Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
88(7) 801–831

DOI: 10.1177/0037549711412237
sim.sagepub.com

Multicore acceleration of Discrete Event System Specification systems

Qi Liu[1] and Gabriel Wainer[2]

Abstract

Parallel discrete-event simulation on heterogeneous multicore platforms requires innovative redesign of existing algorithms in return for better performance. Based on the Discrete Event System Specification (DEVS) methodology, a technique called Multicore Acceleration of DEVS Systems is proposed for efficient parallel discrete-event simulation on the IBM Cell processor. The technique combines multi-grained parallelism and various optimizations to overcome performance bottlenecks, while hiding the technical details of multicore programming from non-expert users. By explicitly exploiting the data- and event-level parallelism inherent in the simulation, the technique significantly accelerates both memory-bound and compute-bound computational kernels in demanding parallel DEVS simulations, as shown in the experimental results. Several key concepts and methods derived from this research can also be applied to other multicore and shared-memory architectures.

Keywords

cell processor, discrete-event simulation, Discrete Event System Specification formalism, multicore computing

1. Introduction

As the monolithic approach to microprocessor design reaches a point of diminishing return, the industry has moved towards multicore chip-multiprocessor (CMP) architectures. Previous studies suggest that *heterogeneous* CMP designs, in which different types of cores of varying size and complexity are integrated on a single die, have the potential to meet the needs of a broad spectrum of applications.[1–3] One such example is the IBM Cell processor,[4,5] which includes two types of cores based on different instruction sets and memory subsystems. While the architectural features of the Cell processor are attractive, its asymmetric design of heterogeneous cores with explicit memory control requires redesign of existing algorithms for optimized performance.

Although the Cell processor has gained popularity in scientific and multimedia applications,[6,7] its potential has yet to be realized in parallel discrete-event simulation (PDES). Existing PDES techniques usually use logical processes (LPs) to parallelize a simulation on multiprocessor systems,[8] whereas parallel simulation on the Cell (and on CMP architectures in general) is more efficient when all of the parallelization options are exploited *simultaneously* at different system levels. Moreover, PDES programs typically involve irregular, control-intensive computation with complex data dependency and unpredictable memory access patterns, a class of workload not well suited for the Cell.[9] Recent advances in compilation and middleware technologies (e.g. Eichenberger et al.;[10] Knight et al.;[11] McCool;[12] Perez et al.[13]) offer little help in parallelizing PDES systems on the Cell, due to the lack of adequate knowledge for exploiting application-level parallelism effectively. Several programming models and strategies attempt to provide guidance for porting applications to the Cell.[4,14,15] However, developers still need to handle such issues as computational kernel analysis, data layout and movement, and task synchronization explicitly.

With the advent of these CMP architectures and their use in large parallel computers (such as the

[1]IBM T. J. Watson Research Center, USA.
[2]Department of Systems and Computer Engineering, Carleton University, Canada.

Corresponding author:
Qi Liu, Exascale Computing, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA
Email: liuqi@us.ibm.com

Roadrunner supercomputer[16]), there is a growing interest in extending PDES techniques to such platforms.[17,18] Nevertheless, efficient PDES on these platforms is a complex and error-prone task. One way to facilitate this task is through the use of sound modeling and simulation (M&S) methodologies. An advanced modeling environment can assist in modeling, experimentation, testing, and software maintenance in large-scale simulations by allowing the user to focus on the problems to solve, reducing the effort on technical details. Among them, the Discrete Event System Specification (DEVS)[19] provides a general framework for hierarchical and modular construction of reusable discrete-event models. Parallel DEVS (or P-DEVS)[20] improves the mechanism for handling simultaneous events in DEVS simulations; Cell-DEVS[21] allows for the construction of discrete-event cellular models using P-DEVS components. Both P-DEVS and Cell-DEVS are implemented in CD++,[22] an optimized discrete-event simulator that has been redesigned to support PDES on the Cell processor.[23–25]

This paper presents a technique called Multicore Acceleration of DEVS Systems (MADS) for efficient DEVS-based parallel simulation on the Cell. The MADS technique adopts a *data-flow oriented* strategy to exploit the inherent data and event parallelism, while combining multi-grained parallelism and various optimizations to accelerate both memory-bound and compute-bound computational kernels. In addition, the technique hides the technical details of multicore programming from general users, allowing them to benefit from increased computing capacity with minimal knowledge of the execution environment. Several key concepts and methods proposed here can be applied to other CMP and shared-memory platforms, bridging the gap between PDES algorithms and emerging CMP architectures.

In the following, Section 2 reviews related work. Section 3 introduces the DEVS methodology and its realization in CD++. Section 4 presents the workload characteristics of demanding DEVS-based simulations, along with preliminary optimizations. The MADS technique is proposed in Section 5, while the performance results are discussed in Section 6. Section 7 concludes the paper with suggestions on future research directions. A glossary of the acronyms used in this paper is given in the Appendix.

2. Background and related work

2.1. The Cell architecture

The Cell processor employs a heterogeneous architecture with nine independent cores: a main two-way hardware multithreading Power Processor Element (PPE) and eight co-processors called Synergistic Processing Elements (SPEs).[5] The PPE is intended to execute control-intensive code using a conventional cache hierarchy (32 kB L1, 512 kB L2), while each SPE is optimized to execute compute-intensive code using a private on-chip Local Storage (LS) of 256 kB. Data sharing relies on software-managed DMA (direct memory access), which requires proper address alignment in both memory domains (i.e. main memory and LS) and transfer size to attain peak performance.[26,27] The cores can also communicate 32-bit messages through the Element Interconnect Bus (EIB) channels, namely mailboxes and signals. Moreover, the SPEs support both scalar and 128-bit SIMD (Single Instruction, Multiple Data) operations at different granularities. These architectural features result in an increase in software complexity, requiring an application to be partitioned in a way that not only fits the functional specialization of the cores, but also meets the requirements of DMA transfer to reduce memory latency.

2.2. Related work

As noted by Gschwind,[28,29] the performance of a Cell application relies on *simultaneous* exploitation of multi-grained parallelism at different system levels. While this has been explored in scientific applications,[30,31] doing so in PDES remains a challenge. A PDES system typically has computational kernels with varied data access and workload characteristics, requiring different parallelization strategies. Moreover, most existing PDES techniques realize coarse-grained parallelization at the LP level, making it difficult to parallelize kernels that are not naturally aligned with LP boundaries.

Various programming models have been used on CMP architectures. In Stamatakis and Ott,[32] a bioinformatics application was studied using the MPI (Message Passing Interface), Pthreads, and OpenMP. The authors suggest that the selection of the programming model should rest on software engineering criteria and promote data locality. *Stream programming* is a known programming model that organizes parallel computation from a data-flow perspective.[33,34] In particular, six programming models were proposed to improve programmability on the Cell.[4] Although they provide guidelines for developing new computing techniques, significant efforts are still required to implement PDES algorithms.

Advanced compilation techniques can facilitate software development. For example, the IBM XL C/C++ Compiler for Multicore Acceleration[35] includes automated branch prediction, instruction prefetching, and SPE code vectorization on the Cell.[10] An optimizing compiler was developed for processors with software-managed memory hierarchies,[11] addressing such issues

as data padding, software pipelining, and memory space allocation. Without a full understanding of the application logic, these techniques are inadequate for PDES systems with irregular computation and complex data dependency. On the other hand, compiler-assisted optimization can help improve PDES performance when application-level parallelism has already been properly extracted.

Efforts towards providing a layer of abstraction on top of the Cell programming primitives have resulted in the development of several middleware frameworks. Among them, RapidMind[12] uses an embedded programming language inside C++ to construct a computation using the data-parallel programming model. CellSs[13] exploits functional parallelism based on user-supplied source code annotations. MPI Microtask[36] enables certain MPI applications on the Cell, assuming that the application can be partitioned manually to fit into the LS. Although these frameworks have shown noticeable success, some of them assume a strict data-parallel programming model or adhere to a pure C programming language, while others provide only a minimal set of functionality of a standard library for specific applications. These limitations hinder their applicability to complex object-oriented PDES systems.

The Cell has been used in different applications, such as scientific kernels,[37] Fourier transformation,[38] regular expression scanning,[39] and image processing.[40] Most of them exploit task and/or data perallelism based on a priori knowledge of the workload. The Cell has also been used to host M&S applications, including molecular dynamics,[41] lattice Boltzmann,[42] and financial simulations.[43] Instead of aiming at PDES, they focus on porting numerically intensive computational kernels (e.g. random number generation) to the SPEs.

Novel approaches for PDES have focused on improving simulation performance by offloading and parallelizing certain computation on special hardwares, ranging from embedded processing units[44,45] to network interface co-processors.[46–50] Efforts have also been made towards PDES on Graphical Processing Units (GPUs),[51–53] demonstrating that the synchronous stream processing style of GPU computation can speed up discrete-event simulaitons. The MADS technique proposed in this paper is complementary to these attempts, addressing the issue of PDES on heterogeneous CMP platforms (exemplified by the Cell) based on the general-purpose DEVS methodology.

3. Discrete Event System Specification methodology and its implementation

3.1. P-DEVS and Cell-DEVS formalisms

In P-DEVS, a model is defined as a mathematical entity composed of a hierarchy of *atomic* (behavioral) and *coupled* (structural) components, which are executed by an underlying simulation engine that can be implemented as LPs specialized into *Simulators* and *Coordinators*.[20] A P-DEVS atomic model is specified formally as

$$M = < X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \text{ta} > .$$

An atomic model is in some state $s \in S$ and it will remain in state s for $ta(s)$ time units. When $ta(s)$ expires, the atomic model executes the *output function* $\lambda(s)$, which can send outputs through a set of ports (Y). The model immediately triggers the *internal transition function* $\delta_{\text{int}}(s)$, which can change its state. In this case, the model is called an *imminent* component. If an external event $x \in X$ occurs before the expiration of $ta(s)$, the model changes to a state defined by the *external transition function* $\delta_{\text{ext}}(s,e,X^b)$, where e is the elapsed time since the last transition and X^b is a *bag* to collect the simultaneous external events received at a given virtual time. A *confluent transition function* $\delta_{\text{con}}(s,e,X^b)$ decides the new state in the case of state transition collisions, when δ_{int} and δ_{ext} occur at the same virtual time.

A P-DEVS coupled model specifies how its components are connected with each other and with the external environment in a hierarchical fashion, as follows:

$$DN = < X, Y, D, \{M_d | d \in D\}, \text{ EIC, EOC, IC} > .$$

The input/output (I/O) ports and values are defined by X and Y. The external input and output couplings are specified by EIC and EOC respectively, while the internal coupling within the coupled model is specified by IC. The basic components (D and M_d) are P-DEVS structures.

Cell-DEVS defines n-dimensional cell spaces as discrete-event models where each cell is specified as a P-DEVS atomic model with explicit timing delays,[21] as follows:

$$TDC = < X^b, Y^b, S, N, \text{d}, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \tau, \tau_{\text{con}}, \lambda, D > .$$

A cell can exchange bags of inputs and outputs (X^b and Y^b) with a set of neighboring cells (N) and other P-DEVS components outside of the cell space. A cell's future state is determined by the *local transition function* (τ), based on its current state, and the values arrived at the input ports. If the future state is different from the current one, a state change is scheduled and the value of the future state will be transmitted after a delay period (d). As in P-DEVS atomic models, the output and state transitions are defined by λ, δ_{int}, and δ_{ext}. In case of transition collisions, δ_{con} activates the *confluent local transition function* (τ_{con}), which presents a unique set of inputs for the cell to compute the next state.

The cells are coupled with each other through the neighborhood relation to form a cell space, as follows:

$$GCC = < X_{\text{list}}, Y_{\text{list}}, X, Y, n, \{t_1, \ldots, t_n\}, N, C, B, Z > .$$

A cell space (C) is a coupled model with a fixed size ($t_1 \times \cdots \times t_n$). The neighborhood set (N) specifies the relative positions of the neighboring cells. The cells on the border of the cell space are included in set B. The interface of the cell space itself is defined by the I/O couplings (X_{list} and Y_{list}), whereas the coupling between cells inside the cell space is given by the Z function, which translates an output of a cell into an input of a neighboring cell.

3.2. DEVS-based simulation in CD++

As illustrated in Figure 1, CD++ adopts a *flat LP structure* to parallelize a DEVS-based simulation on distributed-memory multiprocessors.[54] The sequential simulation on a node is executed by a *Node Coordinator* (NC), a *Flat Coordinator* (FC), and a group of *Simulators*. The NC is responsible for inter-node MPI messaging and virtual time management on the host, while the FC is in charge of synchronizing the child Simulators and routing events between the local LPs based on user-defined model coupling. A Simulator is paired with an atomic model to trigger the model's behavior during a simulation. The behavior of P-DEVS atomic models is defined by implementing the output and transition functions in C++, whereas a built-in specification language is provided for specifying the behavior of coupled and Cell-DEVS models using a set of *descriptive* transition rules.[22] This extra level of abstraction is valuable on multicore platforms, since modelers can focus on their modeling issues without being distracted by technical details.

The simulation is driven by events of two kinds: *content and control events*. Content events, *external* (X) and *output* (Y), carry input and output data. Control events, *initialization* (I), *collect* (@), *internal* (*), and *done* (D), control the execution at each virtual time. Specifically, (I), (@), and (*) are passed down from the NC to trigger initialization, output, and state transitions, respectively, at the appropriate Simulators, whereas (D) events are sent up the hierarchy to deduce the next local simulation time. Detailed event-processing algorithms can be found in Liu.[55]

A schematic view of DEVS-based simulation is given in Figure 2, where three Simulators execute at different virtual times (the FC and the NC are not shown for simplicity).

In addition to FEL (Future Event List)-based scheduling, the coordinators execute DEVS-specific tasks as part of their event processing. In particular, the FC synchronizes its child Simulators in two functions: `findMinTime` and `findImminents`. When the *last* Simulator S_1 finishes its state update at current virtual time t_1, it sends a (D) event to the FC carrying the timestamp of the next scheduled state change (i.e. t_4). In response, the FC invokes `findMinTime` to find the minimum timestamp among all child Simulators. This minimum time is sent to the NC to determine the next local simulation time, taking into account events that may arrive from the other nodes and/or the environment. The NC then notifies the FC of the new local simulation time, say t_4, and the FC invokes `findImminents` to obtain the IDs of the child Simulators with state changes scheduled at t_4 and sends activation events to them. The activated Simulators (i.e. S_1 and S_2) thus advance their virtual time to t_4 and trigger state updates in their associated atomic models.

In Liu and Wainer,[54] a *multi-phased* abstraction is used to represent the sequential simulation on a node, illustrating a high-level event execution pattern at each virtual time, as shown in Figure 3. The simultaneous events sent between the LPs at a virtual time are organized into an optional ***collect phase*** and a mandatory ***transition phase***. The simulation begins with an

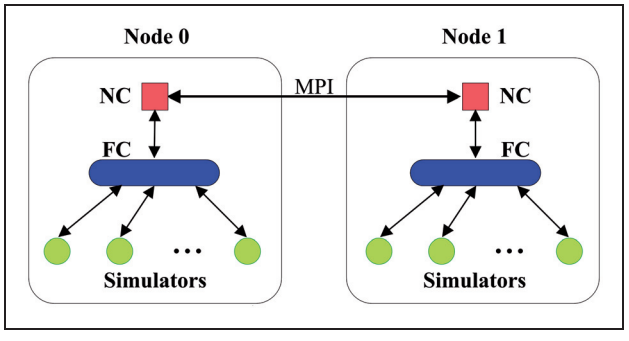


Figure 1. Flat logical process structure.

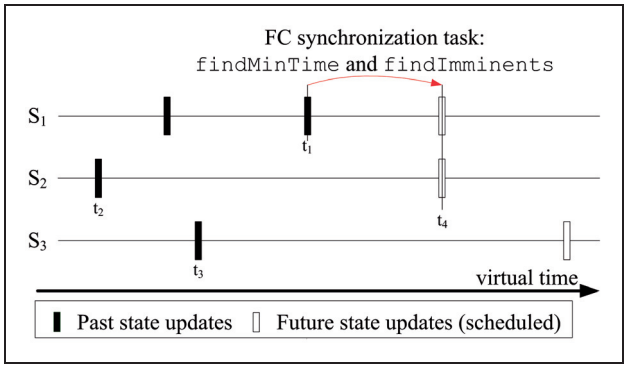


Figure 2. Schematic of Discrete Event System Specification-based simulation.

