are imminent at any virtual time, this array is terminated by a -1 so that the FC can retrieve the imminent IDs without performing a full traversal.

With this data organization, Figure 3 summarizes the parallelization strategy for porting the FSK to the SPE cores.

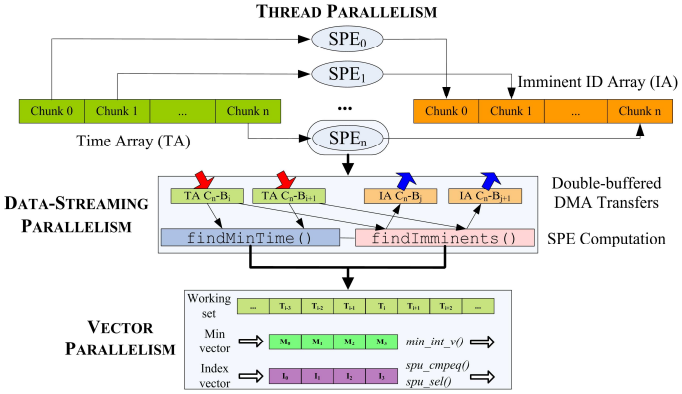

Figure 3. Parallelization strategy for the FSK

Both TA and IA contain independent data, an ideal case for exploring data-level parallelism. To process the time values in parallel, TA is divided into multiple chunks, each of which is handled by an SPE independently to realize thread-level parallelism across multiple SPEs. On each SPE, data-streaming parallelism is utilized to process the chunk of data as a stream of blocks (or working sets) with regular sizes. The synchronization functions are decoupled from the FC and implemented in C using explicit SPE SIMD intrinsics to exploit vector parallelism.

```
Input: MinVector, TABlock[2]

1. when function findMinTime is invoked
2.   Calculate numOfDMATrans based on chunk size & block size
3.   Start an inbound DMA to transfer the 1st block of TA data to TABlock0
4.   for i = 0 to (numOfDMATrans - 1) do
5.     Wait for ith inbound DMA to complete in current TABlock
6.     Start (i+1)th inbound DMA to transfer next block of TA data to next TABlock
7.     for j = 0 to (block size/4 - 1) do
8.       Compare MinVector to 4 values in current TABlock with min_int_v
9.     endfor
10.  endfor
11.  Calculate chunk-wise min_time = min_vec_int4(MinVector)
12.  Send min_time to PPE through mailbox channel
13. end when
```

Figure 4. A skeleton of function findMinTime

On each SPE core, as illustrated in Figure 4, function `findMinTime` uses a 128-bit, 4-way integer Min Vector to scan the virtual time values that are DMA transferred into the current working set (line 7-9). When the full chunk of data is processed, the Min Vector contains the 4 minimum values obtained in the 4 ways. These values are then compared horizontally to calculate the chunk-wise minimum (line 11), which is sent to the PPE code through the outbound mailbox channel (line 12).

As we can see in Figure 5, `findImminents` replicates the PPE-determined global minimum time in the Min Vector (line 2). It uses another 128-bit Index Vector to keep track of 4 Simulator IDs corresponding to the entries in the current working set when the time values are sifted with the Min Vector (line 9-14). At the end of the function, a status value is sent to the PPE (line 22), indicating that the imminent IDs are available in the IA chunk.

```
Input: MinVector, IndexVector, TABlock[2], IABlock[2], baseId

1. when function findImminents is invoked
2.   MinVector = spu_splats( spu_read_in_mbox() )
3.   IndexVector = {baseId, baseId+1, baseId+2, baseId+3}
4.   Calculate numOfDMATrans based on chunk size & block size
5.   Start an inbound DMA to transfer the 1st block of TA data to TABlock0
6.   for i = 0 to (numOfDMATrans - 1) do
7.     Wait for ith inbound DMA to complete in current TABlock
8.     Start (i+1)th inbound DMA to transfer next block of TA data to next TABlock
9.     for j = 0 to (block size/4 - 1) do
10.      Compare MinVector to 4 values in current TABlock with spu_cmpeq
11.      Select the indexes of equal values from IndexVector with spu_sel
12.      Write the selected indexes continuously into current IABlock
13.      IndexVector = spu_add(IndexVector, 4)
14.    endfor
15.    if current IABlock is not full then
16.      Terminate current IABlock with -1
17.    endif
18.    Wait for (i-1)th outbound DMA to complete in previous IABlock
19.    Start an outbound DMA to transfer the current IABlock to IA
20.  endfor
21.  Wait for the last outbound DMA to complete
22.  Notify PPE of completion through mailbox channel
23. end when
```

