Exploring Multi-Grained Parallelism in Compute-Intensive DEVS Simulations

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

***Abstract*—We propose a computing technique for efficient parallel simulation of compute-intensive DEVS models on the IBM Cell processor, combining multi-grained parallelism and various optimizations to speed up the event execution. Unlike most existing parallelization strategies, our approach explicitly exploits the massive fine-grained event-level parallelism inherent in the simulation process, while most of the logical processes are virtualized, making the achievable parallelism more deterministic and predictable. Together, the parallelization and optimization strategies produced promising experimental results, accelerating the simulation of a 3D environmental model by a factor of up to 33.06. The proposed methods can also be applied to other multicore and shared-memory architectures.**

Keywords-DEVS formalism; Cell-DEVS formalism; multi-grained parallelism; multicore computing; Cell processor

I. Introduction

Chip Multiprocessor (CMP) architectures have been used to address the limitations of microprocessor performance. Among them, the IBM Cell processor [1] adopts a heterogeneous CMP architecture with 9 independent cores: a general-purpose dual-threaded Power Processor Element (PPE) and 8 specialized co-processors called Synergistic Processing Elements (SPEs). The PPE uses a conventional cache hierarchy (32KB L1, 512KB L2) to access system main memory and provides top-level thread control for a parallel application. In contrast, each SPE can only directly access a private on-chip Local Storage (LS) of 256KB, which contains both code and data (including the call stack) of an SPE thread. Data sharing is achieved mainly through software-managed, explicitly-addressed Direct Memory Access (DMA) transfers, which require proper address alignment and transfer size to attain peak performance. The cores can also communicate 32-bit short messages via the interconnect bus channels (e.g., mailboxes and signals). Moreover, the SPEs support both scalar and 128-bit SIMD (Single Instruction, Multiple Data) operations, which can be applied at different granularities. Along with the potential of the Cell processor, however, comes a significant increase in software complexity, requiring innovative redesign of existing algorithms in return for better application performance.

The field of Parallel Discrete Event Simulation (PDES), for instance, could benefit from the various parallelization options offered by the Cell processor. Most of the existing PDES techniques adopt a logical process (LP) oriented approach to partition a simulation across multiple nodes of a parallel computing system [2], making it difficult to exploit other forms of fine-grained parallelism, such as those available on multicore platforms. As multicore computing is becoming pervasive, there is now a need to bridge the gap between PDES algorithms designed for traditional architectures and those for emerging CMP platforms.

Although the Cell processor has received increasing interest from the modeling and simulation (M&S) community, general-purpose PDES on such platform is still uncommon. The use of a generic M&S methodology on Cell could facilitate modeling, experimentation, testing and maintenance of the simulation software. In particular, we are interested in exploring the application of the Discrete Event System Specification (DEVS) formalism [3], which allows for hierarchical and modular construction of reusable discrete-event models. The original DEVS formalism has been extended in various ways, and in our work, we follow the P-DEVS formalism [4], which improves the handling of simultaneous events, and the Cell-DEVS formalism [5], which describes cell spaces as discrete-event systems (where each cell is a basic P-DEVS model component). Both P-DEVS and Cell-DEVS have been implemented in the CD++ toolkit, an open-source object-oriented M&S environment programmed in C++ [6].

Recently, we have shown how to host DEVS-based simulations on the Cell processor with a special focus on parallelizing the synchronization task, which becomes the main performance bottleneck when the model size is large [7-8]. In this paper, we extend those initial results and present the details of a computing technique that combines multi-grained parallelism and various optimizations for efficient event processing. This allows for the execution of compute-intensive DEVS models on the Cell processor, while hiding the complexity of multicore programming from users. The technique is based on explicit exploitation of the inherent event-level parallelism in the simulation process, whereas most of the LPs are virtualized. Furthermore, it enables fine-grained dynamic load balancing between the cores. The proposed methods can also be applied to other CMP and shared-memory architectures. Using an example of a 3D Cell-DEVS model that simulates hydrological dynamics, our experiments demonstrate that the proposed parallelization and optimization strategies can accelerate the simulation by a factor of up to 33.06 over the original CD++ implementation on the PPE.

The rest of the paper is organized as follows: Section II reviews related work, while Section III introduces DEVS simulation in CD++. Section IV covers the event-level parallelism. Section V shows the computational kernel and the workload characteristics. The computing technique is presented in Section VI, and the experimental results are discussed in Section VII. Section VIII concludes the paper.

II. RELATED WORK

Most existing PDES techniques employ a coarse-grained parallelization strategy at the LP level, without paying much attention to other fine-grained parallelism available on CMP platforms. Multi-grained parallelism has been recently explored on multicore processors for scientific and multimedia applications [9-10]. Nevertheless, new parallelization strategies are still needed in order to exploit multi-grained parallelism for PDES systems on the Cell processor.

Different programming models were proposed to improve programmability on the Cell processor [1]. Although these abstract models provide guidelines for developing new computing techniques, significant efforts are still required to implement PDES algorithms on this architecture.

One way to facilitate software development on the Cell processor is to use compiler-assisted vectorization and pipelining [11-12]. Without a full understanding of the application logic, however, this technique is still inadequate on its own for complex PDES systems with highly irregular computation and complex data dependency.

Although several middleware frameworks were developed for the Cell processor [13-14], most 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. These issues limit their applicability to complex object-oriented PDES systems.

Most Cell applications perform intensive numerical computation following the SIMD model to exploit data-level parallelism [15-16]. The Cell processor has also been used in complex M&S applications to offload specific compute-intensive functions to the SPEs [17-18]. However, the application of these methods in general-purpose PDES systems is not straightforward.