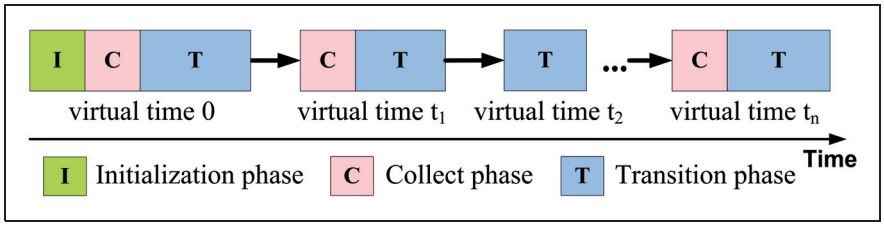


Figure 3. Multi-phased sequential simulation process.[54]

initialization phase. Each phase is initiated by a control event from the NC to the FC, and ended by a (D) event returned to the NC. The control events exchanged between the NC and the FC are also called *phase-changing events*. At the end of a transition phase, the NC determines the next local simulation time and advances the simulation on the node.

As noted by Zeigler,[56] DEVS-based simulation can involve a large number of simultaneous events, especially in large-scale, densely interconnected, highly active models where many imminent components send outputs at the same time.

4. Workload analysis and preliminary optimization

CD++ was ported to the PPE of a Cell processor, resulting in a *sequential* implementation called CD++/PPE. In order to address our goals, preliminary tests were carried out based on two Cell-DEVS models that represent two classes of common workloads: a large-scale simulation over a long time period, and a highly active simulation with complex behavior. These Cell-DEVS models are *discrete-event* models, and they do not use traditional continuous or time-stepped approaches. The cells execute asynchronously with respect to each other; they determine their own time advance independently and become inactive when not used. These models are considered as appropriate benchmarks also because they cover the types of computation that can benefit the most from heterogeneous CMP architectures (i.e. memory-bound and compute-bound). The analysis focuses on the core computational kernels in DEVS simulations (synchronization and event execution). The integration with other types of kernels (e.g. random number generation) is discussed in Section 6.3.

4.1. Computational kernels

4.1.1. Large-scale simulation over a long time period. In these simulations, a large number of Simulators are synchronized by the FC on each node at distinct virtual times, incurring a significant synchronization overhead. This computational kernel is prominent in the wildfire propagation model presented by Wainer,[57] which uses a large two-dimensional (2D) cell space (1024×1024) to simulate fire-spreading scenarios following the Rothermel method.[58] Each cell maintains its own ignition time (zero for non-burning cells). When a cell is ignited, it sends events to its neighboring cells notifying them of the ignition time. In response, each neighboring cell calculates its own earliest ignition time based on the distance to the ignited cell, the type of fuel, and the weather conditions. It then updates its state and sends the new ignition time after a delay, representing the difference between the current virtual time and the new ignition time. The simulation differs from a time-stepped approach in that the cells are asynchronous and use variable delays. Moreover, only a small fraction of the cells (i.e. those on the fire frontline) is active at any time, making the simulation more accurate and efficient than a time-stepped simulation of the Rothermel model.

Table 1 gives the execution profile obtained with CD++/PPE on an IBM BladeCenter QS22 server, showing the distribution of execution time across the major system components and the different types of events processed by the LPs. It is clear that the main bottleneck resides at the FC, which takes over 99% of the runtime. Further, the FC spends most of the time on processing (@) and (*D*) events, during which the Simulators are synchronized.

A closer look at the *FC synchronization task* shows that the sources of the bottleneck are the synchronization functions `findImminents` and `findMinTime`. As shown in Table 2, each function occupies 99.98% of the execution time on processing the corresponding type of events at the FC. Together, they constitute the dominant bottleneck (99.4% of the runtime), not only because they are invoked frequently, but also because a large amount of timing data are processed in each invocation to synchronize all the child Simulators.

Table 1 also shows a secondary bottleneck at the Simulators, consuming 0.32% of the runtime. A major component is the execution of (*) events, where the transition functions are evaluated. This bottleneck depends on the complexity of the model behavior. It is relatively minor in this example because the wildfire model uses simplified rules. If higher fidelity

Table 1. 1024 × 1024 wildfire execution profile on the Power Processor Element

	Components				
Event type	Simulators	FC	NC	Bootstrap	Other overhead
(*I*)	3.06	0.90	–	–	–
(*)	515.69	16.96	–	–	–
(@)	8.13	55,816.60	–	–	–
(*X*)	11.94	0	–	–	–
(*Y*)	–	94.41	–	–	–
(*D*)	–	112,215.00	3.25	–	–
Sum (s)	538.82	168,143.87	3.25	181.57	134.58
Total (s)			169,002.10		

FC: Flat Coordinator, NC: Node Coordinator.

Table 2. Flat Coordinator synchronization task in 1024 × 1024 wildfire

Function name	No. of invocations	Accumulated runtime (s)
`findImminents`	535,549	55,804.30
`findMinTime`	1,071,099	112,189.00

were required, more complex rules would be used, leading to a higher computational cost.

Table 3 gives the event counts in the simulation, which includes over one million simulation phases, as indicated by the numbers of phase-changing events. Out of the over 49 million events executed, the phase-changing events constitute only 4.31% of the population, while all the others are simultaneous events executed within different phases at distinct virtual times.

4.1.2. Highly active simulation of complex model behavior. Another type of computational kernel exists in highly active simulations of complex behavior, where the performance is dominated by the computation required to execute the transition functions, as exemplified in the watershed model discussed by Zeigler et al.[59] and later redefined by Wainer.[57] This model simulates water accumulation in a drainage basin under constant rain using a set of hydrological equations. The three-dimensional (3D) cell space ($320 \times 320 \times 2$) consists of two planes. The bottom plane defines the topographical configuration of the terrain, while the upper plane represents the heights of retained water at different cells. The transition functions compute future height values based on the initial water level, the cumulative rain precipitation, the dynamic water flow between the cells, and the soil condition. In the simulation, the cells change states

Table 3. Event counts in 1024 × 1024 wildfire

	LPs		
Event type	Simulators	FC	NC
(*I*)	1,048,576	1	–
(*)	10,285,266	535,549	–
(@)	2,076,507	535,549	–
(*X*)	18,666,212	0	–
(*Y*)	–	2,076,507	–
(*D*)	–	13,410,349	1,071,099
Sum	32,076,561	16,557,955	1,071,099
Total		49,705,615	

LP: Logical process, FC: Flat Coordinator, NC: Node Coordinator.

asynchronously, advancing their virtual time based on the hydrological dynamics. At any virtual time, only the upper-plane cells with water level changes become active.

Table 4 gives the watershed execution profile obtained with CD++/PPE on a QS22 server. As expected, the *Simulator event-processing task* becomes the primary bottleneck, representing 98% of the runtime. It is evident that the Simulators perform compute-intensive state transitions during the processing of (*) events. Unlike in the wildfire simulation, the *FC synchronization task* incurs only a negligible computational cost, as shown in Table 5, for two reasons. Firstly, the watershed model uses a much smaller cell space with reduced timing data to be processed in the synchronization functions. Secondly, the watershed simulation consists of fewer simulation phases with decreased synchronization frequency.

Table 6 gives the event counts in the watershed simulation. Although the simulation consists of just 663 phases, the event population is 6.8 times larger than what is observed in the wildfire simulation,

Table 4. 320 × 320 × 2 watershed execution profile on the Power Processor Element

	Components				
Event type	Simulators	FC	NC	Bootstrap	Other overhead
(*I*)	0.65	0.15	–	–	–
(*)	78,082.60	75.27	–	–	–
(@)	95.09	72.16	–	–	–
(*X*)	122.58	0	–	–	–
(*Y*)	–	905.40	–	–	–
(*D*)	–	16.42	0.002	–	–
Sum (s)	78,300.92	1069.40	0.002	25.22	488.12
Total (s)			79,883.66		

FC: Flat Coordinator, NC: Node Coordinator.

Table 5. Flat Coordinator synchronization task in 320 × 320 × 2 watershed

Function name	No. of invocations	Accumulated runtime (s)
`findImminents`	331	0.74
`findMinTime`	663	1.60

demonstrating a much higher level of activity with an even larger proportion of simultaneous events exchanged between the LPs.

The FC synchronization task and the Simulator event-processing task, referred to as the *FC Synchronization Kernel* (FSK) and the *Simulator Event-processing Kernel* (SEK) thereafter, represent the core computational kernels in many demanding DEVS-based simulations.

4.2. Preliminary FSK optimization

From the phase-oriented view shown in Figure 3, the FC synchronization task is performed at regular points in a simulation: `findImminents` is called at the *beginning* of a collect phase, while `findMinTime` is called at the *end* of each collect ***and*** transition phases. A closer examination, however, shows that it is unnecessary to compute the minimum next state change time at the end of *collect* phases, since these transitory phases do not advance virtual time (i.e. each collect phase must be followed by a transition phase at the same virtual time, as required by the P-DEVS formalism). Therefore, it is safe to eliminate the redundant invocations of `findMinTime` in all of the collect phases.

This seemingly trivial optimization was not considered in the sequential CD++ because the original algorithms were developed from an *event-oriented* perspective (focusing on the processing of individual events and overlooking the intrinsic correlation between events

Table 6. Event counts in 320 × 320 × 2 watershed

	LPs		
Event type	Simulators	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		339,221,007	

LP: Logical process, FC: Flat Coordinator, NC: Node Coordinator.

executed at different points in a simulation). Consequently, the FC is *memoryless* in terms of event execution. When a (*D*) event is received from a Simulator, the FC cannot determine if this is the result of a (@) or (*) event previously sent to that Simulator. To ensure correct synchronization, the FC has to invoke `findMinTime` for *every* last (*D*) event from the Simulators, even though it suffices to do so only if the (*D*) event is a logical consequence of a previous (*) event.

To implement this optimization in CD++/PPE, the FC is turned into a *context-aware* LP that keeps track of the type of the current phase so that `findMinTime` is called at the *end* of transition phases only. The performance gain is immediate in large-scale, long-running simulations.

In the updated wildfire execution profile given in Table 7, the time for processing (*D*) events at the FC is decreased by 50.5% as `findMinTime` is invoked less frequently, and the overall performance is improved by 34% accordingly.

Depending on the relative weight of the FSK, performance can also improve in highly active simulations of complex behavior. In the watershed simulation,

Table 7. 1024 × 1024 wildfire execution profile on the Power Processor Element (Flat Coordinator (FC) synchronization optimized)

	Components				
Event type	Simulators	FC	NC	Bootstrap	Other overhead
(*I*)	3.07	0.91	–	–	–
(*)	497.40	14.38	–	–	–
(@)	7.66	55,044.50	–	–	–
(*X*)	11.79	0	–	–	–
(*Y*)	–	93.52	–	–	–
(*D*)	–	55,526.50	2.32	–	–
Sum (s)	519.92	110,679.81	2.32	180.65	121.89
Total (s)			111,504.59		

NC: Node Coordinator.

however, this optimization does not lead to a noticeable improvement, since the overall performance is primarily dominated by the SEK.

5. Multicore acceleration of Discrete Event System Specification systems

The MADS technique explicitly exploits the data and event parallelism inherent in DEVS-based simulations to accelerate both the FSK and the SEK. As data locality has a significant impact on performance,[60,61] this issue is addressed carefully in the MADS technique. Note that the array-based data management proposed in this section is also applied to CD++/PPE to improve data locality in the main memory for optimized sequential performance, as analyzed in Section 6.

5.1. Architecture overview

Figure 4 shows an overview of the MADS technique. During simulation bootstrap, the *PPE main thread* spawns a *PPE helper thread*, which in turn creates a set of *SPE threads* (one on each SPE). The NC and the FC are executed by the two PPE threads respectively, sharing the FEL in the main memory. The SPE threads are divided into two groups: one for the FSK and the other for the SEK. The number of SPE threads in a group can be adjusted based on the relative weights of the kernel computation (in terms of the *size* and *duration* of the simulation, as well as the *complexity* of the simulated model). If no SPE thread is allocated to a group, the kernel runs on the PPE helper thread instead. This is useful if the performance is dominated by just one kernel, obviating the need for paralleling the other one that constitutes a negligible bottleneck.

The kernels are orchestrated by mailbox messages sent between the PPE helper and the SPE threads through the EIB channels. The simulation data used by the kernels (e.g. events, states, transition rules, and Simulator timing data) are managed in different buffers allocated in the main memory. These data are fetched and stored across memory domains using SPE-initiated double-buffered DMA. When the simulation starts, the addresses of the buffers are passed to the SPE threads in a control block. 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.[26] This provides the rationale behind the data management used in the MADS technique.

The FSK and SEK algorithms realize multi-grained parallelism as follows.

5.1.1. Multi-grained parallelization of the FSK.

Thread-level parallelism is applied across SPEs. They host FSK instances working on different chunks of the Simulators' timing data (i.e. next state change times) to exploit *data-level parallelism*. *Data-streaming parallelism* is used on each SPE to process data as a stream of blocks, hiding memory latency with double-buffered DMA. The FC synchronization functions are implemented using SPE SIMD intrinsics to explore *vector parallelism*, and *loop-level parallelism* is achieved by unrolling the compute-intensive loops in the functions.

5.1.2. Multi-grained parallelization of the SEK.

The SEK algorithms realize *thread-level parallelism* both on the PPE, and between the PPE and a group of SPEs. Each SPE thread executes a SEK instance. Double-buffered DMA is applied at multiple layers to transfer pending job IDs, event and state data of individual jobs, and rule data of Cell-DEVS models, hiding memory latency with *data-streaming parallelism*. The SEK algorithms are implemented on SPEs using SIMD intrinsics whenever possible, exploring *vector parallelism*. Due to the irregular nature of the computation, only partial vectorization is applied to the most

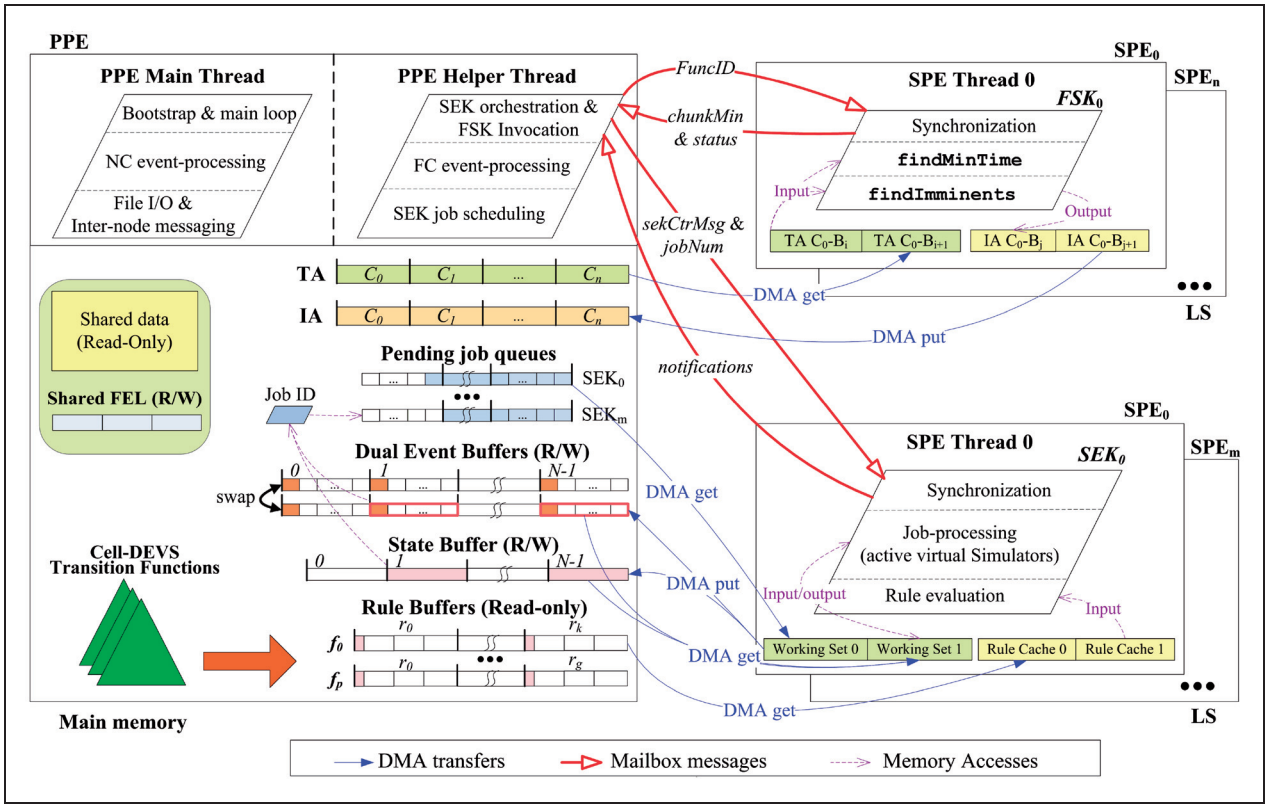


Figure 4. Architectural overview of the Multicore Acceleration of DEVS Systems technique on the Cell processor.
DEVS: Discrete Event System Specification.

time-consuming loops in the SEK. During the simulation, the PPE main thread handles file I/O and inter-node MPI messaging in parallel with event processing on the helper and SPE threads, exploiting the PPE hardware multithreading to realize *compute-I/O parallelism*. During each phase, independent events of different active Simulators are executed concurrently at distinct SPE threads, realizing *event-embarrassing parallelism*. Moreover, *event-streaming parallelism* is utilized by executing the causally related events passed between the Simulators (on the SPEs) and the FC (on the PPE) in a two-stage pipeline. These two types of event-level parallelism are analyzed next.