Figure 5. A skeleton of function findImminents

Both functions use doubled-buffered inbound and outbound DMA transfers to hide memory latency. By using multiple Min and Index Vectors as simultaneous logical threads, loop-level parallelism can be exploited to further accelerate the computation.

5.2. Strategy for the SEK

Although event-processing is fundamental in discrete-event simulations, direct and explicit exploitation of fine-grained event-level parallelism is still uncommon in the literature. A key challenge is that the parallel computation must respect the causal dependency among individual events. Figure 6 gives a phase-wise step-by-step view of P-DEVS simulation with the flat LP structure, showing that two types of event parallelism can be exploited following a conservative approach without violating causal consistency.

1. **Embarrassing parallelism** exists between the *independent* events executed *within* each step at the FC and the Simulators. Since there is no causal dependency between these events, they can be processed concurrently in an arbitrary order.
2. **Event-Streaming parallelism** exists between *causal-dependent* 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.

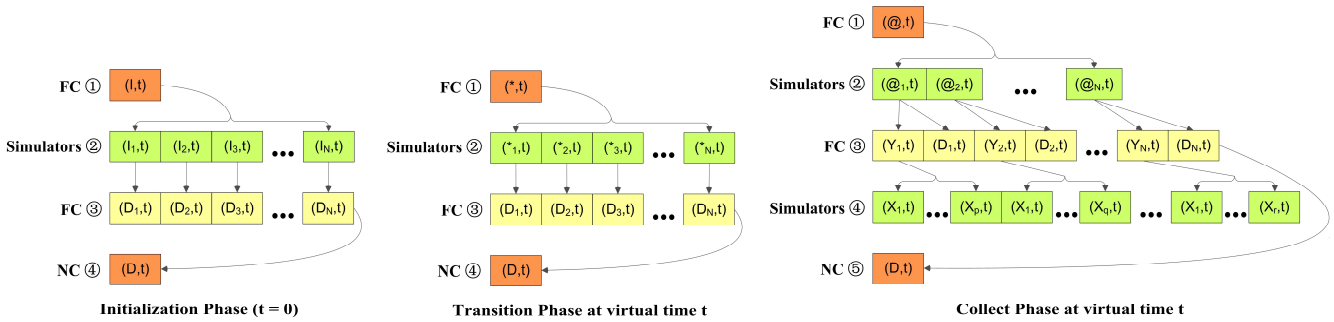

Figure 6. Event-level parallelism in P-DEVS simulation

The first and last steps in each phase serve as *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.

In order to port the SEK to SPE cores, the simulation data is reorganized as follows.

- The original C++ hierarchy of event classes is replaced with a uniform 32-byte C struct, which encodes the data of all types of events in a compact way.
- The state data encapsulated in an atomic model and its associated Simulator is repacked in a C struct of 512 bytes, and all the state data is stored in a flat 128-byte aligned array (***state buffer***), one entry for each 512-byte struct.
- Each transition rule defined in a Cell-DEVS model is converted to a sequence of float numbers organized in a postfix format, where a syntax node is represented by 2 float values. The resulting rules are concatenated in a flat 128-byte aligned array (***rule buffer***).
- A flat 128-byte aligned array (***event buffer***) is allocated to pass events between the FC and the Simulators. Each entry has an adjustable size of 1KB with a capacity of up to 32 slots that is dedicated to a Simulator with ID equal to the array index, as illustrated in Figure 7.