III. DEVS SIMULATION IN CD++

The P-DEVS formalism [4] defines a model as a mathematical entity that is composed of a hierarchy of *atomic* (behavioral) and *coupled* (structural) components. DEVS theory separates the model from the underlying simulation engine, which can be implemented as LPs specialized into *Simulators* and *Coordinators*. A *Simulator* is paired with an *atomic* model to trigger the model's state transition functions, while a *Coordinator* is attached to a *coupled* model to schedule events in the model hierarchy and route data between the model components. The Cell-DEVS formalism [5] defines n-dimensional cell spaces as discrete-event models, in which complex behavior at the global level emerges from a set of local rules. Each cell is defined as a P-DEVS atomic model, allowing for asynchronous execution. Both P-DEVS and Cell-DEVS are implemented in CD++, which provides a built-in specification language to specify Cell-DEVS model behavior using a set of local transition functions [6]. Each function consists of several state transition rules that are represented as syntax trees in the main memory. This specification language is valuable on the Cell processor since general users can focus on their modeling issues without being distracted by the technical details of multicore programming.

CD++ has been extended to support parallel simulation on distributed-memory clusters using a flat LP structure [19], as shown in Figure 1. The sequential simulation on each node is controlled by a *Node Coordinator* (NC), a *Flat Coordinator* (FC), and a group of *Simulators*. The NC is the endpoint of inter-node MPI (Message Passing Interface) messaging, while the FC synchronizes the child Simulators underneath.

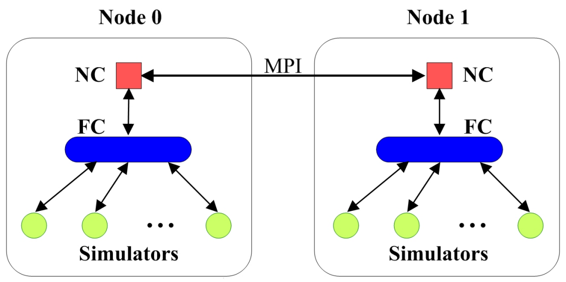


Figure 1. Flat LP structure

The LPs exchange messages that fall into two groups: ***content messages*** include the *external* (X) and *output* (Y) events that represent the actual model input and output data, while ***control messages*** include the *initialization* (I), *collect* (@), *internal* (*), and *done* (D) events that control the simulation execution. As depicted in Figure 2, the sequential simulation on a node can be viewed as being executed in a well-structured manner [19]. At any virtual time, the simultaneous events exchanged between the LPs are organized into 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 determines the next virtual time and advances the simulation on the node.

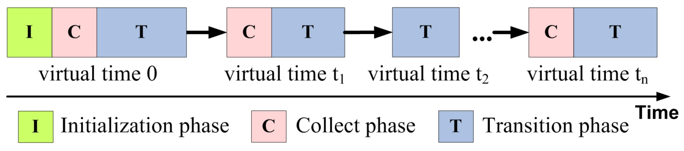


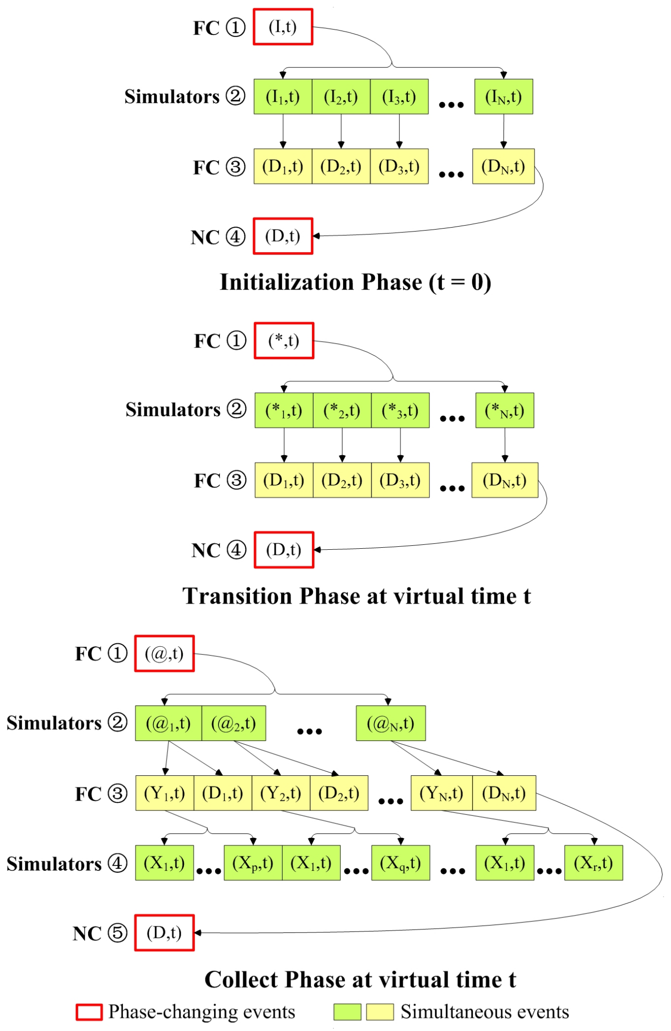
Figure 2. Structured simulation process

In large-scale, densely-interconnected, and highly-active models, DEVS-based simulations have a relatively large number of simultaneous events to be executed at each virtual time. These events include not only the causally-independent content messages for exchange of model data, but also the causally-dependent control messages that enforce a partially-ordered event execution in line with the P-DEVS formalism.