5.2. Event-level parallelism

A key challenge to the parallelization of the SEK is to expose the event-level parallelism that is inherent in DEVS-based simulations from a *data-flow* perspective. To this end, Figure 5 shows a step-by-step view of event execution in different phases based on the flat LP structure. During the *initialization phase*, the FC forwards the (I) event from the NC to **N** child Simulators. Each Simulator then returns a (D) event to the FC, which sends a (D) event back to the NC. A similar pattern can be seen in a *transition phase*, except that in this case the FC sends (*) events to only **K** *active* Simulators ($K \leq N$), which have state transitions scheduled at the current virtual time. In a *collect phase*, model outputs are emitted from **M** *imminent* Simulators ($M \leq N$) in response to the (@) events from the FC. These (Y) events are routed to their destinations as (X) events, which are then consumed by the receiving Simulators. Likewise, the FC sends a (D) event to the NC after processing all of the (D) events from the Simulators.

Two types of fine-grained event-level parallelism exist in the simulation, referred to as *event-embarrassing parallelism* and *event-streaming parallelism*,[25] which can be exploited as follows without violating causal consistency.

- **Event-embarrassing parallelism** exists between the *causally independent* events *within* each step (shaded *at* the FC and the Simulators). As there is no causal or data dependency between them, these events can be executed concurrently in an arbitrary order.
- **Event-streaming parallelism** exists between the *causally related* events in *consecutive* steps (arcs *between* the FC and the Simulators). As the outputs from the preceding step are the inputs to the next step, these events can be executed concurrently in a pipelined manner.

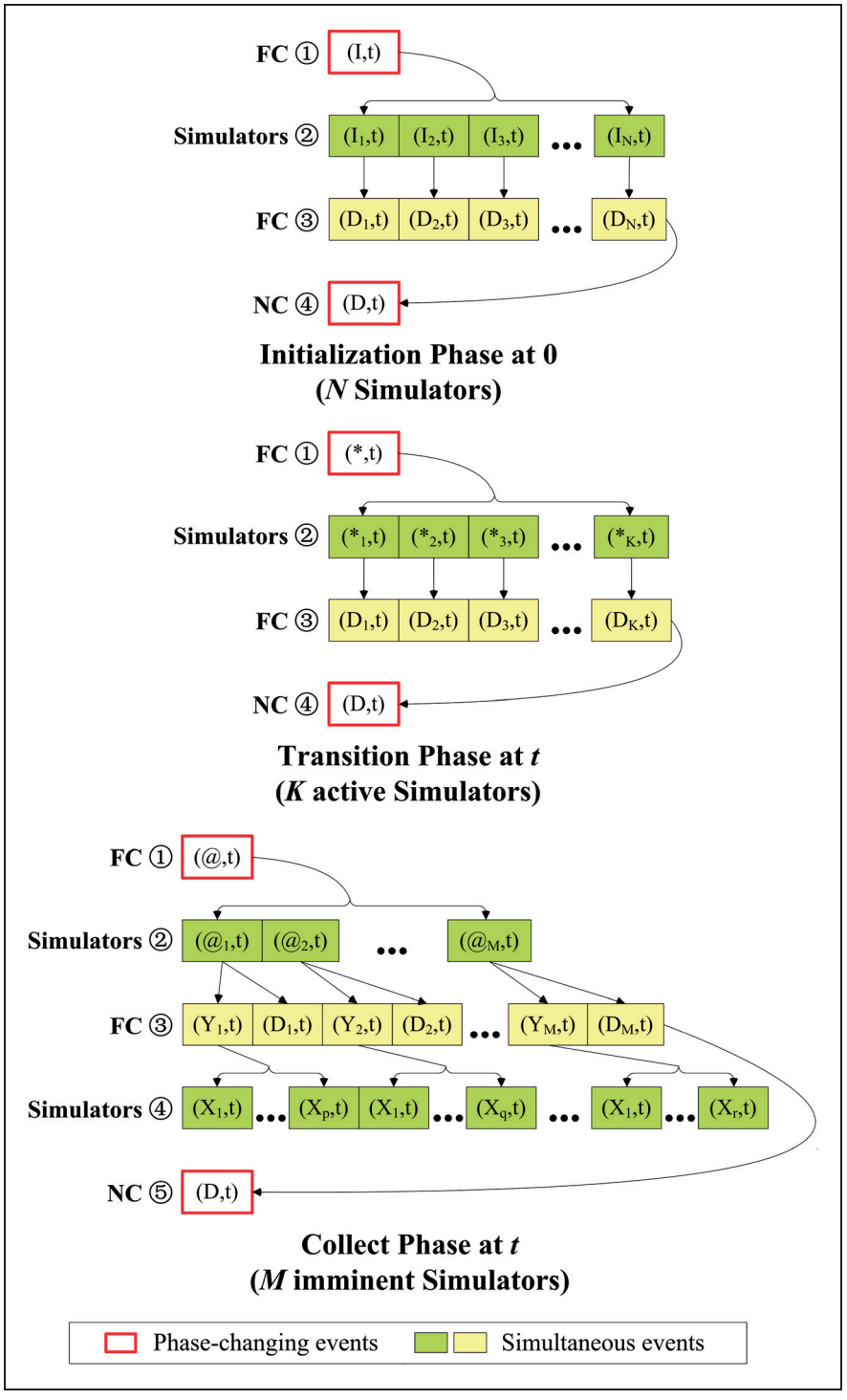


Figure 5. Event parallelism in Discrete Event System Specification-based simulation.[25]

Let us take the initialization phase to illustrate how the event parallelism can be exploited. At step 2, it does not matter which Simulator is initialized first, as long as all of the Simulators are initialized by the end of the phase. Similarly, the FC can process the (D) events from the Simulators in an arbitrary order at step 3, so far as the FC knows that all these (D) events are processed before it sends a (D) event back to the NC. In addition, the FC can immediately process the (D) events from some of the Simulators, while the other Simulators are still being executed by different threads, thus pipelining the event execution between steps 2 and 3.

Phase-changing events are sent between the NC and the FC at the first and last steps of each phase, providing natural *fork* and *join* points for synchronization. Note that, according to P-DEVS, the simultaneous (X) events received by a Simulator in the collect phase (at step 4) are cached in an internal *bag* structure. These (X) events will be consumed *as a whole* along with the (*) event received by the Simulator in the ensuing transition phase (at step 2).[20]

5.3. FSK parallelization

The FSK consists of the two synchronization functions, which rely on the Simulator timing data previously contained in a C++ Standard Template Library map that associates the IDs of the Simulators with the next state change times. The computation is inherently *data intensive* and *memory bound* in large-scale simulations.

5.3.1. Timing data management. To improve data locality and DMA efficiency, an *ID allocation scheme* is used to give positive IDs ($[0\ldots(N-1)]$) to the Simulators, whereas the NC and the FC use negative IDs. Hence, the FC can use a **Time Array (TA)** to hold the Simulator timing data, using the *array indexes* as Simulator IDs. The FC also uses an **Imminent ID Array (IA)** to contain the *imminent* Simulator IDs found by `findImminents` in a collect phase. The map structure is thus replaced by a flat array-based data layout, as shown in Figure 6.

Both arrays are partitioned into m chunks ($[C_0\ldots C_m]$) as evenly as possible, where m is the number of SPEs allocated for the FSK. Each chunk is aligned on a 128-byte boundary in the main memory for efficient DMA. Since not all of the Simulators are imminent in a collect phase, the imminent IDs are terminated by a –1 so that the FC can retrieve them without a full traversal of the array.

As the TA indexes are used *implicitly* as Simulator IDs, this method reduces the amount of data transferred across memory domains. It also facilitates the parallelization of the FSK using the data-parallel model. Moreover, simulation performance can benefit from the improved data locality even when the FSK is executed on the PPE alone (refer to Section 6 for relevant performance results).

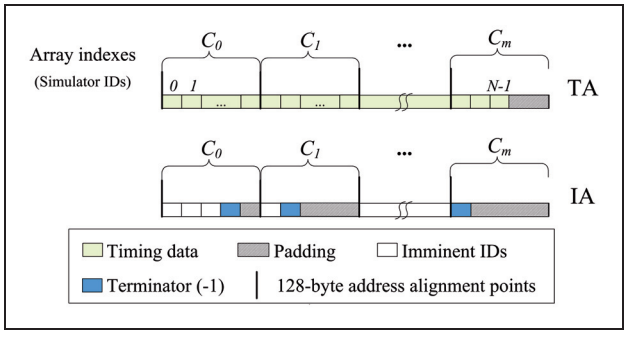


Figure 6. Flat timing data layout for the Flat Coordinator Synchronization Kernel.
TA: Time Array, IA: Imminent ID Array.

5.3.2. Parallel data processing on the SPE. As the timing data are mutually independent, different chunks of data can be processed in parallel at different SPEs. In large-scale simulations, each chunk can contain a large amount of values. Due to the limited size of LS, it is necessary to divide a TA chunk further into a set of blocks of regular sizes, which can be adjusted for different models and optimal DMA performance. An IA chunk is also divided into the same number of blocks.

On a SPE, four local buffers are allocated in the LS: *two input buffers* for reading data from a given TA chunk, and *two output buffers* for writing imminent Simulator IDs to the corresponding IA chunk. The local buffers have the same size as a TA block. Note that using a larger block size (e.g. 16 kB) not only improves DMA performance as more data are transferred in one stroke, but also increases the granularity of computation on the SPEs, making it more likely to overlap SPE computation with concurrent memory I/O.

Figure 7 illustrates the parallel data processing using multiple SPEs. Each SPE streams in and out data blocks with double-buffered DMA. The synchronization functions are vectorized using SPE SIMD intrinsics to process data in the local buffers.

As shown in Figure 8, `findMinTime` uses a 128-bit, four-way MinVector to scan the timing data prefetched into the local input buffer (lines 8–11). When the full chunk of data is processed, the MinVector contains the four minima obtained in the four ways. These values are merged into the chunk minimum (line 13), which is sent to the PPE through the outbound mailbox channel.

Given in Figure 9, `findImminents` stores the starting index of a TA chunk locally in *baseId*. The *global* minimum, determined on the PPE, is replicated in the MinVector (line 3). A 128-bit, four-way IndexVector is used to keep track of the four Simulator IDs corresponding to the entries in the current input buffer when the timing data are shifted using the MinVector (lines 10–15). At the end of the function, a status is sent to the PPE, indicating that the imminent IDs are available in the IA chunk.

Although not shown explicitly, the compute-intensive loops (lines 8–11 in Figure 8 and lines 10–15 in Figure 9) can be unrolled by using multiple Min and Index Vectors.

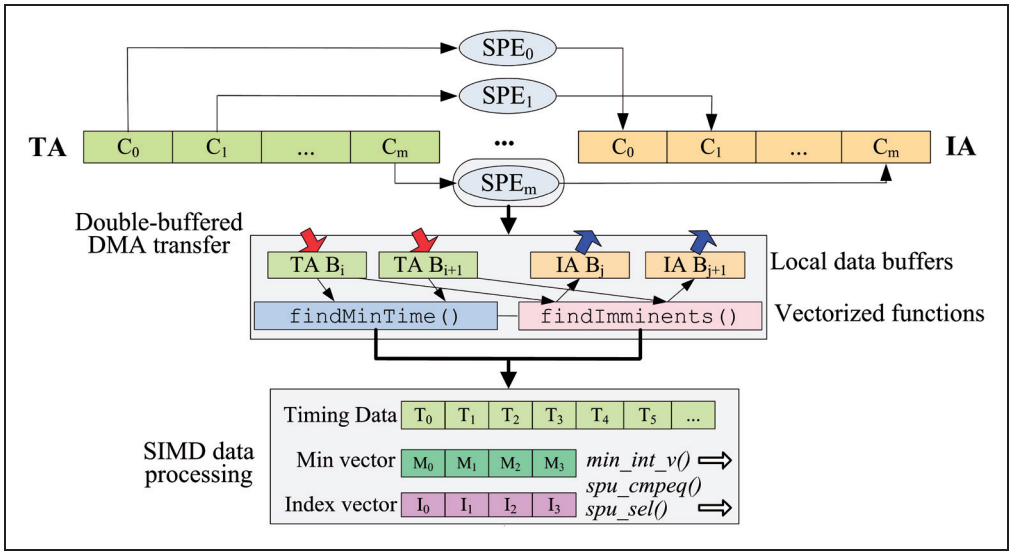


Figure 7. Parallel processing of Simulator timing data on the Synergistic Processing Elements.[24]

Input: *MinVector, inputBuf[2]*

1. **when `findMinTime`** is invoked
2. Initialize *MinVector* = {Inf, Inf, Inf, Inf}
3. Calculate *numOfTransfers* based on chunk size and block size
4. Schedule an inbound DMA from the 1st TA block to *inputBuf[0]*
5. **for** i = 0 to (*numOfTransfers* − 1) **do**
6. Wait for ith inbound DMA to complete in *current* input buffer
7. Schedule (i+1)th inbound DMA from next TA block to *next* input buffer
8. **for** j = 0 to (block size / 4 − 1) **do**
9. Compare *MinVector* to 4 values in current input buffer
10. Update *MinVector* to record the smaller values
11. **end for**
12. **end for**
13. Merge *MinVector* to get chunk-wise min time
14. Send chunk-wise min time to PPE via outbound mailbox channel
15. **end when**

Figure 8. A skeleton of function `findMinTime`.

5.3.3. FSK orchestration. The FSK instances are invoked in a Remote Procedure Call style. A termination function is also defined to exit the FSK at the end of a simulation. Each function is associated with an ID: 0 for `findMinTime`, 1 for `findImminents`, and 2 for termination. Figure 10 gives the FSK orchestration algorithm defined on the PPE.

When the last (*D*) event is received from the Simulators at the end of a *transition* phase, the FC invokes `findMinTime` by sending a mailbox message of 0 to *all* of the FSKs. When the chunk minima are returned, the FC merges them into a global minimum, and sends it to the NC. Note that `findMinTime` is invoked *in place* by the FC in the sense that the FC is blocked until all the chunk minima are available. Moreover, the returned values are recorded in the main memory to improve the performance of `findImminents` in the next collect phase.