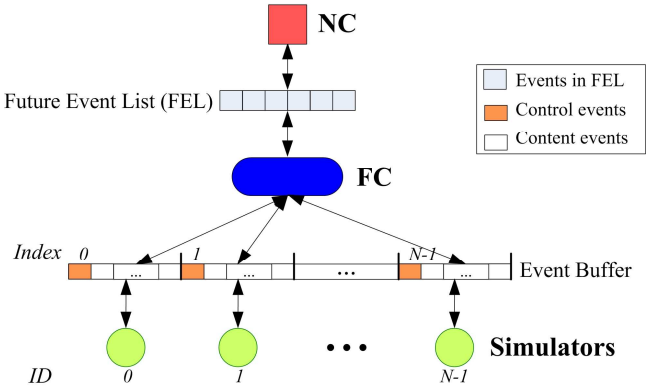

Figure 7. Decentralized event management

At any time, a Simulator and the FC may exchange exactly one control message and optionally a list of content messages. Thus, the first slot in each event buffer entry is reserved for passing the control event, while the following slots are used to hold content events, if any. The original Future Event List (FEL) is only used to send events between the FC and NC at the beginning and end of each simulation phase. Together, the FEL and the event buffer entries can be viewed as one-to-one bidirectional communication channels, forming a star topology with the hub at the FC.

This decentralized event management has several major advantages over the original FEL-centered scheme. First, multiple events targeting a Simulator can be transferred efficiently to an SPE for execution with one DMA operation. Secondly, most of the events executed in the simulation are removed from the FEL, reducing the overhead of event queue operations considerably. Thirdly, events passed between the FC and the Simulators are directly written/read in the event buffer without memory allocation/deallocation, further reducing the operational cost. Fourthly, the Simulators no longer need to define extra message bags to hold simultaneous (X) events. Instead, the slots in each event buffer entry are naturally suited for this purpose. As these (X) events are written directly into the slots by the FC, the Simulators also do not need to receive (X) messages any more, simplifying Simulators' event-processing algorithm and enhancing simulation performance. Finally, this scheme improves data locality and cache utilization even in the case of sequential simulation on traditional processors.

The SEK includes the algorithms for processing (I), (*) and (@) messages at the Simulators as well as the DEVS functions defined in the atomic models (detailed definition of these algorithms can be found in [21]). To parallelize the SEK, these algorithms are converted to efficient C code and recompiled on the SPE. Due to the irregular nature of the computation, only partial vectorization is applied using SPE SIMD intrinsics. The performance is further enhanced by loop unrolling, branch hints, and proper data aligning techniques. Thread-level parallelism is exploited across multiple SPEs, each of which hosts an instance of the SEK. An SEK executes a stream of events chosen by a PPE-side event-dispatching algorithm, which schedules one/more independent events targeting the same Simulator to an SEK at a time to realize the event-level embarrassing parallelism. On the other hand, the event-streaming parallelism is achieved by executing causal-dependent events between the SPEs and the PPE as a two-stage pipeline. Double buffering is used extensively in DMA transfers of event, state and rule data to/from the buffers to tap data-streaming parallelism.