In the following sections, we show how to parallelize the sequential simulation on the Cell processor so as to combine parallel simulation at the cluster level with accelerated parallel simulation on each multicore node.

IV. Event Level Parallelism

A key challenge in the parallelization effort is to expose the event-level parallelism inherent in the DEVS simulation process from a ***data-flow*** perspective. Figure 3 shows a step-by-step view of event execution in the simulation phases based on the flat LP structure.



FC ① (I,t)
Simulators ② (I_1,t) (I_2,t) (I_3,t) … (I_N,t)
FC ③ (D_1,t) (D_2,t) (D_3,t) … (D_N,t)
NC ④ (D,t)
Initialization Phase (t = 0)
FC ① (*,t)
Simulators ② $(*_1,t)$ $(*_2,t)$ $(*_3,t)$ … $(*_N,t)$
FC ③ (D_1,t) (D_2,t) (D_3,t) … (D_N,t)
NC ④ (D,t)
Transition Phase at virtual time t
FC ① (@,t)
Simulators ② $(@_1,t)$ $(@_2,t)$ … $(@_N,t)$
FC ③ (Y_1,t) (D_1,t) (Y_2,t) (D_2,t) … (Y_N,t) (D_N,t)
Simulators ④ (X_1,t) … (X_p,t) (X_1,t) … (X_q,t) … (X_1,t) … (X_r,t)
NC ⑤ (D,t)
Collect Phase at virtual time t
Phase-changing events
Simultaneous events

Figure 3. Event-level parallelism [8]

Borrowing terminology from the parallel computing community, the fine-grained event-level parallelism can be classified into two categories, which are referred to as *event-embarrassing parallelism* and *event-streaming parallelism* in [7]. These two types of event parallelism can be exploited following a conservative approach that avoids causality errors at any virtual time, as follows.

- **Event-Embarrassing Parallelism** exists between the *independent* events executed *within* each step at the FC and the Simulators (shaded). Since there is neither causal nor data dependency between them, these events can be executed concurrently in an arbitrary order.

- **Event-Streaming Parallelism** exists between the *causally-dependent* events executed in *consecutive* steps. As the output events from the preceding step serve as the inputs to the step that follows, the events can be executed concurrently in a pipelined manner.

At the first and last steps of each phase, the NC and FC exchange *phase-changing events*, providing the natural *fork* and *join* points for simulation synchronization. Thanks to the flat LP structure, the number of synchronization points is minimized, exposing the maximum degree of event-level parallelism in the simulation process. Note that, according to the P-DEVS formalism, the simultaneous (X) events received by a Simulator in the collect phase (in step 4) must be consumed as a whole by the Simulator in the ensuing transition phase (in step 2) [4].

V. Event Execution Kernel

In the simulation of DEVS models with complex behavior, the performance is primarily dominated by the intensive computation required to execute the state transition functions at the Simulators (evaluation of transition rules in the case of Cell-DEVS models). To illustrate this performance bottleneck, the original CD++ has been ported to the PPE core using the flat LP structure, resulting in a PPE-based sequential version. The watershed model, previously discussed in [20] and later redefined as a Cell-DEVS model in [21], was used as an example for performance evaluation. This model simulates environmental influence on the hydrological dynamics of water accumulation using a 3D cell space (320×320×2). Table 1 gives an execution profile of the watershed simulation with the PPE-based CD++ on an IBM BladeCenter QS22 server.

TABLE I. Watershed Execution Profile on PPE

Message Type	Components				
	Simulator	*FC*	*NC*	*Boot-strap*	*Other overhead*
(I)	0.65	0.15	–	–	–
(*)	78082.60	75.27	–		
(@)	95.09	74.32	–		
(X)	122.58	0	–		
(Y)	–	905.40	–		
(D)	–	14.26	0.002		
Sum (s)	**78300.92**	**1069.40**	**0.002**	**25.22**	**488.12**
Total (s)	**79883.66**				

It is clear that the dominant bottleneck resides at the Simulators (shaded), called as the *Simulator Event-processing Kernel* (SEK) thereafter, consuming more than 98% of the total execution time. Further, the Simulators spend most of the time on processing (*) events where the transition rules are evaluated. The SEK includes the Simulator algorithms for processing (I), (*), and (@) events as well as the DEVS state transition functions defined in the atomic models. Note that the SEK does not include the Simulator algorithm for processing (X) events, as will be explained in Section VI-D. Definition of these event-processing algorithms can be found in [19] and [22]. During each simulation phase, the SEK executes the input events received from the FC based on the current states of the Simulators (and their associated atomic models), and returns a set of output events to the FC. The input/output events and states are handled independently between individual Simulators.

TABLE II. EVENT COUNTS IN WATERSHED SIMULATION

Message Type	LPs		
	Simulator	*FC*	*NC*
(I)	204,800	1	–
(*)	33,996,800	331	–
(@)	33,527,325	331	–
(X)	167,723,421	0	–
(Y)	–	33,527,325	–
(D)	–	67,728,925	663
Sum	**235,452,346**	**101,256,913**	**663**
Total Events	**339,221,007**		

Table 2 gives the event counts in the watershed simulation, which includes 663 simulation phases. As we can see, the watershed model executes a high proportion of simultaneous events. Only 1326 out of the over 300 million events are the phase-changing events (shaded), while all the others are the simultaneous events executed at distinct virtual times. Another property of the workload is reflected in the relatively low event rate at the Simulators due to the intensive state transitions. These two properties are representative in DEVS simulation of densely coupled models with complex behavior, which justifies our effort to accelerate the SEK on the Cell processor.