In contrast, `findImminents` is invoked *in advance* by the NC, when the local simulation is about to advance to the next virtual time (line 2.1). Hence, when the FC needs to retrieve the imminent IDs (in the next *collect* phase), the FSKs have already executed, overlapping the computation on the PPE and SPEs. Further, a

```
Input: MinVector, IndexVector, inputBuf[2], outputBuf[2], baseId

1. when findImminents is invoked
2.   Read global min time from inbound mailbox channel
3.   Replicate global min time in MinVector
4.   Initialize IndexVector = {baseId, baseId+1, baseId+2, baseId+3}
5.   Calculate numOfTransfers based on chunk size and block size
6.   Schedule an inbound DMA from the 1st TA block to inputBuf[0]
7.   for i = 0 to (numOfTransfers - 1) do
8.     Wait for ith inbound DMA to complete in current input buffer
9.     Schedule (i+1)th inbound DMA from next TA block to next input buffer
10.    for j = 0 to (block size / 4 - 1)  do
11.      Compare MinVector to 4 values in current input buffer
12.      Select Simulator IDs from IndexVector based on MinVector
13.      Write selected Simulator IDs continuously to current output buffer
14.      Increment IndexVector by 4
15.    end for
16.    Wait for (i-1)th outbound DMA to complete in previous output buffer
17.    Schedule ith outbound DMA from current output buffer to IA block
18.  end for
19.  Wait for the last outbound DMA to complete
20.  Notify PPE via the outbound mailbox channel
21.end when
```

Figure 9. A skeleton of function `findImminents`.

```
1.1. when the last (D) is received by the FC (end of a transition phase)
1.2.   Send 0 to each FSK via inbound mailbox channel
1.3.   Wait for chunk-wise min times to be returned from all of the FSKs
1.4.   Record chunk-wise min times for later use in the next collect phase
1.5.   Merge chunk-wise min times to obtain the global minimum time
1.6. end when

2.1. when NC is about to send a (@) event to the FC
2.2.   for each FSK with recorded chunk-wise min = current global min do
2.3.     Send 1 and the current global min time to the FSK as 2 mailbox messages
2.4.   end for
2.5. end when

3.1. when a (@) event is received by the FC (beginning of a collect phase)
3.2.   for each FSK that has been invoked by the NC do
3.3.     Wait for a status value to be returned from the FSK
3.4.     Fetch imminent IDs from the corresponding IA chunk
3.5.   end for
3.6. end when

4.1. when NC is about to terminate the simulation
4.2.   Send 2 to each FSK via inbound mailbox channel
4.3. end when
```

Figure 10. A skeleton of the Flat Coordinator Synchronization Kernel orchestration algorithm.

FSK runs only if it has found the current *global* minimum (line 2.2). As imminent IDs may not exist, this reduces the number of SPE threads used, reducing memory contention with enhanced FSK scalability.

During a *collect* phase, the FC waits for a status message from each FSK invoked by the NC and retrieves the imminent IDs from the corresponding IA chunk (line 3.4). At the end of a simulation, the NC terminates the FSKs (line 4.2).

5.4. SEK parallelization

The SEK includes the Simulator algorithms for processing (I), (@), and (*) events, as well as the P-DEVS functions of the atomic models. Note that the SEK does not include the Simulator algorithm for processing (X) events, as will be explained shortly. During each phase, the SEK processes a set of input events from the FC based on the current states of the active Simulators, and returns a set of output events back to the FC. The events and states are handled *independently* for different Simulators.

5.4.1. LP virtualization. Several issues need to be addressed when mapping the SEK to the SPEs. Firstly, the small size of the LS imposes a tight limit on the number of Simulators hosted simultaneously on a SPE. Secondly, a SPE can execute only one thread at a time; a SPE context switch is very expensive.[62] Hence, it is impractical to swap in and out Simulators as SPE threads without incurring an excessive overhead. Thirdly, not all Simulators are active at any virtual time; hence, the partitioning scheme should map only active Simulators to the available SPEs. Finally, the partitioning scheme should also facilitate dynamic load balancing between SPEs.

We addressed these issues through the concept of *LP virtualization*. The Simulators (and their associated atomic models) are turned into *virtual LPs*, sharing the functionality provided by a limited group of SPE threads; the mapping of active Simulators to the SPEs is determined dynamically at each virtual time. The state data originally encapsulated in a Simulator–atomic pair are separated from the event-processing logic. While the state data are maintained in the main memory, events are processed on the SPEs. The state of an *active* Simulator is matched to a SPE thread using a SEK job-scheduling algorithm, whereas the PPE is used to host the remaining *concrete LPs*, such as the NC and the FC.

The following discussion assumes that *all* of the Simulators are mapped to the SPEs. However, as will be discussed in Section 6.3, certain types of Simulators might still need to be implemented as concrete LPs on the PPE.

5.4.2. Virtual LP state management. With the ID allocation scheme introduced earlier, the current states of the Simulators are stored in a flat 128-byte aligned array called the *state buffer* in the main memory, using the array indexes as Simulator IDs, as shown in Figure 11.

Each state buffer entry has an adjustable size of 512 bytes to contain the state variables extracted from a Simulator–atomic pair. On a SPE, a pair of 128-byte aligned ***local state caches*** is allocated to hold the states of *at most two* active Simulators. Besides the improvement in data locality, this circumvents the limitation of small LS size, reducing memory latency by prefetching the state of the *next* active Simulator in parallel with event execution of the *current* one.

5.4.3. Decentralized event management. To transfer events across memory domains, the raw data of each CD++ event are encoded in 32 bytes; a pair of flat 128-byte aligned arrays (***current event buffer*** and ***backup event buffer***) is allocated in the main memory to exchange *simultaneous* events between the FC and the virtual Simulators. Each event buffer entry has an adjustable size of 1 kB to hold up to 32 events at a time for a dedicated virtual Simulator. At any step, the FC and a Simulator may exchange exactly one control event and optionally a list of content events. Hence, the first slot in each event buffer entry is reserved for passing control events, whereas the following slots are for content events. This convention allows for separating the control and content events without checking their actual types. On a SPE, a pair of 128-byte aligned ***local event caches*** is allocated to hold the events for the *current* and the *next* active Simulators.

The FEL is used to send *phase-changing events* between the NC and the FC only at the beginning and the end of a simulation phase. Together, the FEL and the event buffer entries form a network of

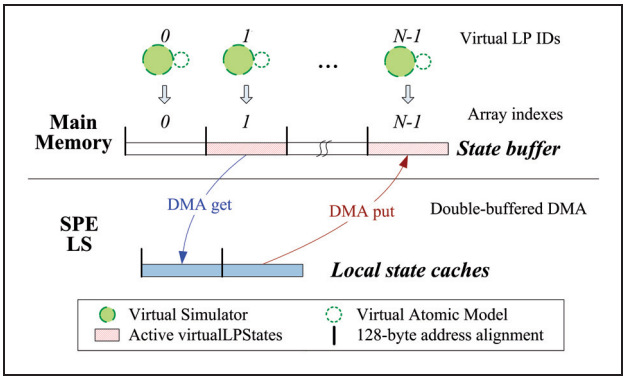


Figure 11. Virtual Simulator state management.[25]
LP: logical process, DMA: direct memory access, LS: local storage.

bidirectional communication channels with a star topology centered at the FC, as shown in Figure 12.

During a collect phase, the FC translates (*Y*) events from the source Simulators into (*X*) events to be received by the destination Simulators. As mentioned in Section 5.2, these (*X*) events need to be *cached* temporarily at the destination Simulators so that they can be consumed along with the (*) events in the *next* transition phase. To fulfill this P-DEVS requirement (while preventing data corruption), the FC caches the events on behalf of the Simulators. To do so, it writes the translated (*X*) events into the corresponding entries of the backup event buffer instead of the current event buffer, as this could still contain (*Y*) events generated in the *current* collect phase. For example, as seen in Figure 12, Simulator (*N*–1) sends a (*Y*) event that will be received by Simulator 0, 1, and itself as (*X*) events. If these events were written into the current event buffer, they would overwrite the (*Y*) events being processed by the current active Simulators in entry 1 and (*N*–1). 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 explicit synchronization. After writing the (*) events and any additional (*X*) events in the backup event buffer at the beginning of the ensuing transition phase, the FC resets a flag, `eventBufferIndex`, to swap the two event buffers. On the other hand, the virtual Simulators always work on the current event buffer determined by the FC.

This decentralized event management has several advantages. Firstly, multiple I/O events of a Simulator are stored in the same event buffer entry, allowing them to be transferred efficiently in a single DMA operation. Secondly, all simultaneous events are removed from the FEL, reducing event queue operational cost. Thirdly, the simultaneous events are read/written directly in the event buffers without memory allocation and de-allocation, further reducing the operational overhead. Fourthly, the Simulators no longer need to cache (*X*) events explicitly in collect phases, simplifying the SEK algorithms. As (*X*) and (*) events scheduled for a Simulator are packed in the same buffer entry by the FC, they can be consumed as a whole in the transition phases, satisfying the P-DEVS requirement without introducing extra synchronization overhead. Finally, the PPE cache memory is better utilized because of increased event data locality.

5.4.4. Rule evaluation on the SPE.

As discussed in Section 3, the behavior of Cell-DEVS models is defined by a set of *local transition functions*, each of which includes several *state transition rules* that are evaluated by active cells.[22] A transition rule is composed of three expressions: a *post-condition*, a *delay*, and a *pre-condition*. During evaluation, a rule is fired if its pre-condition is true; the post-condition defines the cell's future state, which is transmitted after a period derived from the delay expression.

Originally, these transition rules are represented as syntax trees that are evaluated *recursively* at runtime.[22] However, recursion on the SPEs is problematic due to the limited size of runtime call stack and the lack of stack overflow protection.[62] Moreover, syntax trees are not well suited for efficient DMA. To solve these problems, the syntax trees are converted into a sequence of floating-point values in *postfix* format and concatenated in a flat 128-byte aligned array (***rule buffer***) in the main memory, as shown in Figure 13.

Each packed rule has four components: *pre-condition tree*, *delay tree*, *post-condition tree*, and *rule header*, which indicates the number of nodes in the syntax trees. Each syntax node is encoded as an integer

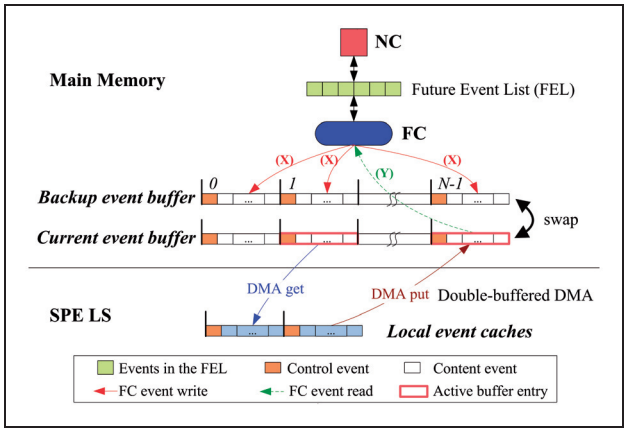


Figure 12. Virtual Simulator event management.[25]
DMA: direct memory access, NC: Node Coordinator, FC: Flat Coordinator, SPE: Synergistic Processing Element, LS: Local Storage.

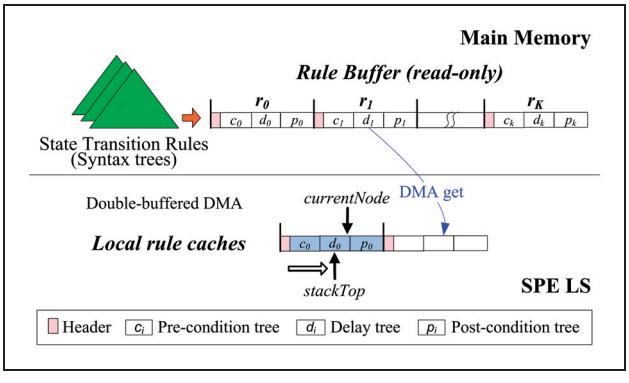


Figure 13. Packing of a local transition function.[25]
DMA: direct memory access, SPE: Synergistic Processing Element, LS: Local Storage.

operation type and an optional floating-point operand. On a SPE, a pair of 128-byte aligned ***local rule caches*** is allocated for the double-buffered DMA of rule data. As the transition rules are read only, they can be accessed simultaneously by multiple SPE threads.

The recursive algorithm is redesigned to scan the local rule cache iteratively (one syntax node at a time), as shown in Figure 14. Note that the current local rule cache is used as a *software-managed call stack* to hold intermediate operands, allowing for *in-place* rule evaluation without burdening the SPE runtime call stack.

For each type of syntax node, an evaluation function performs the required operation based on operands retrieved from the syntax node (line 1.2), the data in the current local state cache (line 2.2), or the intermediate data in the current local rule cache. The result is pushed back into the local rule cache at *stackTop*. If required, new types of syntax nodes can be defined by implementing additional evaluation functions.

```
Input: currentNode, stackTop, ruleCache[2]
       void (*evalFunctions[])(void) = {evalConstant, evalCellVar, ⋯}

1.1. when evalConstant is called       //push a constant into stack
1.2.  currentRuleCache[++stackTop] = constant operand in current node
1.3. end when

2.1. when evalCellVar is called        //push the cell value into stack
2.2.  Retrieve the value of a neighboring cell from current local state cache
2.3.  currentRuleCache[++stackTop] = retrieved cell value
2.4. end when
//Other evaluation functions are omitted here

3.1.  when a local transition function is evaluated
3.2.   Schedule an inbound DMA from 1st rule buffer entry to ruleCache[0]
3.3.   for each rule in the rule buffer do
3.4.    Wait for current inbound DMA to complete in current rule cache
3.5.    Schedule next inbound DMA from next rule to next rule cache
3.6.    Reset currentNode = 0; stackTop = -1
3.7.    for each node in pre-condition tree do
3.8.     Call evaluation function to evaluate the node
3.9.     Move currentNode to the next syntax node in pre-condition tree
3.10.   end for
3.11.   if pre-condition tree is true then
3.12.    Update stackTop to evaluate delay tree
3.13.    for each node in delay tree do
3.14.     Call evaluation function to evaluate the node
3.15.     Move currentNode to the next syntax node in delay tree
3.16.    end for
3.17.    Record the delay value obtained
3.18.    Update stackTop to evaluate post-condition tree
3.19.    for each node in post-condition tree do
3.20.     Call evaluation function to evaluate the node
3.21.     Move currentNode to the next syntax node in post-condition tree
3.22.    end for
3.23.    Record the new cell value obtained
3.24.    break;        //Skip the remaining rules
3.25.   end if
3.26.  end for
3.27.end when
```

Figure 14. A skeleton of doubled-buffered rule evaluation on the Synergistic Processing Elements.

These functions are called through the `evalFunctions` function pointer array using the operation types as array indexes, reducing branching instructions in the SPE code.

When a valid rule is identified (line 3.11), the new delay and state are computed from the delay and post-condition trees, respectively (lines 3.13–3.17 and lines 3.19–3.23), and the local transition function is terminated (line 3.24).

5.4.5. SEK job processing. A *SEK job* executes a set of events scheduled for an active Simulator, assuming that the event and state data have already been made available in the current local event and state caches in the LS, whereas memory control and SEK orchestration are considered as separate services, which will be discussed shortly. Decoupling SEK job execution from these services allows one to implement the P-DEVS functions on the SPE in a similar way as in the original CD++ (code vectorization is desired, but not absolutely required), without having to cope with issues related to DMA and thread scheduling.

The SEK job-processing algorithms are implemented by three *job handlers* invoked, respectively, during the initialization, collect, and transition phases. In essence, they follow the Simulator algorithms for processing (I), (@), and (*) events, with a few exceptions. Firstly, these algorithms are *coarse grained*, since the local event cache may contain *multiple* input events scheduled for a Simulator. All these events are processed in a single invocation of the respective job handler. In addition, the output events and updated states are written *directly* in the local event and state caches, which are then transferred to the main memory event and state buffer entries by the memory control service. Barring these subtleties, the definition of the SEK job-processing algorithms is straightforward (further details can be found in Liu[55]).