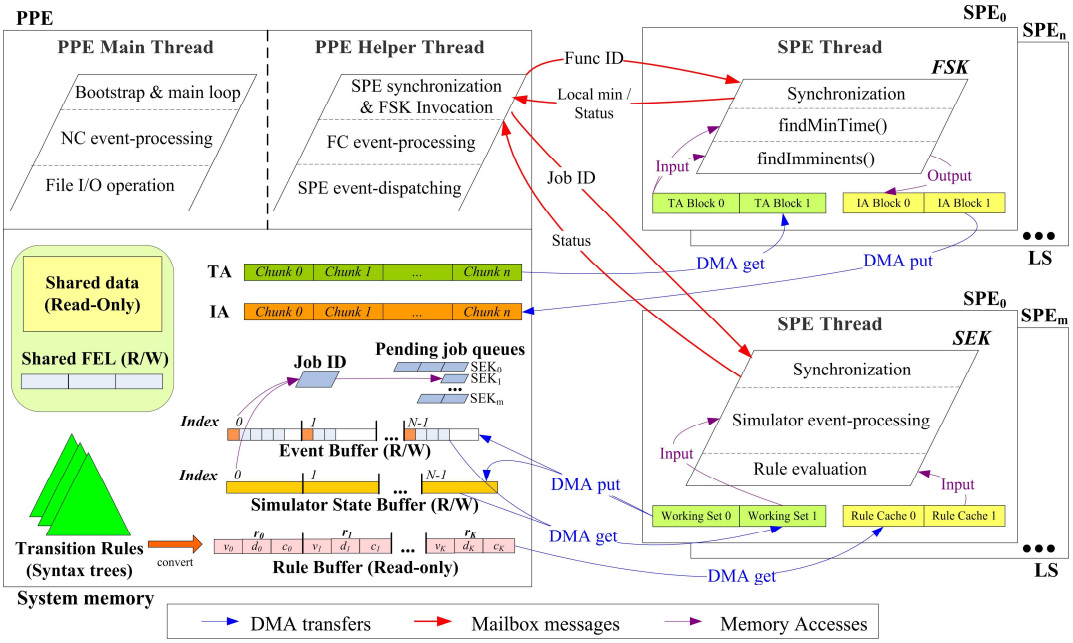

Figure 8. Overview of DEVS-based parallel simulation on Cell

6. PARALLEL SIMULATION ON CELL

As shown in Figure 8, the parallel simulation involves 10 threads on a Cell processor. During bootstrap, the PPE main thread spawns a helper thread, which in turn creates eight SPE threads (one for each SPE). The SPEs are divided into two groups, one for FSK and the other for SEK. For large-scale models with moderate complexity, more SPEs should be used to handle the synchronization task of FSK. On the contrary, for complex models with moderate sizes, the SEK requires more SPEs to speed up intensive computation at the Simulators. Based on our parallelization strategies, the Simulators and their associated atomic models are ***virtualized*** in the sense that all of them share a limited set of SPE threads to fulfill their functionalities, and the mapping of imminent Simulators onto the SPEs at each virtual time is determined dynamically in the simulation.

```
1. when the last (D) message is processed at the FC (end of a transition phase)
2.   Send 0 to each FSK via in-bound mailbox channel
3.   Wait for local min times sent back from all FSKs
4.   Record the local min times for later comparison
5.   Merge local min times to obtain the global min next state change time
6. end when

7. when a (@) message is processed at the FC (beginning of a collect phase)
8.   for each FSK with recorded local min time = current global min time do
9.     Send 1 and the current global min time to the FSK via 2 mailbox messages
10.  end for
11.  Wait for status values returned from the FSKs that have been invoked
12.  Fetch imminent IDs made available in IA
13. end when

14. when the simulation terminates
15.  Send 2 to each FSK via in-bound mailbox channel
16. end when
```

Figure 9. A skeleton of FSK orchestration algorithm

The FSKs are invoked in a RPC (Remote Procedure Call) style. Each function included in the FSK has an ID: 0 for `findMinTime`, 1 for `findImminents`, and 2 for FSK termination. As given in Figure 9, the orchestration algorithm for FSK is quite straightforward. Note that, before calling `findImminents`, the current global minimum time, as determined by the NC, is compared to the recorded local minimums previously found by the FSKs (line 8) to ensure that only those FSKs that actually found the global minimum are involved in the computation.

```
1. when a simulation phase starts
2.   FC executes one/more FEL events received from the NC
3.   FC writes generated events for Simulators in corresponding event buffer entries
4.   FC invokes an event-dispatching algorithm to schedule events for Simulators
5.   Event-dispatching algorithm inserts Job IDs into pending job queues
6.   while pending job queues are not empty do
7.     for each SPE that hosts an instance of SEK do
8.       if empty entries become available in the SPE inbound mailbox channel then
9.         Send one/more Job IDs to the SPE as mailbox messages
10.      end if
11.    end for
12.    if finished Job IDs are sent back from SPE outbound mailbox channels then
13.      FC executes output events in event buffer entries based on finished Job IDs
14.    end if
15.  end while
16.  FC waits for the remaining SEKs to complete
17.  FC generates one/more events for the NC and inserts them into FEL
18. end when
```