VI. PARALLEL DEVS SIMULATION ON CELL

A. Architecture Overview

Figure 4 shows an overview of the computing technique. During simulation bootstrap, the PPE main thread spawns a helper thread, which in turn creates a group of SPE threads (one on each SPE). The NC and FC are hosted respectively by the two PPE threads, which share the Future Event List (FEL) to execute the phase-changing events in a producer-consumer fashion. At each virtual time, the SPE threads are orchestrated to process events for the Simulators using a set of pending job queues. The Simulators' data (i.e., events, states, and transition rules) are managed in various buffers allocated in the main memory. These data are fetched and stored across memory domains with SPE-initiated double-buffered DMA transfers. When the simulation starts, the addresses of the data buffers are passed to the SPE threads in a control block.

Multi-grained parallelism is achieved as follows. The SEK algorithms are implemented in C using explicit SPE SIMD intrinsics to exploit vector parallelism. Due to the irregular nature of the computation, only partial vectorization is applied to parallelize the most time-consuming loops in the SEK code. Thread-level parallelism is explored across the SPEs, where each SPE thread hosts an instance of the SEK. At any virtual time, the independent events targeting different Simulators are executed concurrently at distinct SEKs, realizing event-embarrassing parallelism. In addition, event-streaming parallelism is utilized by executing the causally-dependent events passed between the Simulators (running on the SPEs) and the FC (running on the PPE) in a two-stage pipeline. Besides, double-buffered DMA is used at three layers to transfer the pending job IDs, event and state data of individual jobs, and rule data required in the execution of (*) events, hiding memory latency with data-streaming parallelism. Throughout the simulation, the PPE main thread handles file I/O and inter-node messaging in parallel with the helper thread to overlap communication and computation.

Note that peak DMA performance is achievable when the addresses of the data in both memory domains are cache-line (128-byte) aligned and when the size of transfer is 512 bytes or larger [23], which underlies the rationale behind the simulation data management schemes that will be discussed next.

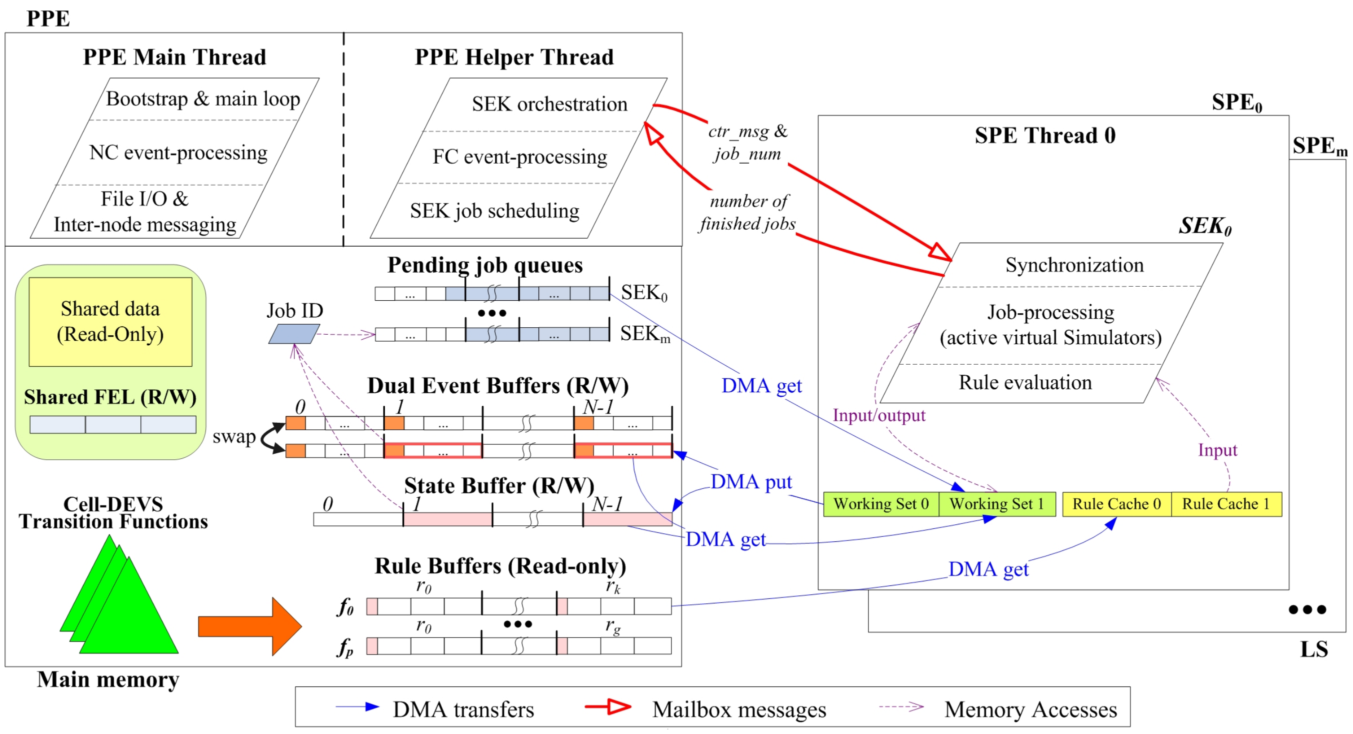


Figure 4. Architectural overview of parallel DEVS simulation on the Cell processor

B. LP Virtualization

Several issues must be addressed when mapping the Simulators to the SPE threads. First, the small size of the LS imposes a tight limit on the number of Simulators that can be hosted simultaneously on an SPE. Secondly, an SPE can only execute one thread at a time; and SPE context switch is considered as a very expensive operation. For this reason, the SPEs are usually assigned with reusable tasks operating on a stream of data. Thirdly, in a typical simulation, only a fraction of the Simulators are active at any virtual time. Hence, the partition scheme should only map the active Simulators to the SPE threads. Finally, the partition scheme also needs to facilitate dynamic load balancing between the SPEs.