On the PPE, the IDs of the *active* Simulators are used as SEK job IDs, which are scheduled by the FC in each phase through a set of ***pending job queues***, as depicted in Figure 15. Each job queue is a 128-byte aligned integer array containing the pending job IDs scheduled for a SEK instance. A pair of ***local job caches*** is allocated on a SPE so that the job IDs can be transferred in chunks with double-buffered DMA. Each chunk has an adjustable size of 128 bytes for 32 jobs.

A SEK instance processes the job IDs in the *current* local job cache sequentially. These IDs are used as offsets to calculate the addresses of the event and state buffer entries in the main memory when accessing data of active Simulators from the SPEs.

At the beginning of a simulation phase, the FC executes the phase-changing events in the FEL, and then writes the events generated into the event buffer 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. As the SEK jobs executed in a phase are of the same type with similar computational intensity, simple yet effective scheduling policies (e.g. round-robin, shortest-queue-first, or weighted round-robin) can be used to achieve fine-grained dynamic load balancing between the SPEs.

5.4.6. SEK memory control and notification. To support SEK job processing, a memory control and notification service is provided on the SPE. As shown in Figure 16, the SEK memory control and notification algorithm is implemented in a function called `provideService`.

The purpose of the memory control service is to prefetch the input events and states of the active Simulators into the local event and state caches, and to transfer the generated output events and updated states back to the original event and state buffer entries after job execution. The transfer of events and states relies on the availability of pending job IDs in the local job caches. Hence, double-buffered DMA is performed at two layers: the *job-data layer* for transfer of pending job IDs in chunks (lines 2 and 7) and the *simulation-data layer* for transfer of events and states of individual jobs (lines 9, 11, 17, 18, 20, and 21). Once the events and states become available in the local caches, the corresponding SEK job handlers are invoked through a function pointer array called `jobHandlers` (line 19), overlapping SEK job execution with concurrent memory I/O.

On the other hand, the notification service sends signals periodically to the PPE (line 24), indicating the number of jobs that have been processed in the pending job queue since the last notification. In this way, the FC can process the output events from those finished Simulators without waiting for the completion of all

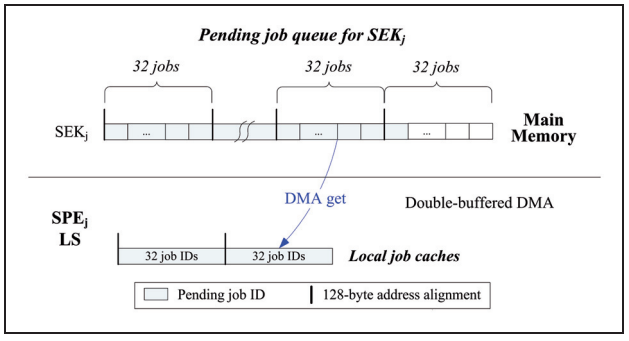


Figure 15. Pending job queues for a Simulator Event-processing Kernel instance.[25]
DMA: direct memory access, SPE: Synergistic Processing Element, LS: Local Storage.

```
Input: NOTIFY_FREQ, JOB_CACHE_SIZE
       jobCache[2], eventCache[2], stateCache[2]   //local job, event, and state caches
       sekCtrMsg, totalPendingJobs                 //messages received from PPE
       void (*jobHandlers[])(void) = {initHandler, colHandler, transHandler}

1. when provideService is invoked
2.   Schedule an inbound DMA from the 1st chunk of pending jobs to jobCache[0]
3.   Determine the address of current event buffer based on sekCtrMsg
4.   Calculate numOfChunks = totalPendingJobs / JOB_CACHE_SIZE
5.   for i = 0 to (numOfChunks – 1) do          //loop over chunks of pending jobs
6.     Wait for current inbound DMA to complete in current job cache
7.     Schedule next inbound DMA from next chunk of pending jobs to next job cache
8.     Calculate current event buffer entry address for the 1st pending job
9.     Schedule an inbound DMA from the event buffer entry to eventCache[0]
10.    Calculate state buffer entry address for the 1st pending job
11.    Schedule an inbound DMA from the state buffer entry to stateCache[0]
12.    Calculate numOfSignals = JOB_CACHE_SIZE / NOTIFY_FREQ
13.    for n = 0 to (numOfSignals - 1) do      //send one signal for every NOTIFY_FREQ jobs
14.      for k = 0 to (NOTIFY_FREQ - 1) do     //process jobs within a NOTIFY_FREQ interval
15.        Wait for the kth inbound DMA to complete in current event and state caches
16.        Wait for (k-1)th outbound DMA to complete in next event and state caches
17.        Schedule an inbound DMA to fetch input events for next pending job to next event cache
18.        Schedule an inbound DMA to fetch state for next pending job to next state cache
19.        Call SEK job handler to process events for current pending job k
20.        Schedule kth outbound DMA from current event cache back to event buffer entry
21.        Schedule kth outbound DMA from current state cache back to state buffer entry
22.      end for
23.      Wait for (NOTIFY_FREQ-1)th outbound DMA to complete
24.      Send the actual number of completed SEK jobs to PPE via outbound mailbox channel
25.    end for
26.  end for
27.end when
```

Figure 16. A skeleton of the Simulator Event-processing Kernel memory control and notification algorithm.

pending jobs, exploiting event-streaming parallelism between the PPE and the SPEs.

5.4.7. SEK orchestration. Similar to the FSK, a termination function is defined to exit the SEK at the end of a simulation. When a simulation phase starts, a SEK instance is triggered by two mailbox messages from the PPE: a control message (`sekCtrMsg`) and the total number of pending jobs scheduled for the SEK (`totalPendingJobs`). As a result, either `provideService` or the termination function is invoked.

As shown in Figure 17, the four least significant bits of the control message are used to carry three parameters, which are extracted in the SEK algorithms to determine the address of the current event buffer, the ID of the SEK function, and the ID of the SEK job handler.

The SEK orchestration algorithm consists of two parts, as shown in Figures 18 and 19. At the beginning of a simulation phase, the FC sends two mailbox messages to each SEK instance when the pending jobs have been scheduled in the job queues (lines 1.6, 1.7, 2.7, 2.8, 3.8, and 3.9 in Figure 18). The FC then waits for notifications from the SEKs. Upon notification, it processes the output events generated by the completed SEK jobs immediately, while the other active Simulators are processed concurrently on the SPEs (lines 9–14 in Figure 19).

Note that the SPE outbound mailbox channel can contain at most one message at a time;[62] thus, a SPE thread will block when there is a message that has not yet been cleaned in the channel by the PPE. For this reason, the FC uses *non-blocking* polling to quickly scan the status of the channels after processing the output events from just *one* job (even when they are generated from multiple SEK jobs) in order to reduce the possibility of blocking the SEKs. As long as there are uncompleted SEK jobs being processed on the

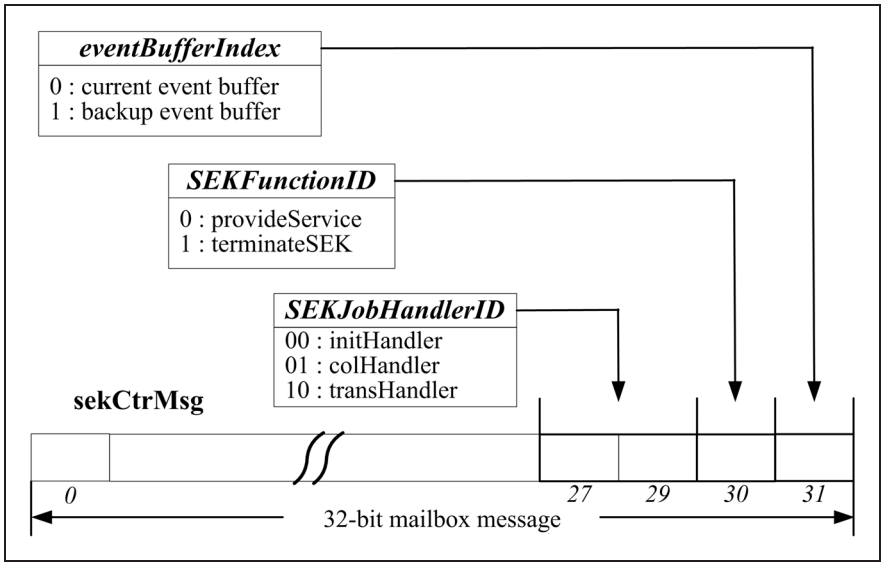


Figure 17. Bit pattern of the Simulator Event-processing Kernel control message.

SPEs, the algorithm tries to balance short channel-polling interval and concurrent event execution on the PPE.

When the SEKs finish all pending jobs, the FC executes the remaining output events and sends phase-changing events to the NC through the FEL, ending the current phase. At the end of a simulation, the NC terminates the SEKs by sending SEK_TERMINATE_SIG and zero to them.

6. Performance analysis and discussion

The sequential CD++/PPE was parallelized on the Cell using the MADS technique. The parallel simulator called **CD++/Cell** was implemented on Red Hat Enterprise Linux 5.2 with the IBM SDK for Multicore Acceleration 3.1. Using the wildfire and watershed simulations as benchmarks, this section analyzes the performance of the FSK and SEK algorithms on an IBM BladeCenter QS22 server, which features two 3.2 GHz IBM PowerXCell 8i processors with 32 GB main memory. The Cell processors in a QS22 server are connected through the Rambus FlexIO™ interface using the coherent Broadband Interface protocol,[62] allowing an application to scale transparently across the two processors (two PPEs and 16 SPEs in total). However, inter-processor communication via FlexIO has lower bandwidth and higher latency than intra-processor communication via the EIB.

The major performance metrics presented include the *total execution time* (T) of the simulation and the *turn-around time* (T) of the FSK synchronization functions. The *scale-up* metric measures how performance scales as a function of the number of SPEs involved in a computation as follows.

$$\text{Scale-up} = \frac{T(\text{PPE with one SPE})}{T(\text{PPE with } N \text{ SPEs})}, \text{ where } N > 1$$

As the PPE is very different from the SPEs, we use the execution on *the PPE with one SPE* (instead of on the PPE alone) as baseline. As a result, the scale-up is more conservative than what would be obtained from a traditional definition of speedup (based on a purely sequential execution), because the baseline case exploits a certain degree of parallelism (e.g. data-streaming parallelism and SIMD vector parallelism). In the experiments, SEK jobs were scheduled using a round-robin policy. For the FSK, the TA block size was 16 kB. To minimize the impact of file I/O, no event logging was used. The results were averaged over 20 independent runs to balance data reliability and testing effort.

6.1. Performance of the FSK algorithms

Figure 20 shows the total execution time in the 1024×1024 wildfire simulation on PPE (CD++/PPE) and across 1–16 SPEs (CD++/Cell). The PPE-based sequential executions are denoted as ORG (Table 1), SYN (Table 7), and FLT. When the various types of simulation data are managed using flat arrays in the main memory, as discussed in Section 5, the execution time (FLT) is reduced by a factor of 13.79 from what is attained with the original CD++/PPE (ORG) and by a factor of 9.09 from the synchronization-optimized execution (SYN). As the wildfire simulation performance is dominated by the FSK, which has a regular access pattern (Simulator timing data are contiguous in the main memory), the increased locality of

```
Constants: SEK_INIT_SIG = 0;            SEK_TERMINATE_SIG = 2;
           SEK_COLLECT_SIG = 4;         SEK_INTERNAL_SIG = 8;
Input:     eventBufferIndex                  //index of current event buffer
           numOfPendingJobs[numOfSEK]        //numbers of pending jobs

1.1. when (I, 0) is received by FC (beginning of initialization phase)
1.2.   Process the (I, 0) event
1.3.   Insert all Simulator IDs into pending job queues using a given policy
1.4.   Set sekCtrMsg = SEK_INIT_SIG | eventBufferIndex
1.5.   for each SEK_i do
1.6.     Send sekCtrMsg through the inbound mailbox channel
1.7.     Send numOfPendingJobs[i] through the inbound mailbox channel
1.8.   end for
1.9.   Call processSEKOutputs
1.10.end when

2.1. when (@, t) is received by FC (beginning of collect phase)
2.2.   Process the (@, t) event
2.3.   Insert imminent Simulator IDs into pending job queues using a given policy
2.4.   Set sekCtrMsg = SEK_COLLECT_SIG | eventBufferIndex
2.5.   for each SEK_i do
2.6.     if numOfPendingJobs[i] is not zero then
2.7.       Send sekCtrMsg through the inbound mailbox channel
2.8.       Send numOfPendingJobs[i] through the inbound mailbox channel
2.9.     end if
2.10.  end for
2.11.  Call processSEKOutputs
2.12.end when

3.1. when (*, t) is received by FC (beginning of transition phase)
3.2.   Process the (*, t) event
       //(X, t) and (*, t) events are now packed in the backup event buffer
3.3.   Update eventBufferIndex = (eventBufferIndex + 1) & 1    //swap event buffer
3.4.   Insert active Simulator IDs into pending job queues using a given policy
3.5.   Set sekCtrMsg = SEK_INTERNAL_SIG | eventBufferIndex
3.6.   for each SEK_i do
3.7.     if numOfPendingJobs[i] is not zero then
3.8.       Send sekCtrMsg through the inbound mailbox channel
3.9.       Send numOfPendingJobs[i] through the inbound mailbox channel
3.10.    end if
3.11.  end for
3.12.  Call processSEKOutputs
3.13.end when
```

Figure 18. A skeleton of the Simulator Event-processing Kernel orchestration algorithm (part I).

timing data improves the utilization of the PPE cache memory, leading to a significant reduction in memory contention and runtime.

This effect is illustrated in Table 8. Comparing it with Table 7, it is evident that the biggest improvement comes from the FC, and the time spent on processing (@) and (*D*) events is reduced by more than 89% from the SYN execution. In addition, the bootstrap time is decreased by 50%, as most of the simulation data are allocated and initialized as big blocks in flat arrays. The use of the event buffers also accelerates Simulator event execution and FEL operations (other overhead) by 7.1% and 23.81%, respectively.

With one SPE, the parallel simulation runs 1.78 times faster than the FLT, thanks to more efficient data processing and the exploitation of data-streaming parallelism on the SPE. Overall, the total execution time is reduced from the best sequential performance of over

```
Input: totalPendingJobs          //sum of numOfPendingJobs[i]
1.  when processSEKOutputs is invoked
2.    while totalPendingJobs is not zero do
3.      for each SEK that has been invoked do
4.        non-blocking poll outbound mailbox channel for SEK notifications
5.        if a notification signal is received from SEK_i then
6.          Mark the completed job IDs in SEK_i's pending job queue
7.        end if
8.      end for
9.      for each non-empty pending job queue do
10.       for each completed SEK job with unprocessed output events do
11.         Process the output events in the current event buffer entry
12.       end for
13.       break;                          //go back to polling (line 3)
14.     end for
15.   end while
16.   Process output events for any remaining completed SEK jobs
17.   Send phase-changing events to the NC through the FEL
18. end when
```

Figure 19. A skeleton of the Simulator Event-processing Kernel orchestration algorithm (part II).