Figure 10. A skeleton of SEK orchestration algorithm

Figure 10 shows the SEK orchestration algorithm. At the beginning of each simulation phase, the FC executes FEL events scheduled by the NC, generating a sequence of events for the Simulators. These events are directly written into the event buffer entries based on Simulator IDs. The index of a modified event buffer entry serves as the Job ID, which is then inserted into one of the pending job queues by an event-dispatching algorithm (line 5), thus mapping a Simulator to a chosen SPE. Since the events executed by the

SEKs at any given time have similar computational intensity, simple yet effective scheduling policies such as shortest-queue-first can be used to achieve fine-grained load-balancing among the SEKs. Multiple pending Job IDs can be sent to an SEK once empty entries become available in the SPE inbound mailbox channel (line 9), allowing the SEK to pre-fetch data for the next job while executing the current one. On the SPE side, an SEK fetches data from the state, event, and rule buffers with double-buffered DMA transfers. At the end of event processing, the SEK puts the updated data back to the buffers and sends the finished Job ID to the PPE helper thread, allowing the FC to execute the output events from the event buffer concurrently (line 13). Finally, the FC sends events to the NC via the FEL when all output events from the SEKs are processed (line 17), finishing the current phase. During the simulation, the PPE main thread performs file I/O and remote communication with other nodes, if necessary, overlapping computation with file I/O and communication to enhance performance.

7. EXPERIMENTAL RESULTS

The baseline CD++ implementation has been enhanced with the data optimization strategies as presented in Section 5. The resulting PPE-optimized sequential version was then parallelized to execute the FSK across multiple SPEs, while the parallelization of the SEK is still underway. This section analyzes the performance impact of the FSK in the fire and flood simulations on IBM BladeCenter® QS22.

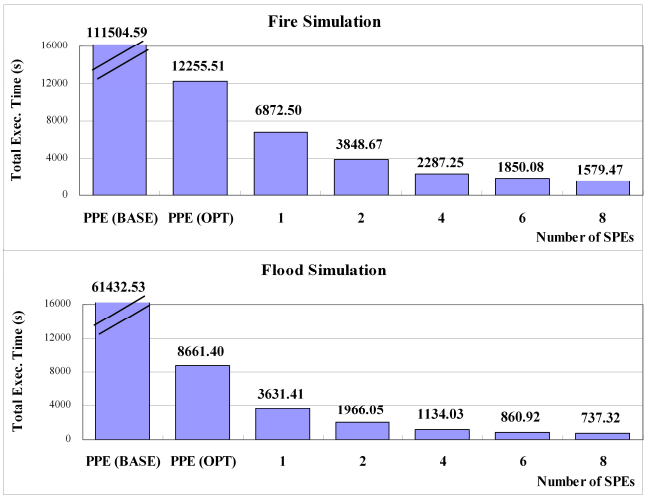

Figure 11. FSK data optimization and parallelization impact

Figure 11 summarizes the total simulation time attained by the baseline and optimized CD++ on PPE as well as the parallelized FSK on 1 to 8 SPEs. With the data optimization alone, the optimized CD++ runs 9.1 and 7.09 times faster than the baseline version in the fire and flood simulations respectively. Executing the FSK on 8 SPEs further reduces the fire and flood simulation time from over 3.4 and 2.4 hours with the optimized CD++ to just 26.3 and 12.3 minutes respectively. The overall simulation speedup over the optimized CD++ is given in Figure 12. In most cases, the parallelized FSK achieves super-linear speedups thanks to the multi-dimensional parallelization strategy. The flood model achieves higher speedups than the fire model since the synchronization bottleneck is more prominent in the flood model relative to its rule evaluation. Comparing to the baseline CD++, the parallelized FSK accelerates the fire and flood simulations by a factor of up to 70.6 and 83.32.