We solve these issues through the concept of LP virtualization, by which the Simulators (and their associated atomic models) are turned into *virtual LPs* that share the functionalities provided by a limited group of SPE threads, and the mapping of active Simulators to the SPEs is determined dynamically at each virtual time throughout the simulation. To this end, the state data originally encapsulated in the Simulator-atomic pairs of objects are separated from the event-processing logic. While the state data are stored in the main memory, the event-processing logic is wrapped into the SEK and hosted on the SPEs. At runtime, the state of an active Simulator is matched to a specific SPE thread using an SEK job-scheduling algorithm, as will be discussed in Section VI-F. On the contrary, the NC and the FC remain the two concrete LPs running on the PPE core.

C. State Management

For efficient DMA transfer of Simulator state data, a new process ID allocation scheme is used to allocate the Simulator IDs continuously from 0 to (N-1) where N is the total number of Simulators created in the system. One the other hand, the NC and the FC are assigned with negative IDs. In this way, the states of all of the Simulators can be stored in a flat 128-byte aligned array (***state buffer***) where the array indexes serve as the Simulator IDs, as shown in Figure 5.

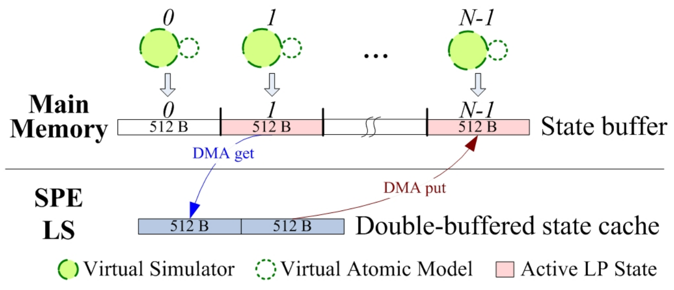


Figure 5. Virtual Simulator state management

Each state buffer entry has an adjustable size of 512 bytes to contain the state data extracted from the corresponding Simulator-atomic pair of objects. On each SPE, a 128-byte aligned local state cache is used to hold the states of at most two active Simulators at any time, allowing for pre-fetching the state of the next active Simulator while processing events for the current one.

D. Decentralized Event Management

To transfer events across memory domains, the raw data included in each type of CD++ event objects are encoded in 32 bytes; and a pair of flat 128-byte aligned arrays (***event buffers***) is allocated in the main memory to exchange the simultaneous events passed between the FC and the virtual Simulators. Each event buffer entry has an adjustable size of 1KB to hold up to 32 events at a time for a dedicated Simulator. At any step of a simulation phase, the FC and a Simulator may exchange at most one control message and optionally a list of content messages. Hence, the first slot in each event buffer entry is reserved for passing the control events, while the following slots are used to pass the content events, if any. The original FEL is only used to send phase-changing events between the FC and the NC. Together, the FEL and the event buffer entries form a network of bidirectional communication channels with a star topology centered at the FC.

During a collect phase, the FC translates (Y) events received from the source Simulators into (X) events that will be sent to the destination Simulators. Instead of putting these (X) events into the entries of the *current event buffer*, the FC writes them into the corresponding entries of the *backup event buffer*, as illustrated in Figure 6. Hence, using a pair of event buffers allows the FC to process (Y) events on the PPE concurrently with Simulator event execution on the SPEs without additional synchronization. After writing the (*) events into the backup event buffer entries at the beginning of the ensuing internal phase, the FC resets an integer flag, called as *event-buffer-index* (0 or 1), to swap the two buffers. On the other hand, the virtual Simulators always work on the current event buffer as determined by the FC.

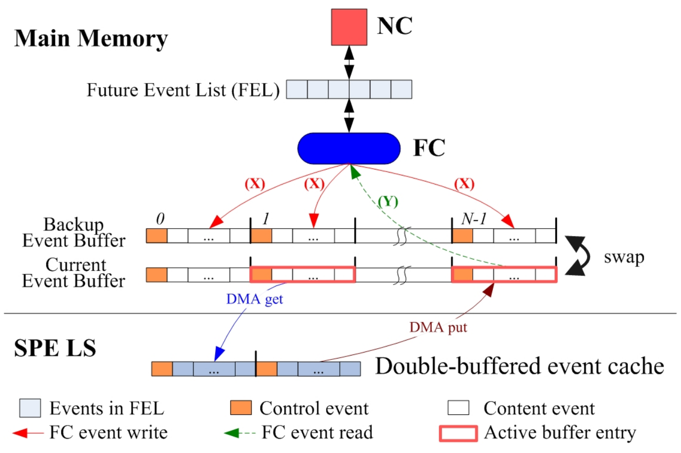


Figure 6. Virtual Simulator event management

This decentralized event management scheme has several major advantages. First, multiple input/output events of a Simulator are stored in the same event buffer entry, which can be transferred efficiently with one DMA operation. Secondly, all the simultaneous events are removed from the FEL, with lowered event queue operation cost. Thirdly, the simultaneous events are read/written directly in the event buffers without memory allocation and deallocation, further reducing the operational overhead. Fourthly, the Simulators no longer need to process (X) events in the collect phases, minimizing the required DMA operations for transferring event data and simplifying the SEK algorithms. As the (X) and (*) events targeting a Simulator are packed together in the same event buffer entry, they can be consumed as a whole in the transition phases. Finally, the on-chip PPE cache hierarchy is better utilized because of enhanced data locality.