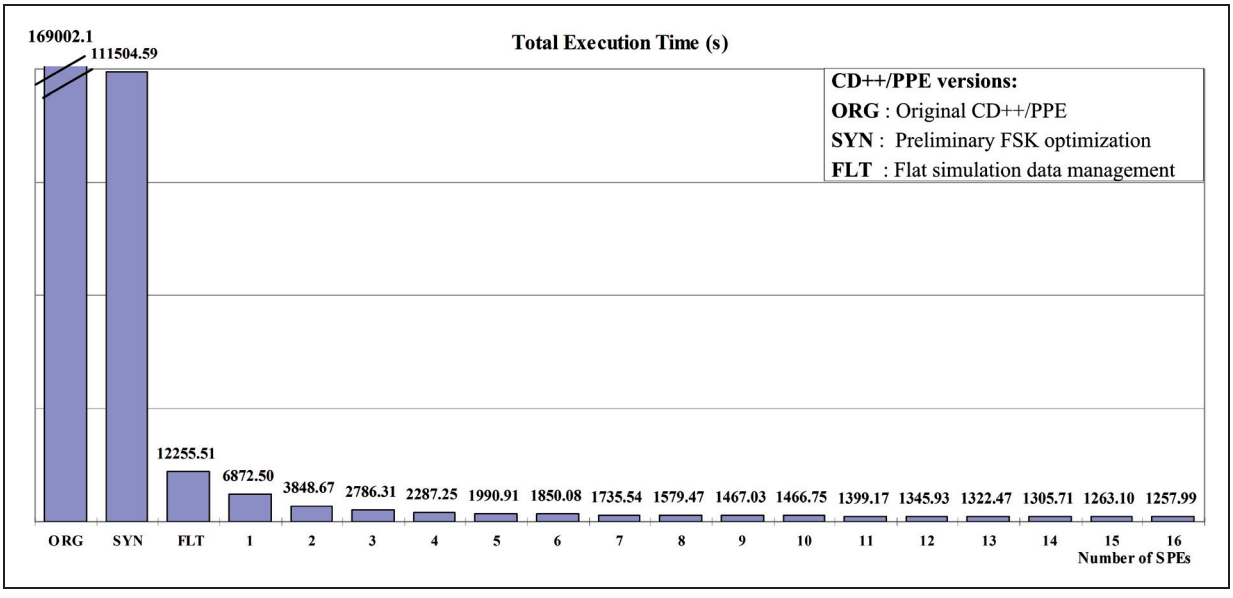


Figure 20. Total execution time in 1024 × 1024 wildfire simulation.

3 hours (FLT) to approximately 20 minutes when the FSK is parallelized on 16 SPEs (or a factor of up to 9.74). When compared with the original CD++/PPE, the FSK algorithms accelerate the 1024 × 1024 wildfire simulation by a factor of up to 134.34.

Figure 21 shows the scale-ups of the FSK itself. Due to SIMD vectorization and double-buffered DMA, both functions exhibit significant scalability (super-linear for `findImminents`). Function `findImminents` performs better than `findMinTime` for two reasons. Firstly, `findMinTime` uses all of the SPEs (whereas a SPE is engaged in `findImminents` only if it has found the global minimum). Secondly, `findMinTime` is invoked *in place* by the FC (whereas `findImminents` is invoked *in advance* by the NC). Overall, a scale-up of 13.4 is attained on 16 SPEs.

Table 8. 1024 × 1024 wildfire execution profile on the Power Processor Element (flat array–based simulation data management)

	Components				
Event type	Simulators	FC	NC	Bootstrap	Other overhead
(*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)	519.92	11,561.75	1.62	89.10	102.53
Total (s)			12,255.51		

FC: Flat Coordinator, NC: Node Coordinator.

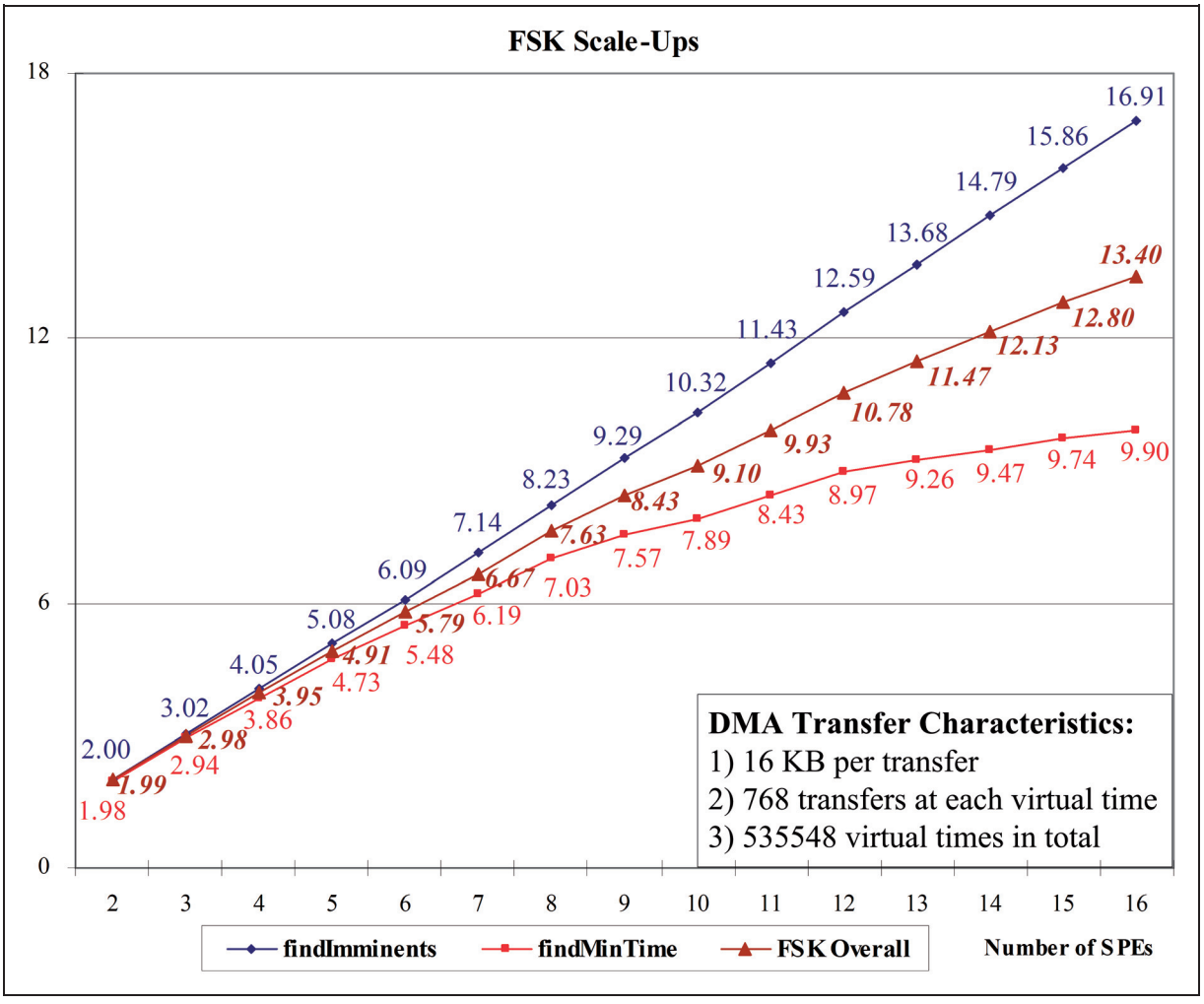
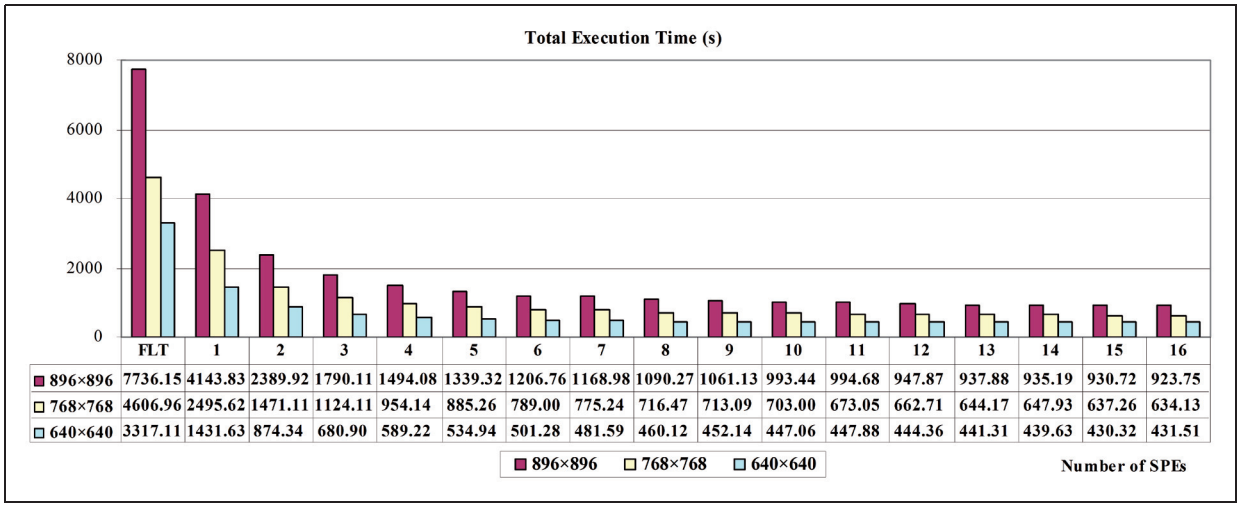


Figure 21. Flat Coordinator Synchronization Kernel scale-ups in the 1024 × 1024 wildfire simulation.[24]
DMA: direct memory access.



	FLT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
896×896	7736.15	4143.83	2389.92	1790.11	1494.08	1339.32	1206.76	1168.98	1090.27	1061.13	993.44	994.68	947.87	937.88	935.19	930.72	923.75
768×768	4606.96	2495.62	1471.11	1124.11	954.14	885.26	789.00	775.24	716.47	713.09	703.00	673.05	662.71	644.17	647.93	637.26	634.13
640×640	3317.11	1431.63	874.34	680.90	589.22	534.94	501.28	481.59	460.12	452.14	447.06	447.88	444.36	441.31	439.63	430.32	431.51

Figure 22. Total execution time in wildfire simulations of varied sizes.
SPE: Synergistic Processing Element.

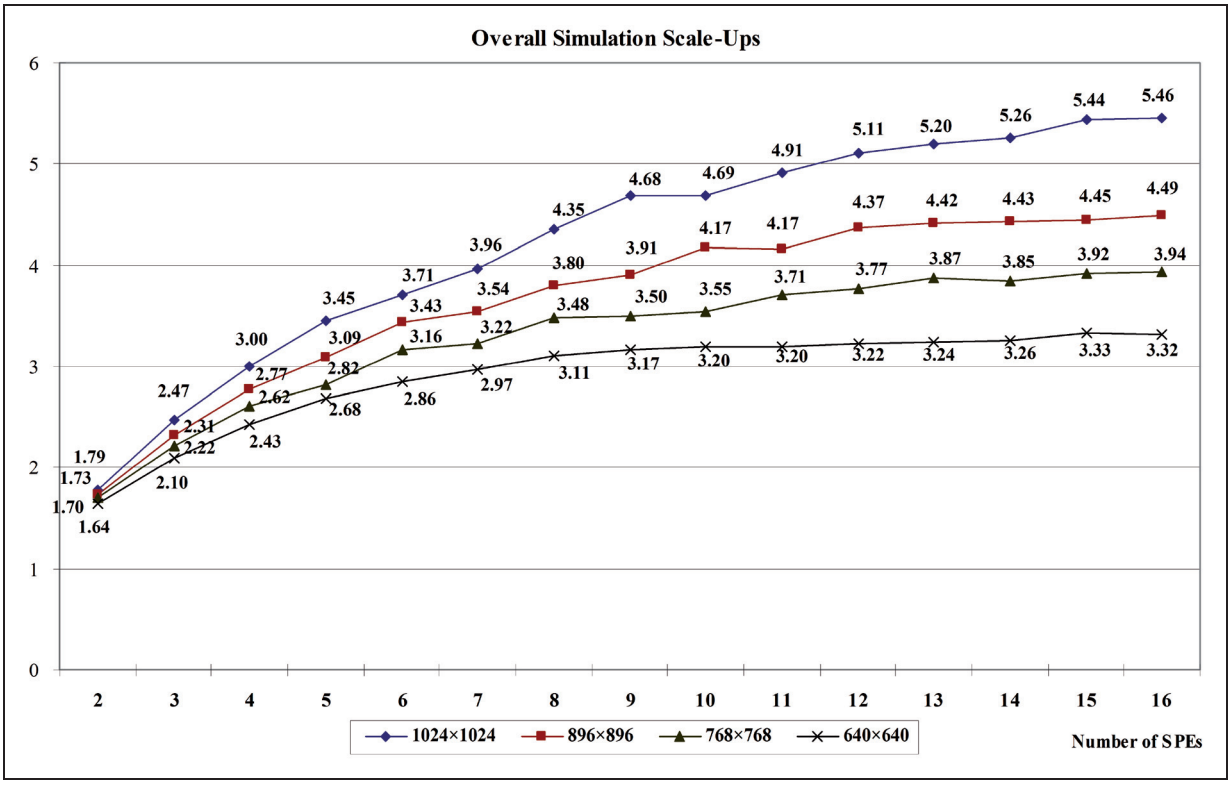


Figure 23. Overall scale-ups in wildfire simulations.

Figure 22 shows the total execution time for the wildfire simulation of varied sizes using the optimized CD++/PPE and the CD++/Cell. The parallel simulation on 16 SPEs runs 8.37, 7.27, and 7.69 times faster than the best PPE-based sequential execution (FLT) for 896×896, 768×768, and 640×640 wildfire models, respectively.

Figure 23 gives the overall simulation scale-ups achieved by CD++/Cell on 2–16 SPEs. The results indicate that the FSK algorithms can obtain better scalability in larger simulations and on greater number of SPEs. Since the FSK is a data-intensive, high-throughput kernel with relatively light computation, the performance depends primarily on the effective memory

bandwidth provided by the Cell processor. As long as the memory path is not saturated, more DMA requests from the SPEs can be handled concurrently with improved memory bandwidth utilization, resulting in higher scalability in larger simulations across multiple SPEs.

6.2. Performance of the SEK algorithms

Figure 24 shows the total execution time for the $320 \times 320 \times 2$ watershed simulation. The original and optimized sequential executions with CD++/PPE are denoted as **ORG** (Table 4) and **FLT**, respectively. The execution time with preliminary FSK optimization (**SYN**) is not shown because this optimization alone does not have a noticeable impact on the watershed simulation performance.

The flat simulation data management leads to a marginal improvement of 4.78% in the watershed simulation (FLT versus ORG), due to the following reasons. Firstly, the watershed simulation performance is dominated by the compute-intensive SEK, making it less sensitive to improved data locality. Secondly, at any virtual time, the events and states of different active Simulators are stored in different event and state buffer entries, which may not be contiguous in the main memory. Hence, while the FSK can fully benefit from the enhanced locality of Simulator timing data, the SEK may exploit increased data locality only when processing multiple events for a *single* active Simulator, as these events are packed contiguously within the same event buffer entry.

The impact of flat simulation data management is given in Table 9. Compared to the ORG execution, the Simulator and FC event execution time is reduced by 3.92% and 37.91%, respectively, thanks to improved event data locality in the main memory. The FEL operational cost is decreased by 70.12% as all *simultaneous* events are exchanged through event buffers. The bootstrap time is reduced by 16.61% due

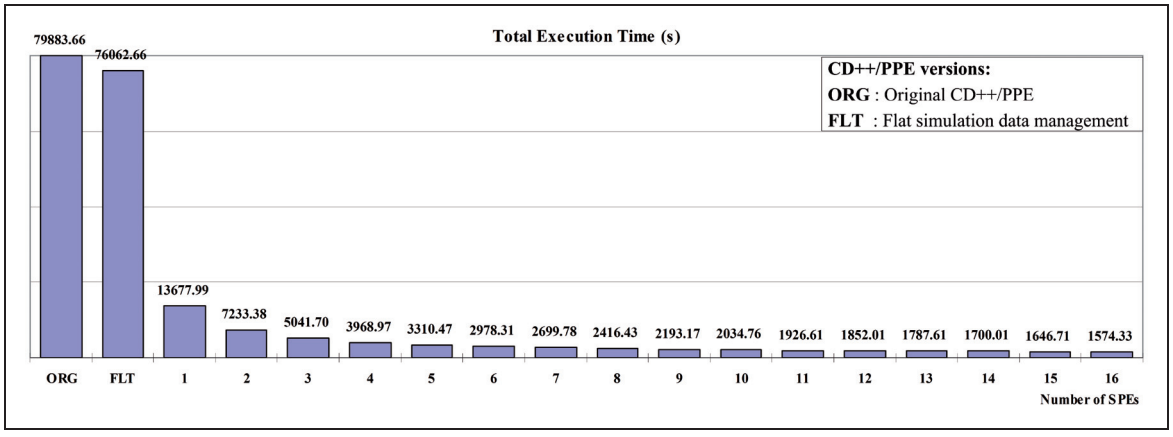


Figure 24. Total execution time in $320 \times 320 \times 2$ watershed simulation.
SPE: Synergistic Processing Element.