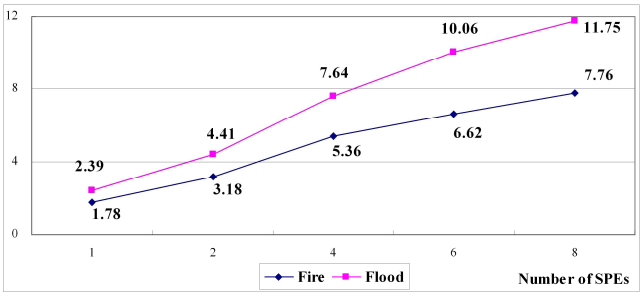

Figure 12. Simulation speedup over optimized CD++

Figure 13 shows the scaling of the FSK itself as exhibited in the simulations. Both synchronization functions attain super-liner speedups due to SIMD code vectorization and hiding memory latency with double buffering. Overall, the parallelized FSK accomplishes the synchronization task up to 14.36 and 18.75 times faster than the optimized sequential version on PPE.

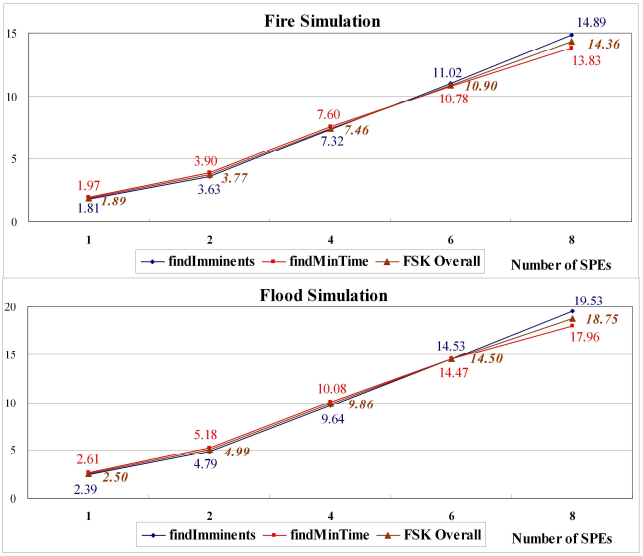

Figure 13. FSK speedup over the PPE-optimized version

8. CONCLUSION AND FUTURE WORK

Multicore technologies offer great opportunities for delivering an unprecedented performance that previously could only be attained on high-end super clusters. They also bring software development challenges that require the integration of complex, multigrain parallelism in a coherent way. This paper presents a new technique for efficient parallel simulation of large-scale DEVS-based models on

the IBM Cell processor. Following a performance-centered approach, we addressed all major performance bottlenecks by exploiting multi-dimensional parallelism. New forms of fine-grained event-level parallelism were analyzed, which is fundamental in the parallelization of DEVS-based simulation systems. Our preliminary experiments have already produced very promising results, proving that optimizing and porting DEVS simulations to multicore platforms like the Cell processor is worth the effort. The proposed technique not only allows a broad community of DEVS users to tap the potential of the Cell processor without being distracted by the complexity of multicore programming, but also provides insights on migration of legacy software to current and future multicore platforms.

We are implementing the SEK in CD++, which will be the first object-oriented full-fledged DEVS simulation engine on the Cell processor. We are also investigating new methods to combine cluster-based parallel simulation with Cell-accelerated simulation on large-scale hybrid systems.

References

[1] Khale, J.A.; M.N. Day; H.P. Hofstee; C.R. Johns; T.R. Maeurer; D. Shippy. 2005. "Introducing to the Cell Multiprocessor". IBM Journal of Research and Development 49(4/5), pp. 589-604.