E. Rule Evaluation on SPE

The evaluation of local transition functions is the core computation of Cell-DEVS atomic models. Originally, the rules included in a transition function are represented as syntax trees, and the evaluation is performed recursively [6]. However, recursion on the SPEs is problematic due to the very limited size of the call stack and the lack of stack overflow protection on these co-processors. Besides, the syntax trees are not well-suited for efficient DMA transfer. To solve these problems, the rules of each transition function are converted to a sequence of floating-point values organized in postfix format and packed in a flat 128-byte aligned array (***rule buffer***), as shown in Figure 7. Each packed rule has four components, including three sub-trees that define the value (*v*), delay (*d*), and precondition (*c*) of the rule respectively, and a header that indicates the total numbers of syntax nodes included in the sub-trees. A syntax node is encoded with two values: an integer operation type and an optional floating-point operand value.

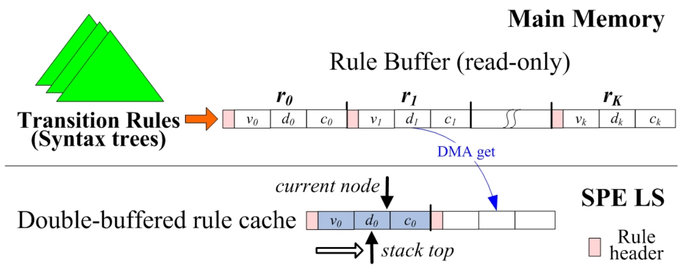

Main Memory
Rule Buffer (read-only)
Transition Rules (Syntax trees)
r_0
r_1
r_K
v_0 d_0 c_0 v_1 d_1 c_1 v_k d_k c_k
DMA get
current node
SPE LS
Double-buffered rule cache
stack top
Rule header

Figure 7. Packing of a local transition function

Double-buffered DMA is used to fetch the rules for evaluation on the SPEs. The recursive algorithm is replaced by an iterative one that scans a rule one syntax node at a time. The LS rule cache itself is used as a software-managed stack to hold the intermediate operands, allowing the rules to be evaluated in place without burdening the SPE call stack.

F. SEK Job Scheduling

An SEK job executes the events scheduled for a given active Simulator in a specific simulation phase. The input events and state data are fetched from the corresponding entries in the current event buffer and state buffer. After event execution, the generated output events and updated state are transferred back to the original buffer entries. Depending on the type of the current simulation phase, the SEK jobs are handled in one of three SEK functions (each has a unique *SEK function ID*), which implement the Simulator algorithms for processing (I), (@), and (*) events respectively. The IDs of the active Simulators are used as the SEK job IDs, which are scheduled by the FC using the pending job queues. Each job queue is a 128-byte aligned integer array created for an SEK to contain the pending job IDs, as depicted in Figure 8. The job IDs are transferred to the SEK in chunks using double-buffered DMA. Each chunk has an adjustable size of 32 job IDs (128 byte in total). The SEK handles the job IDs available in the local job cache sequentially. These job IDs are used as offsets to access data in the event and state buffer entries in the main memory.

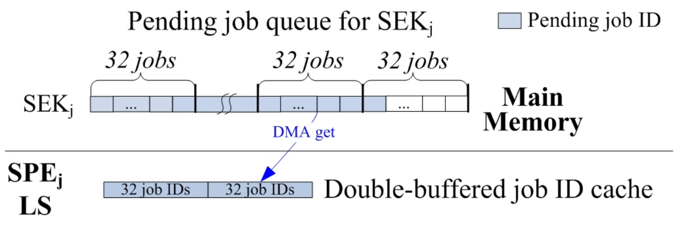


Figure 8. Pending job queue for an SPE thread

At the beginning of a simulation phase, the FC first executes the phase-changing events in the FEL, and then directly writes the generated events into the event buffer entries based on the IDs of the receiving Simulators. These Simulator IDs are inserted into the pending job queues under a certain job-scheduling policy, thus mapping the active Simulators to the SEKs. Since the SEK jobs executed in a simulation phase are of the same type with similar computational intensity, simple yet effective scheduling policies (e.g., round-robin or shortest-queue-first) suffice to achieve fine-grained dynamic load-balancing between the SPE threads.