Table 9. $320 \times 320 \times 2$ watershed execution profile on the Power Processor Element (flat array–based simulation data management)

	Components				
Event type	Simulators	FC	NC	Bootstrap	Other overhead
(*I*)	0.36	0.03	–	–	–
(*)	75,198.80	15.42	–	–	–
(@)	32.64	37.27	–	–	–
(*X*)	–	0	–	–	–
(*Y*)	–	596.99	–	–	–
(*D*)	–	14.26	0.001	–	–
Sum (s)	75,231.80	663.97	0.001	21.03	145.86
Total (s)			76,062.66		

FC: Flat Coordinator, NC: Node Coordinator.

to efficient allocation and initialization of simulation data in flat arrays.

Using one SPE, the total execution time is reduced by a factor of 5.56 over the FLT execution for several reasons. Firstly, memory latency is reduced effectively with the multi-layered double-buffering strategy. Using the PPE with a SPE allows for pipelined event execution between the FC and the virtual Simulators, exploiting event-streaming parallelism. Likewise, the SEK is implemented in SIMD-aware code on the SPE, making it more efficient than the object-oriented scalar implementation on the PPE. Further, various low-level optimizations are used to streamline the SEK computation, such as proper address alignment for efficient data access, in-place rule evaluation, branch reduction and/or compiler-assisted branch hints, loop unrolling, and in-line substitution. Overall, the total execution time is reduced from over 21 hours (FLT) to 26 minutes when the SEK runs on 16 SPEs (or a factor of up to 48.31). When compared to the ORG execution, the SEK algorithms accelerate the $320 \times 320 \times 2$ watershed simulation by a factor of up to 50.74.

Figure 25 shows the total execution time attained in watershed simulation of varied sizes with the optimized CD++/PPE and the CD++/Cell. The parallel simulation on 16 SPEs runs 28.09, 27.69, and 29.46 times faster than the best sequential execution on PPE (FLT) for $256 \times 256 \times 2$, $192 \times 192 \times 2$, and $128 \times 128 \times 2$ watershed models, respectively.

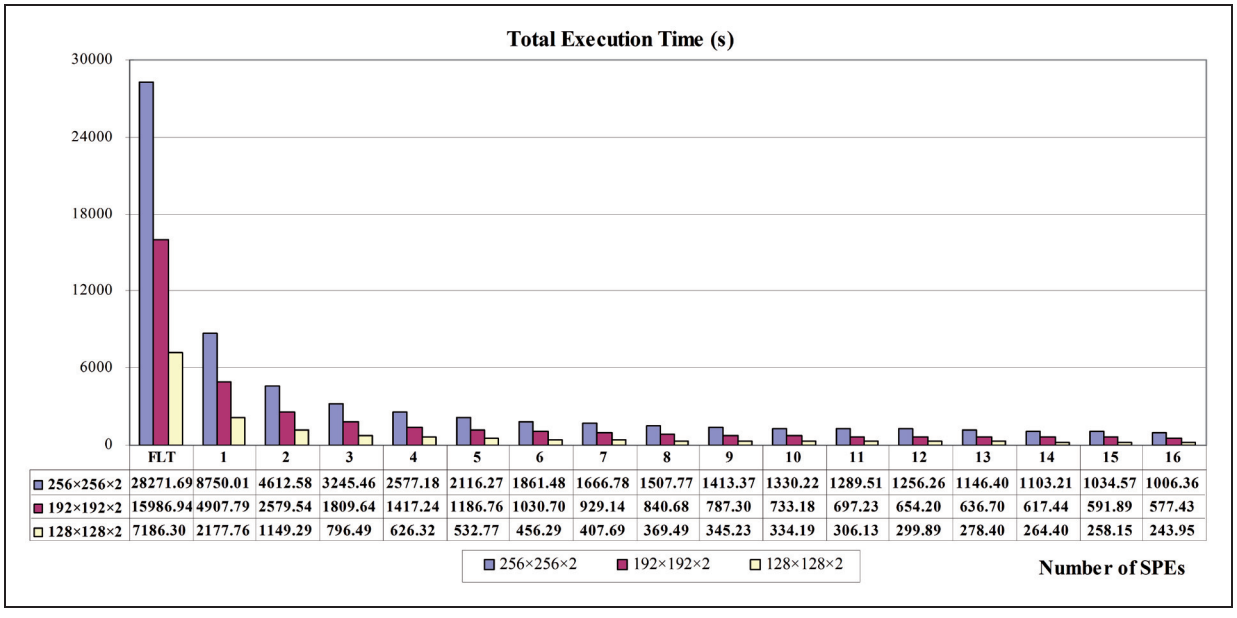
Figure 26 gives the overall scale-ups achieved by CD++/Cell. The results suggest that, regardless of the difference in the model sizes, the watershed simulation attains similar scalability across the SPEs. As the watershed simulation is dominated by a *compute-intensive* SEK, the cumulative computing power of the available SPEs is a limiting factor in the overall performance. As long as the SPEs are fully utilized, the execution time can improve at a similar rate when increasing the number of SPEs in the parallel simulation.

As seen in Figures 23 and 26, the scale-ups grow a little slower when more SPEs are used, mainly because the kernel orchestration overhead increases along with the number of SPEs. In addition, when the number of SPEs goes beyond eight, the simulation performance suffers from the increased overhead of inter-processor communication.

6.3. Implications of the MADS technique

The implications of the MADS technique are discussed as follows.

6.3.1. Additional computational kernels. The MADS technique can be extended to accommodate other types of kernels, such as random number generators.[63] For this purpose, one can reserve a group of SPEs to parallelize the kernel computation, while using the rest of the SPEs to support the FSK and/or the SEK. In an extreme case, both the FSK and the SEK can run on the PPE if the new kernel becomes the primary bottleneck. Besides the FSK and SEK orchestration tasks, the PPE helper thread can be extended to synchronize



	FLT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
256×256×2	28271.69	8750.01	4612.58	3245.46	2577.18	2116.27	1861.48	1666.78	1507.77	1413.37	1330.22	1289.51	1256.26	1146.40	1103.21	1034.57	1006.36
192×192×2	15986.94	4907.79	2579.54	1809.64	1417.24	1186.76	1030.70	929.14	840.68	787.30	733.18	697.23	654.20	636.70	617.44	591.89	577.43
128×128×2	7186.30	2177.76	1149.29	796.49	626.32	532.77	456.29	407.69	369.49	345.23	334.19	306.13	299.89	278.40	264.40	258.15	243.95

Figure 25. Total execution time in watershed simulations of varied sizes.
SPE: Synergistic Processing Element.

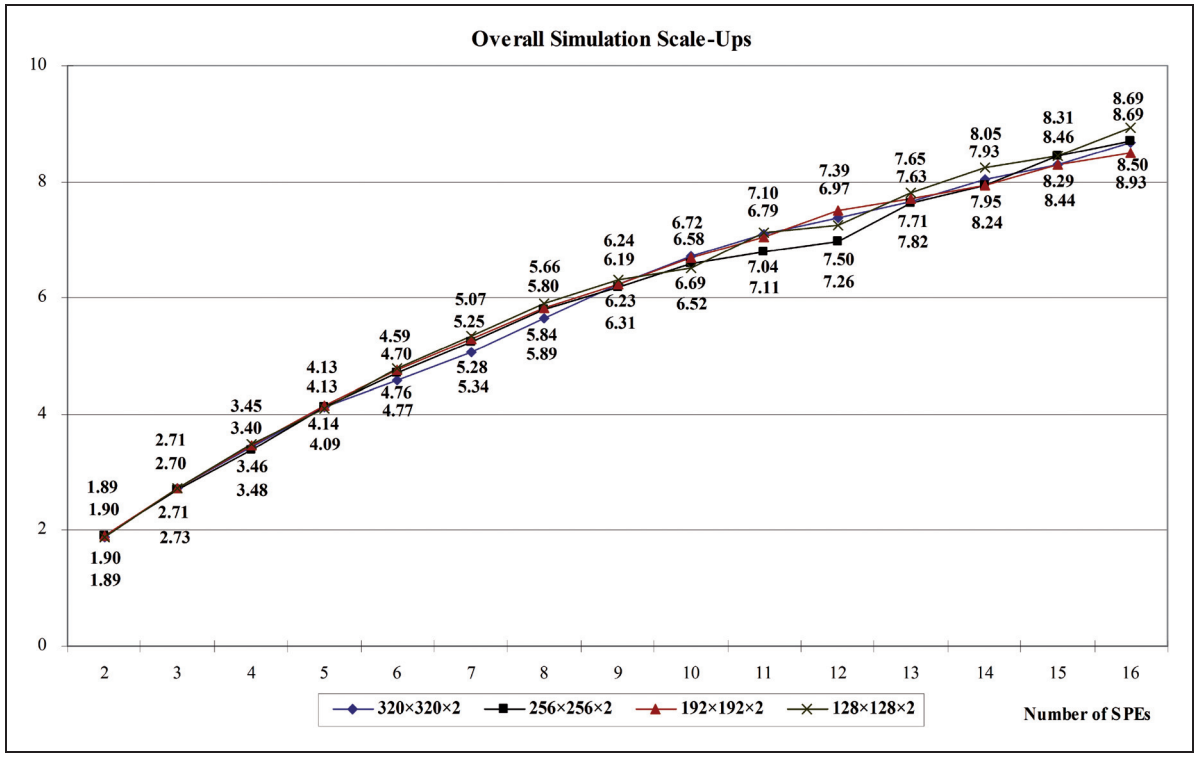


Figure 26. Overall scale-ups in watershed simulations.
SPE: Synergistic Processing Element.

the SPE threads dedicated to the new kernel. In this sense, the MADS technique allows for a modular and extensible software architecture that can be adapted to varied simulation requirements.

6.3.2. SPE-incompatible model components. As mentioned in Section 5.4, it may not always be possible to port all the Simulators to the SPEs. One example is that certain atomic models require frequent access to a legacy software library that is unsuitable to be hosted on the co-processors. A simulation thus consists of a mix of *virtual* and *concrete* Simulators, which must be scheduled properly during each phase. This issue can be solved within the MADS technique as follows. During simulation bootstrap, the concrete Simulators are allocated with greater IDs than that of the virtual Simulators, forming two distinct groups. The simulation data of both groups of Simulators are still managed in the main memory buffers, using the array indexes as SEK job IDs. For concrete Simulators, the SEK jobs are processed directly on the PPE by accessing the various buffers through hardware-controlled cache instead of software-managed DMA, taking advantage of the increased data locality in the main memory. The SEK orchestration algorithm can be enhanced so that the concrete Simulators process events (on the PPE) in parallel with virtual Simulator job execution (on SPEs) during each phase.

6.3.3. Integration with PDES techniques. The MADS technique can be integrated with existing PDES techniques to achieve efficient conservative and optimistic PDES on hybrid platforms with multicore nodes. Due to the separation between *inter-node synchronization* (the NC on the PPE) and *intra-node parallel kernel computation* (the FC and the Simulators on the PPE helper and SPE threads), the parallel simulation can be viewed, at the cluster level, as a set of NCs running asynchronously on different nodes. Hence, many existing conservative PDES algorithms, such as the null message algorithm[64] and its variants,[65–68] can be applied transparently across the NCs to determine the safe events that can be executed on a node, without interference with the kernel computation on each multicore node.

On the other hand, integrating the MADS technique with optimistic PDES algorithms, such as the Time Warp protocol[69] and its optimizations,[70–73] is a more elaborate task, mainly because the individual LPs are required to perform irregular checkpointing and rollback operations on complex data structures. This task, however, can be simplified by using the Lightweight Time Warp protocol,[74,75] which turns the Simulators into lightweight LPs, allowing them to be readily implemented as *virtual LPs*. In addition, the MADS technique can be used along with other hardware-accelerated Time Warp algorithms (e.g. Quaglia and

Santoro;[48] Santoro and Quaglia[49]) to further speed up optimistic parallel simulations.

6.3.4. Applicability to other multicore platforms.

The key method used in the MADS technique is to exploit the fine-grained data and the event parallelism inherent in DEVS-based simulations, while simultaneously combining the multi-grained parallelization options provided by the underlying hardware. Although the technique was developed on the Cell processor, the method can be generalized to PDES on other homogeneous/heterogeneous multicore platforms. Specifically, both FSK and SEK algorithms can be applied to general-purpose multicore central processing units (CPUs) that support SIMD operations (e.g. the Intel Core™ i7 processor). Although it is beneficial to use software-managed DMA for hiding memory latency, this is *not* a requirement to implement the MADS technique. It is expected that significant speedups can also be obtained on CMP architectures with hardware-controlled cache memory, owing to the enhanced data locality and the exploitation of other types of parallelism. Applying the MADS technique to special accelerator architectures, such as GPUs, is more complex, due to the constraints imposed by such processors.[76] Whereas the FSK algorithms can be ported to GPU in a relatively straightforward way (because of the pure data-parallel computing model), the SEK algorithms need to be adapted to fit the synchronous stream computing style. A hybrid scheme, such as the one proposed by Perumalla,[51] could be used to combine variable time advance in DEVS simulation with synchronous state updates at all of the Simulators during each phase, an interesting topic that is worthy of further research.

In addition, the data restructuring and optimization strategies employed in the MADS technique provide useful insight on how to reorganize a PDES program from a data-flow perspective to streamline the computation on multicore platforms in general. Furthermore, the concept of LP virtualization could be used to improve processor utilization and to achieve fine-grained dynamic load balancing on CMP architectures. Finally, the use of a descriptive language to hide multicore programming details would be of practical importance in developing PDES systems on CMP architectures in order to enhance system usability and modeler productivity at reduced M&S cost.

7. Conclusion and future work

To address the challenges of DEVS-based parallel simulation on CMP architectures, we proposed a computing technique called MADS that integrates various optimization and parallelization strategies to accelerate both memory-bound and compute-bound computational kernels, which reflect the core performance bottlenecks in the system. In addition to the exploitation of data and event parallelism inherent in the simulation, the technique combines multi-grained parallelism at different system levels to leverage the potential of the underlying hardware architecture, while hiding the technical details of multicore programming from non-expert users. The simulation data are reorganized in flat array-based buffers, improving data locality and facilitating kernel parallelization on multicore platforms. Through the concept of LP virtualization, the technique enables efficient mapping of an arbitrary number of LPs to a limited set of co-processors dynamically in a simulation, allowing for fine-grained dynamic load balancing. Moreover, the resulting software architecture is kept flexible and extensible for future development. Promising performance results have been obtained, demonstrating that the technique can achieve a significant level of scalability in a variety of workloads.

We are currently working on a number of areas related to this work, including integrating the MADS technique with PDES techniques at the cluster level, exploring new ways of leveraging multicore processors in DEVS-based parallel simulations, and applying the MADS technique to other homogeneous and heterogeneous CMP architectures.

Acknowledgment

The authors appreciate the constructive comments of Michael Perrone, Ligang Lu, Daniele Paolo Scarpazza, and Lurng-Kuo Liu from the IBM T. J. Watson Research Center.

This work was done while the corresponding author was at the Department of Systems and Computer Engineering, Carleton University, Canada.

Funding

This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC), the Ontario Graduate Scholarship program (OGS), the Mathematics of Information Technology and Complex Systems (MITACS) Accelerate Ontario Program, Canada, and by the IBM T. J. Watson Research Center, NY.

Conflict of interest statement

None declared.

References

1. Kumar R, Tullsen DM, Jouppi NP and Ranganathan P. Heterogeneous chip multiprocessors. *Computer* 2005; 38: 32–38.
2. Kumar R, Tullsen DM and Jouppi NP. Core architecture optimization for heterogeneous chip multiprocessors. In: *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Seattle, WA, USA, 2006, pp.23–32.