[2] Williams, S.; J. Shalf; L. Oliker; S. Kamil; P. Husbands; K. Yelick. 2006, "The Potential of the Cell Processor for Scientific Computing". In Proceedings of the 3rd Conference on Computing Frontiers, pp. 9-20. Ischia, Italy.

[3] Crawford, C.H.; P. Henning; M. Kistler; C. Wright. 2008. "Accelerating Computing with the Cell Broadband Engine Processor". In Proceedings of the 5th Conference on Computing Frontiers, pp. 3-12. Ischia, Italy.

[4] Zeigler, B.P.; H. Praehofer; T.G. Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, San Diego, CA.

[5] Chow, A.C.; B.P. Zeigler. 1994. "Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism". In Proceedings of the 26th Winter Simulation Conference, pp. 716-722. Orlando, FL.

[6] Wainer, G.; N. Giambiasi. 2002. "N-dimensional Cell-DEVS Models". Discrete Event Dynamic Systems 12 (2), pp. 135-157.

[7] Wainer, G. 2002. "CD++: A Toolkit to Develop DEVS Models". Software: Practice and Experience 32(13), pp. 1261-1306.

[8] Wainer, G. 2004. "Modeling and Simulation of Complex Systems with Cell-DEVS". In Proceedings of the 36th Winter Simulation Conference (WSC), pp. 49-60. Washington, DC.

[9] Wainer, G. 2006. "Applying Cell-DEVS Methodology for Modeling the Environment". SIMULATION 82(10), pp. 635-660.

[10] Wainer, G. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, Boca Raton, FL.

[11] Fujimoto, R.M. 2000. *Parallel and distributed simulation systems*. John Wiley & Sons, New York, NY.

[12] Stamatakis, A.; M. Ott. 2008. "Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study". In Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics, pp. 424-435. Melbourne, Australia.

[13] Kudlur, M.; S. Mahlke. 2008. "Orchestrating the Execution of Stream Programs on Multicore Platforms". In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 114-124. Tucson, AZ.

[14] Varbanescu, A.L.; H. Sips; K.A. Ross; Q. Liu; L.K. Liu; A. Natsev; J.R. Smith. 2007. "An Effective Strategy for Porting C++ Applications on Cell". In Proceedings of the 2007 International Conference on Parallel Processing, pp. 59-68. Xian, China.

[15] Eichenberger, A.E.; K. O'Brien; P. Wu; T. Chen; P.H. Oden; D.A. Prener; J.C. Shepherd; B. So; Z. Sura; A. Wang; T. Zhang; P. Zhao; M. Gschwind. 2005. "Optimizing Compiler for the Cell Processor". In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, pp. 161-172. St. Louis, MO.

[16] Knight, T.J.; J.Y. Park; M. Ren; M. Houston; M. Erez; K. Fatahalian; A. Aiken; W.J. Dally; P. Hanrahan. 2007. "Compilation for Explicitly Managed Memory Hierarchies". In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 226-236. San Jose, CA.

[17] McCool, M.D. 2006. "Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform". In Proceedings of GSPx Multicore Applications Conference. Santa Clara, CA.

[18] Perez, J.M.; P. Bellens; R.M. Badia; J. Labarta. 2007. "CellSs: Making it Easier to Program the Cell Broadband Engine Processor". IBM Journal of Research and Development 51(5), pp. 593-604.

[19] Williams, S.; J. Shalf; L. Oliker; S. Kamil; P. Husbands; K. Yelick. 2007. "Scientific Computing Kernels on the Cell Processor". International Journal of Parallel Programming 35(3), pp. 263-298.

[20] Agarwal, V.; L.K. Liu; D.A. Bader. 2008. "Financial modeling on the Cell Broadband Engine". In Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing, pp. 1-12. Miami, FL.

[21] Liu, Q.; G. Wainer. 2007. "Parallel Environment for DEVS and Cell-DEVS Models". SIMULATION 83(6), pp.449-471.

[22] ARS Laboratory. 2004. *Flood Model*. http://cell-devs.sce.carleton.ca/ars/?q=node/11