G. SEK Orchestration

As shown in Figure 9, an SEK is invoked with two mailbox messages sent from the FC on PPE (line 4 and 5), including a control message (*ctr_msg*) that encodes the current event-buffer-index and the intended SEK function ID; and an integer (*job_num*) that indicates the total number of pending jobs scheduled for the SEK. During job execution, the SEK sends mailbox messages to the PPE periodically at a user-defined rate (default *FREQ* is 8), notifying the FC the number of jobs that have been finished so far in the pending job queue (line 10).

```
1. when the SPE thread is created
2.  Fetch the starting addresses of the event/state/rule/job buffers via a control block
3.  while termination signal is not received do
4.      Wait for an SEK control message (ctr_msg) from the inbound mailbox
5.      Wait for the number of pending jobs (job_num) from the inbound mailbox
6.      Double-buffered DMA - Fetch the current & next chunks of pending job IDs
7.      for each pending job in the current chunk do
8.          Double-buffered DMA - Fetch event/state data for the current & next jobs
9.          Call the corresponding SEK function to process the current job
            (if the current simulation phase is internal, the SEK function uses double-
             buffered DMA to fetch the state transition rules for evaluation)
10.         Send a mailbox message to the PPE after processing every FREQ jobs
11.     end for
12.  end while
13.end when
```

Figure 9. A skeleton of the SEK algorithm

On the PPE side, the FC updates the status of the job queue upon the arrival of each SEK mailbox message (line 13 in

Figure 10). While waiting for the SEKs, the FC processes the output events from the virtual Simulators in parallel (line 16). When all of the pending jobs are finished by the SEKs, the FC executes any remaining output events (line 18), and sends phase-changing events to the NC through the FEL, ending the current simulation phase (line 19).

```
1. when a simulation phase starts
2.   FC executes phase-changing events in the FEL (received from the NC)
3.   FC writes generated events for Simulators in corresponding event buffer entries
4.   FC inserts the pending Job IDs into the job queues (SEK job scheduling)
5.   for each SEK with pending jobs do
6.      Send the SEK a control message (ctr_msg) via mailbox channel
7.      Send the SEK the number of pending jobs (job_num) via mailbox channel
8.   end for
9.   while not all of the pending jobs are processed by the SEKs do
10.     for each SEK with pending jobs do
11.        Non-blocking poll the outbound mailbox channel from the SEK
12.        if a mailbox message is received from the SEK then
13.           Update the status of the job queue (FREQ jobs are finished by the SEK)
14.        end if
15.     end for
16.     FC processes output events from an SEK based on the finished Job IDs
17.  end while
18.  FC processes any remaining output events from the SEKs
19.  FC sends phase-changing events to the NC via the FEL
20. end when
```

Figure 10. SEK orchestration algorithm on PPE

VII. Experimental Results

The PPE-based CD++ has been extended using the proposed computing technique. The parallel simulator, called as CD++/Cell, was implemented on Red Hat Enterprise Linux 5.2 with the IBM SDK for Multicore Acceleration 3.1. This section analyzes the experimental results obtained in the watershed simulation introduced in Section V. The experiments were conducted on an IBM BladeCenter QS22 server with 3.2 GHz PowerXCell 8i processors and 32GB main memory. SEK jobs were scheduled using the round-robin policy. And the CD++ event logging was disabled to minimize the impact of file I/O operations on system performance. The results of each test case were averaged over 10 independent simulation runs to enhance data reliability.

Figure 11 gives the total execution time obtained by the PPE-based CD++ and by the CD++/Cell on 1 to 8 SPEs. With just one SPE, the simulation runs 5.84 times faster than the PPE-only implementation. There are several reasons for this exceptional performance gain. The SEK is implemented in SIMD-aware C code on SPE, making it more efficient than the object-oriented scalar C++ version on PPE. Moreover, memory latency, a performance constraint in the PPE-based simulation due to the random data access pattern and the resulting high cache miss rates in the hardware-controlled PPE cache hierarchy, is effectively minimized with the software-managed multi-layered double buffering strategy for DMA data transfer to/from the SPE. In addition, various optimizations are applied to further streamline the kernel computation, including improving SIMDization by proper LS data alignment, reducing SPE call stack usage by in-place rule evaluation and by replacing function parameters and local variables with aligned global variables and registers, removing branches whenever possible or using branch hints explicitly, and enhancing performance by loop unrolling and in-line substitution. Finally, using the PPE along with an SPE allows for pipelined event execution between the FC and the virtual Simulators, taking advantage of the event-streaming parallelism. Overall, the total executing time is reduced from more than 22 hours on the PPE to just 40 minutes with 8 SPEs (or an improvement by a factor of 33.06).

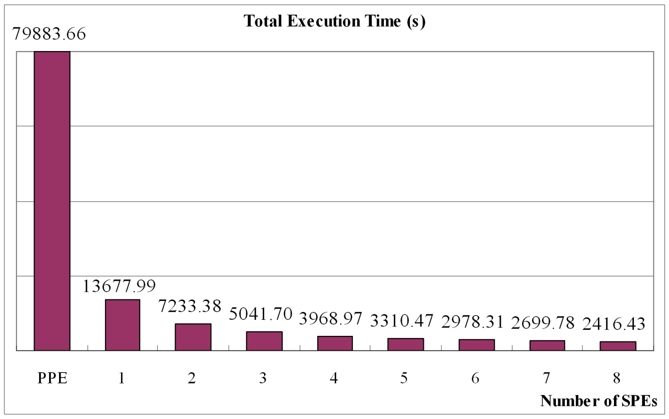


Figure 11. SEK impact on total execution time in the watershed simulation

Using the simulation on the PPE with one SPE as the baseline case, Figure 12 shows the overall simulation speedups. These speedups are conservative estimates because the baseline case, instead of a purely sequential execution, already exploits SIMD and event-streaming parallelism. Even so, CD++/Cell still achieved a significant speedup of 5.66 on 8 SPEs. The speedup grows a bit slower when more and more SPEs are used for two reasons. First, an increasing number of SPEs results in higher overhead for SEK job scheduling and orchestration. Secondly, when all the SEKs transfer data at the same time, the performance suffers from increased DMA contention and channel stalls.