3. Morad TY, Weiser UC, Kolodnyt A, Valero M and Ayguade E. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Comput Archit Lett* 2006; 5: 14–17.
4. Khale JA, Day MN, Hofstee HP, Johns CR, Maeurer TR and Shippy D. Introducing to the Cell multiprocessor. *IBM J Res Dev* 2005; 49: 589–604.
5. Chen T, Raghavan R, Dale JN and Iwata E. Cell Broadband Engine architecture and its first implementation – a performance view. *IBM J Res Dev* 2007; 51: 559–572.
6. Williams S, Shalf J, Oliker L, Kamil S, Husbands P and Yelick K. The potential of the Cell processor for scientific computing. In: *Proceedings of the 3rd Conference on Computing Frontiers*, Ischia, Italy, 2006, pp.9–20.
7. Gedik B, Yu PS and Bordawekar RR. Executing stream joins on the Cell processor. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, Vienna, Austria, 2007, pp.363–374.
8. Fujimoto RM. *Parallel and distributed simulation systems*. New York: John Wiley & Sons, 2000.
9. Scarpazza DP and Braudaway GW. Workload characterization and optimization of high–performance text indexing on the Cell Broadband Engine (Cell/B.E.). In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, Austin, TX, USA, 2009, pp.13–23.
10. Eichenberger AE, O'Brien JK, O'Brien KM, Wu P, Chen T, Oden PH, et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Syst J* 2006; 45: 59–84.
11. Knight TJ, Park JY, Ren M, Houston M, Erez M, Fatahalian K, et al. Compilation for explicitly managed memory hierarchies. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, 2007, pp.226–236.
12. McCool MD. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In: *Proceedings of the GSPx Multicore Applications Conference*, Santa Clara, CA, 2006.
13. Perez JM, Bellens P, Badia RM and Labarta J. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM J Res Dev* 2007; 51: 593–604.
14. McCool MD. Scalable programming models for massively multicore processors. *Proc IEEE* 2008; 96: 816–831.
15. Varbanescu AL, Sips H, Ross KA, Liu Q, Liu LK, Natsev A, et al. An effective strategy for porting C++ application on Cell. In: *Proceedings of the 2007 International Conference on Parallel Processing*, Xi'an, China, 2007, pp.59–68.
16. Barker KJ, Davis K, Hoisie A, Kerbyson DJ, Lang M, Pakin S, et al. Entering the petaflop era: The architecture and performance of Roadrunner. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Austin, TX, 2008.
17. Perumalla KS. Parallel and distributed simulation: Traditional techniques and recent advances. In: *Proceedings of the 2006 Winter Simulation Conference*, Monterey, CA, 2006, pp.84–95.
18. Perumalla KS. Switching to high gear: Opportunities for grand-scale real-time parallel simulations. In: *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, Singapore, 2009, pp.3–10.
19. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. London: Academic Press, 2000.
20. Chow AC and Zeigler BP. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In: *Proceedings of the 1994 Winter Simulation Conference*, Lake Buena Vista, FL, 1994, pp.716–722.
21. Wainer G and Giambiasi N. *N*-dimensional Cell-DEVS models. *Discrete Event Dyn Syst* 2002; 12: 135–157.
22. Wainer G. CD++: A toolkit to develop DEVS models. *Software Pract Ex* 2002; 32: 1261–1306.
23. Liu Q and Wainer G. Accelerating large-scale DEVS-based simulation on the Cell processor. In: *Proceedings of the 2010 Symposium on Theory of Modeling and Simulation – DEVS Integrative M&S Symposium*, Orlando, FL, 2010, pp.191–198.
24. Liu Q, Wainer G, Lu L and Perrone M. Novel performance optimization of large-scale discrete-event simulation on the Cell Broadband Engine. In: *Proceedings of the 2010 International Conference on High Performance Computing & Simulation*, Caen, France, 2010, pp.108–114.
25. Liu Q and Wainer G. Exploring multi-grained parallelism in compute-intensive DEVS simulations. In: *Proceedings of the 24th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Atlanta, GA, 2010, pp.65–72.
26. Araya-Polo M, Rubio F, Cruz R, Hanzich M, Cela JM and Scarpazza DP. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Sci Program* 2009; 17: 185–198.
27. Petrini F, Fossum G, Fernandez J, Varbanescu AL, Kistler M and Perrone M. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In: *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007, pp.1–10.
28. Gschwind M. Chip multiprocessing and the Cell Broadband Engine. In: *Proceedings of the 3rd Conference on Computing Frontiers*, Ischia, Italy, 2006, pp.1–8.
29. Gschwind M. The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *Int J Parallel Program* 2007; 35: 233–262.
30. Blagojevic F, Nikolopoulos DS, Stamatakis A and Antonopoulos CD. Dynamic multigrain parallelization on the Cell Broadband Engine. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, 2007, pp.90–100.
31. Blagojevic F, Feng X, Cameron KW and Nikolopoulos DS. Modeling multigrain parallelism on heterogeneous multi-core processors: A case study of the Cell BE. In: *Proceedings of the 3rd International Conference on*

High Performance Embedded Architectures and Compilers, Goteborg, Sweden, 2008, pp.38–52.
32. Stamatakis A and Ott M. Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and OpenMP: A performance study. In: *Proceedings of the 3rd International Conference on Pattern Recognition in Bioinformatics*, Melbourne, Australia, 2008, pp.424–435.
33. Gummaraju J, Coburn J, Turner Y and Rosenblum M. Streamware: Programming general-purpose multicore processors using streams. *ACM SIGARCH Comput Archit News* 2008; 36: 297–307.
34. Kudlur M and Mahlke S. Orchestrating the execution of stream programs on multicore platforms. *ACM SIGPLAN Not* 2008; 43: 114–124.
35. IBM Corporation. 'IBM XL C/C++ for Multicore Acceleration for Linux', http://www-01.ibm.com/software/awdtools/xlcpp/multicore/ (accessed 20 Feburary 2011).
36. Ohara M, Inoue H, Sohda Y, Komatsu H and Nakatani T. MPI Microtask for programming the Cell Broadband Engine processor. *IBM Syst J* 2006; 45: 85–102.
37. Williams S, Shalf J, Oliker L, Kamil S, Husbands P and Yelick K. Scientific computing kernels on the Cell processor. *Int J Parallel Program* 2007; 35: 263–298.
38. Chellappa S, Franchetti F and Puschel M. Computer generation of fast Fourier transforms for the Cell Broadband Engine. In: *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, 2009, pp.26–35.
39. Scarpazza DP and Russell GF. High-performance regular expression scanning on the Cell/B.E. processor. In: *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, 2009, pp.14–25.
40. Saidani T, Piskorski S, Lacassagne L and Bouaziz S. Parallelization schemes for memory optimization on the Cell processor: A case study of image processing algorithm. In: *Proceedings of the 2007 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, Brasov, Romania, 2007, pp.9–16.
41. Shi G and Kindratenko V. Implementation of NAMD molecular dynamics non-bonded force-field on the Cell Broadband Engine processor. In: *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, 2008, pp.1–8.
42. Williams S, Carter J, Oliker L, Shalf J and Yelick K. Lattice Boltzmann simulation optimization on leading multicore platforms. In: *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, 2008, pp.1–14.
43. Docan C, Parashar M and Marty C. Advanced risk analytics on the Cell Broadband Engine. In: *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, 2009, pp.1–8.
44. Fujimoto RM, Tsai JJ and Gopalakrishnan GC. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Trans Comput* 1992; 41: 68–82.
45. Lynch EW and Riley GF. Hardware supported time synchronization in multi-core architectures. In: *Proceedings of the 23rd IEEE Workshop on Principles of Advanced and Distributed Simulation*, Lake Placid, NY, 2009, pp.88–94.
46. Rosu MC, Schwan K and Fujimoto RM. Supporting parallel applications on clusters of workstations: The intelligent network interface approach. In: *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, Portland, OR, 1997, pp.159–168.
47. Noronha R and Abu-Ghazaleh NB. Early cancellation: An active NIC optimization for time-warp. In: *Proceedings of the 16th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Washington, DC, 2002, pp.43–50.
48. Quaglia F and Santoro A. Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Trans Parallel Distrib Syst* 2003; 14: 593–610.
49. Santoro A and Quaglia F. Multiprogrammed non-blocking checkpoints in support of optimistic simulation on myrinet clusters. *J Syst Archit* 2007; 53: 659–676.
50. Santoro A and Fujimoto RM. Off-loading data distribution management to network processors in HLA-based distributed simulations. *IEEE Trans Parallel Distrib Syst* 2008; 19: 289–298.
51. Perumalla KS. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In: *Proceedings of the 20th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Singapore, 2006, pp.74–81.
52. Xu Z and Bagrodia R. GPU-accelerated evaluation platform for high fidelity network modeling. In: *Proceedings of the 21st IEEE Workshop on Principles of Advanced and Distributed Simulation*, San Diego, CA, 2007, pp.131–140.
53. Park H and Fishwick PA. A GPU-based application framework supporting fast discrete-event simulation. *Simulation* 2010; 86: 613–628.
54. Liu Q and Wainer G. Parallel environment for DEVS and Cell-DEVS models. *Simulation* 2007; 83: 449–471.
55. Liu Q. Algorithms for parallel simulation of large-scale DEVS and Cell-DEVS models. *PhD Dissertation*. Canada: Carleton University, Ottawa, ON, 2010.
56. Zeigler BP. DEVS today: Recent advances in discrete event-based information technology. In: *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, Orlando, FL, 2003, pp.148–161.
57. Wainer G. Applying Cell-DEVS methodology for modeling the environment. *Simulation* 2006; 82: 635–660.
58. Rothermel RC. A mathematical model for predicting fire spread in wild-land fuels. *Research Paper INT-115*. USDA Forest Service, Intermountain Forest and Range Experiment Station, Ogden, UT, 1972.
59. Zeigler BP, Moon Y, Kim D and Ball G. The DEVS environment for high-performance modeling and simulation. *IEEE Comput Sci Eng* 1997; 4: 61–71.

60. Fujimoto RM and Panesar KS. Buffer management in shared-memory time warp systems. In: *Proceedings of the 9th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Lake Placid, NY, 1995, pp.149–156.
61. Curry R, Kiddle C, Simmonds R and Unger B. Sequential performance of asynchronous conservative PDES algorithms. In: *Proceedings of the 19th IEEE Workshop on Principles of Advanced and Distributed Simulation*, Monterey, CA, 2005, pp.217–226.
62. IBM Corporation. 'Cell Broadband Engine Programming Handbook: Including the PowerXCell 8i Processor (version 1.12)', https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED (accessed 20 Feburary 2011).
63. Bader DA, Chandramowlishwaran A and Agarwal V. On the design of fast pseudo-random number generators for the Cell Broadband Engine and an application to risk analysis. In: *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, 2008, pp.520–527.
64. Chandy KM and Misra J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans Software Eng* 1979; SE-5: 440–452.
65. De Vries RC. Reducing null messages in Misra's distributed discrete event simulation method. *IEEE Trans Software Eng* 1990; 16: 82–91.
66. Wood KR and Turner SJ. A generalized carrier-null method for conservative parallel simulation. In: *Proceedings of the 8th IEEE Workshop on Parallel and Distributed Simulation*, Edinburgh, 1994, pp.50–57.
67. Bagrodia RL and Takai M. Performance evaluation of conservative algorithms in parallel simulation languages. *IEEE Trans Parallel Distrib Syst* 2000; 11: 395–411.
68. Park A, Fujimoto RM and Perumalla KS. Conservative synchronization of large-scale network simulations. In: *Proceedings of the 18th IEEE Workshop on Parallel and Distributed Simulation*, Kufstein, Austria, 2004, pp.153–161.
69. Jefferson DR. Virtual time. *ACM Trans Program Lang Syst* 1985; 7: 405–425.
70. Preiss BR, Loucks WM and Macintyre ID. Effects of the checkpoint interval on time and space in time warp. *ACM Trans Model Comput Simulat* 1994; 4: 223–253.
71. Preiss BR and Loucks WM. Memory management techniques for time warp on a distributed memory machine. In: *Proceedings of the 9th IEEE Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, 1995, pp.30–39.
72. Quaglia F. A scaled version of the elastic time algorithm. In: *Proceedings of the 15th IEEE Workshop on Parallel and Distributed Simulation*, Lake Arrowhead, CA, 2001, pp.157–164.
73. Wang J and Tropper C. Using genetic algorithms to limit the optimism in time warp. In: *Proceedings of the 2009 Winter Simulation Conference*, Austin, TX, 2009, pp.1180–1188.
74. Liu Q and Wainer G. Lightweight time warp – a novel protocol for parallel optimistic simulation of large-scale DEVS and Cell-DEVS models. In: *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Vancouver, BC, Canada, 2008, pp.131–138.
75. Liu Q and Wainer G. A performance evaluation of the lightweight time warp protocol in optimistic parallel simulation of DEVS-based environmental models. In: *Proceedings of the 23rd IEEE Workshop on Principles of Advanced and Distributed Simulation*, Lake Placid, NY, 2009, pp.27–34.
76. Owens JD, Luebke D, Govindaraju N, Harris M, Kruger J, Lefohn AE, et al. A survey of general-purpose computation on graphics hardware. *Comput Graph Forum* 2007; 26: 80–113.

Appendix: Glossary of acronyms

CMP	Chip multiprocessor
DEVS	Discrete Event System Specification
DMA	Direct memory access
EIB	Element Interconnect Bus
FC	Flat Coordinator
FEL	Future Event List
FSK	FC Synchronization Kernel
GPU	Graphical Processing Unit
IA	Imminent ID Array
LP	Logical process
LS	Local Storage
M&S	Modeling and simulation
MADS	Multicore Acceleration of DEVS Systems
MPI	Message Passing Interface
NC	Node Coordinator
PDES	Parallel Discrete-Event Simulation
P-DEVS	Parallel DEVS
PPE	Power Processor Element
SEK	Simulator Event-processing Kernel
SIMD	Single Instruction, Multiple Data
SPE	Synergistic Processing Element
TA	Time Array

Qi Liu received his BEng degree in Information Engineering from the Huazhong University of Science and Technology, China, in 1993 and his MASc and PhD degrees in Electrical and Computer Engineering from the Carleton University, Canada, in 2006 and 2010, respectively. He was a recipient of a Senate Medal for Outstanding Academic Achievement for his research work and a variety of distinguished scholarships. He served as the chair and founding member of the Ottawa Student Chapter of the Society for Modeling and Simulation International (SCS). He is

currently a postdoctoral researcher in the exascale computing group at the IBM T. J. Watson Research Center, NY, USA. His research interests include high-performance computing, multicore computing, programming models for heterogeneous architectures, advanced parallel/distributed simulation algorithms, and operations research and optimization. Further information can be found at https://researcher.ibm.com/researcher/view.-php?person=us-liuqi.

Gabriel A Wainer (SMSCS, SMIEEE) received his MSc (1993) and PhD degrees (1998, with highest honors) from the University of Buenos Aires (UBA), Argentina, and Université d'Aix-Marseille III, France, respectively. After being an assistant professor at the Computer Science Department of UBA, in July 2000 he joined the Department of Systems and Computer Engineering at Carleton University, where he is now an associate professor. He has been a visiting scholar at ACIMS (Arizona Center of Integrative Modeling and Simulation, The University of Arizona); LSIS/CNRS (Laboratory of Informatics and Systems/National Research Council), University of Nice, and INRIA (National Institute for Research in Informatics and Automatics), France. He is the author of three books and over 240 research articles; he edited four other books, and helped organizing over 110 conferences, including being one of the founders of SIMUTools and SimAUD. He is the Vice-President Publications, and was a member of the board of directors of the SCS. He is also the Chair of the Ottawa Center of The McLeod Institute of Simulation Sciences. He is a special issues editor of *Simulation*, member of the editorial board of *Wireless Networks* (Elsevier), *Journal of Defense Modeling and Simulation*, and *SIMULATION: Transactions of The Society for Modeling and Simulation International* (SAGE Publishers). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for Advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Eclipse Innovation Award, SCS Leadership Award, and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005–2006), the First Bernard P. Zeigler DEVS Modeling and Simulation Award, and the SCS Outstanding Professional Award (2011). Further information can be found at http://www.sce.carleton.ca/faculty/wainer.