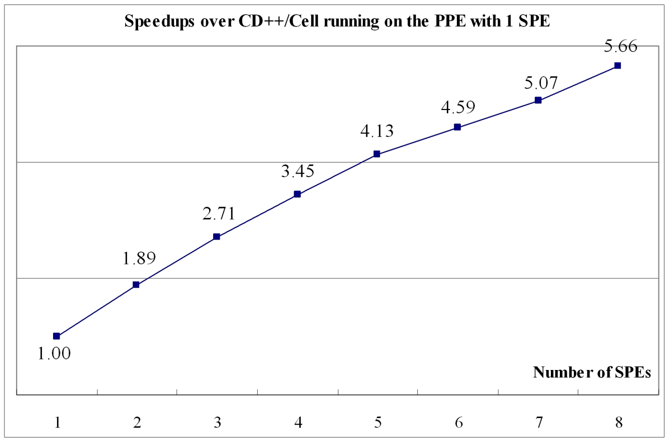


Figure 12. Scaling of SEK with number of SPEs

VIII. Conclusion and Future Work

Multicore computing poses new challenges that demand the exploitation of parallelism at different levels in real-world M&S applications. To address some of these challenges, this paper presents a novel computing technique for efficient parallel simulation of compute-intensive DEVS models on the heterogeneous IBM Cell processor. Following the general-purpose DEVS methodology, the technique combines multi-grained parallelism and various performance optimizations in a coherent way to accelerate the event execution, while hiding the complexity of multicore programming from general users. In contrast to most existing LP-oriented parallelization strategies, our approach explicitly and directly explores the massive fine-grained event-level parallelism that is inherent in DEVS-based systems, whereas most of the LPs are virtualized, making the achievable parallelism more deterministic and predictable. Using a 3D watershed model, as an example, our experiments have already produced very promising results, reducing the total execution time by a factor of up to 33.06 on a Cell processor over the original PPE-based implementation. Moreover, the methods presented in this paper, especially the concept of LP virtualization and data-flow oriented exploitation of event-level parallelism, can be readily applied to other CMP architectures and shared-memory multiprocessors. This paper also illustrates several practical considerations that are of interest to other application developers who intend to port legacy software to the Cell processor.

We are currently integrating the computing technique in large-scale parallel simulation on hybrid super clusters, combining the advantages of cluster-based parallel simulation with the benefits of multicore-accelerated parallel simulation.

Acknowledgment

This research 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. Michael Perrone, Dr. Ligang Lu, Dr. Daniele Paolo Scarpazza and Dr. Lurng-Kuo Liu from IBM Watson Research Center for their insightful feedback.

References

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

[2] R. M. Fujimoto, Parallel and Distributed Simulation Systems, New York: John Wiley & Sons, 2000.

[3] B. P. Zeigler, H. Praehofer, and T. G. Kim, Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Ed., London: Academic Press, 2000.

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

[5] G. Wainer and N. Giambiasi, "N-dimensional Cell-DEVS models". Discrete Event Dynamic Systems, vol. 12, no. 2, pp. 135-157, 2002.

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

[7] Q. Liu, G. Wainer, L. Lu, and M. Perrone, "Novel performance optimization of large-scale discrete-event simulation on the Cell Broadband Engine". Proceedings of HPCS 2010, Caen, France, 2010, in press.

[8] Q. Liu and G. Wainer, "Accelerating large-scale DEVS-based simulation on the Cell processor". Proceedings of DEVS 2010, SpringSim'10, Orlando, FL, 2010, in press.

[9] F. Blagojevic, X. Feng, K. W. Cameron, and D. S. Nikolopoulos, "Modeling multigrain parallelism on heterogeneous multi-core processors: A case study of the Cell BE". Proceedings of HiPEAC, LNCS 4917, Goteborg, Sweden, pp. 38-52, 2008.

[10] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms". Proceedings of ACM SIGPLAN PLDI, Tucson, AZ, pp. 114-124, 2008.

[11] A. E. Eichenberger, et al. "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture". IBM Systems Journal, vol. 45, no. 1, pp. 59-84, 2006.

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

[13] 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.

[14] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making it easier to program the Cell Broadband Engine processor". IBM Journal of Research and Development, vol. 51, no. 5, pp. 593-604, 2007.

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

[16] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, N. Kistler, and M. Perrone, "Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine". Proceedings of IPDPS, Long Beach, CA, pp. 62-71, 2007.

[17] V. Agarwal, L. K. Liu, and D. A. Bader, "Financial modeling on the Cell Broadband Engine". Proceedings of IPDPS, Miami, FL, pp. 1-12, 2008.

[18] G. De Fabritiis, "Performance of the Cell processor for biomolecular simulations". Computer Physics Communications, vol. 176, no. 11-12, pp. 660-664, 2007.

[19] Q. Liu and G. Wainer, "Parallel environment for DEVS and Cell-DEVS models". SIMULATION, vol. 83, no. 6, pp.449-471, 2007.

[20] B. P. Zeigler, Y. Moon, D. Kim, and G. Ball, "The DEVS environment for high-performance modeling and simulation". IEEE Computational Science & Engineering, vol. 4, no. 3, pp. 61-71, 1997.

[21] G. Wainer, "Applying Cell-DEVS methodology for modeling the environment". SIMULATION, vol. 82, no. 10, pp.635-660, 2006.

[22] A. Troccoli and G. Wainer, "Implementing parallel Cell-DEVS". Proceedings of ANSS, Orlando, FL, pp. 273-280, 2003.

[23] M. Araya-Polo, F. Rubio, R. Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors". Scientific Programming, vol. 17, no. 1-2, pp. 185-198, 2009.