A Very Fast Simulator for Exploring the Many-Core Future

Olivier Certner
ST Microelectronics & INRIA Saclay
France

Zheng Li
INRIA Saclay
Orsay, France

Arun Raman
Princeton University
New Jersey, USA

Olivier Temam
INRIA Saclay
Orsay, France

***Abstract*—Although multi-core architectures with a large number of cores (“many-cores”) are considered the future of computing systems, there are currently few practical tools to quickly explore both their design and general program scalability. In this paper, we present *SiMany*, a discrete-event-based many-core simulator able to support more than a thousand cores while being orders of magnitude faster than existing flexible approaches.**

One of the difficult challenges for a reasonably realistic many-core simulation is to model faithfully the potentially high concurrency a program can exhibit. SiMany uses a novel virtual time synchronization technique, called *spatial synchronization*, to achieve this goal in a completely *local* and *distributed* fashion, which diminishes interactions and preserves locality. Compared to previous simulators, it raises the level of *abstraction* by focusing on modeling concurrent interactions between cores, which enables fast coarse comparisons of high-level architecture design choices and parallel programs performance. Sequential pieces of code are executed *natively* for maximal speed.

We exercise the simulator with a set of dwarf-like task-based benchmarks with dynamic control flow and irregular data structures. Scalability results are validated through comparison with a cycle-level simulator up to 64 cores. They are also shown consistent with well-known benchmark characteristics. We finally demonstrate how SiMany can be used to efficiently compare the benchmarks' behavior over a wide range of architectural organizations, such as polymorphic architectures and network of clusters.

I. Introduction

In the past few years, the architecture community has been trying to replace the performance improvements once provided by regular clock frequency increases with a similar regular increase of the number of cores. Today, many company and academic roadmaps mention tens and often hundreds of cores per chip [18, 37]. The main chip manufacturers are already shipping processors with up to 8 cores and GPGPUs with tens of cores, each comprising lots of small SIMD execution units. While the addition of a small number of cores to a general-purpose CPU was proven profitable for a large class of applications, it is still unclear whether many-cores will be able to sustain the processing performance growth that is expected for the next decade. The increasing number of cores on a die creates hardware design difficulties and considerably enlarges the possible design-space, in particular for memory hierarchies and on-chip interconnects. Moreover, the performance of future many-cores will largely depend on parallel programming languages and the efficiency of their supporting run-time environments. For these reasons, researchers and engineers are in a pressing need for fast simulators to explore the design of such architectures, and even faster ones to evaluate programming models and their implementation, such as hybrid hardware/software solutions. At the same time, the increasing number of cores per chip and the rising run-time system complexity of these platforms is posing a major challenge to fast and practical simulation.

For decades, architects have been using cycle-level simulators to design microprocessors. Because cycle-level simulation of complex superscalar architectures was already considered slow enough to hinder the design process, the community has come up with a number of efficient solutions to speed up simulation for single-cores. SMARTS [39] and SimPoint [35] demonstrated that sampling can improve simulation speed by a factor of 10^3, while producing errors varying from 1% to 10%. SimFlex [38] brought significant enhancements for multi-cores running server-type workloads, where all threads operate independently, with 10^4 to 10^5 speedups. Statistical simulation and synthetic benchmarking [12, 26, 28], which recreate a similarly-behaving but much smaller trace or benchmark, are also successful alternatives for single-cores. They bring speedups up to 10^5 with an error varying from 5% to 10%.

However, there is currently no sampling nor statistical simulation technique mature enough for general parallel workloads and many-cores. Earlier proposals require to simulate an exponentially growing number of program *co-phases* [4, 20], which reduces the attainable speedups (less than 10^2 for 2 cores). Recently, Namkung et al. [25] showed that relaxing similarity constraints when clustering samples can practically mitigate this growth up to 16 cores. Perelman et al. [31] improved the speedup to 10^4 for 4 cores by clustering phases accross threads, with errors of a few percents.

Using a different approach, Penry et al. [30] have successfully leveraged modular simulation to parallelize multi-core simulators on a host CMP, achieving super-linear speedups thanks to shared caches and careful scheduling of simulation phases. But the provided speed relief over detailed single-core simulation is bounded by twice the number of available cores on a chip at design time, i.e., 8 to 16 today. More recently, Genbrugge et al. [13] applied interval analysis, which consists of a mechanistic model of superscalar processors able to produce performance numbers without having to model individual pipeline stages, to multi-core simulation. This approach currently yields a speedup of 10 over traditional cycle-accurate simulation, with a mean error rate of 5% for 8 cores.

Only a few existing simulators are able to sustain hundreds to thousands of cores with practical simulation time. The

COTSon team at HP labs proposed an approach where a trace-driven simulator is fed with thread instruction streams computed by a single-core full system simulator [24]. They present speedups relatively close to that of SiMany, but only consider an idealized architecture with a perfect memory hierarchy, i.e., without any interconnect, caches nor distribution of memory banks. Most of the other recent approaches are parallelized discrete-event simulators with varying levels of detail. They allow some events to be committed out of virtual time order, trading accuracy for speed. SlackSim [8] is a cycle-level simulator that allows individual cores to progress at different paces in a controlled manner. Notably, it proposes a *bounded slack* scheme where cores can run ahead of the current global time by at most a fixed amount of cycles. Graphite [23], a higher-level simulator, dynamically instrument executables and run sequential pieces of code natively. It explores additional *lax synchronization* schemes, and in particular a distributed one, LaxP2P, in which the progress of one core is periodically checked against another randomly chosen core.

In this paper, we propose *SiMany*, a new discrete-event simulator for many-core architectures supporting modern task-based programming models, like Cilk [5] or TBB [17]. It improves on the previous simulation approaches in three main directions. First, it introduces a novel synchronization technique, *spatial synchronization*, in order to approximately reproduce the concurrency of interactions between threads/tasks and the hardware components that the simulated program would experiment on a real many-core machine. This scheme is *distributed* and, contrary to previous ones, purely *local*. A core is allowed to make progress ahead of its topological neighbors in the interconnection network by at most a fixed time drift, which involves only nearby information. Cores can be simulated without interruption during longer phases than in schemes where they have to check their progress against a unique global window. Both these properties have the effect of improving the locality of accesses, which ultimately speeds up the simulation.

Second, SiMany pushes further the current trend of raising the level of abstraction in simulators. This trend was recently illustrated by interval simulation [13] and Graphite [23]. It is also widespread in the embedded systems domain, where systems-on-chip (SoC) typically comprise tens of off-the-shelf IP blocks. To cover the resulting large design-space, the initial stages of a SoC design process are mainly concerned with capturing the interactions between the IP blocks and the interconnects (bus or networks-on-chip), using simple, and thus potentially inaccurate, block models. Approaches of this kind have been successfully used to choose components, evaluate performance and verify general operation of SoC in early design stages [14, 21, 34]. SiMany includes simple models for caches and cores, decreasing the time required to simulate these components. It is highly configurable and can explore a wide range of designs, such as shared-memory and distributed-memory architectures and arbitrary network topologies. All operations that do not require interactions are executed natively.

Third, we evaluate the relevance of the scalability results reported by SiMany by comparing them to those obtained with a cycle-accurate simulator [2] up to 64 cores. We show that the main performance trends are successfully captured and that SiMany can be used to forecast scalability and performance variations resulting from coarse-grain architecture changes. To the best of our knowledge, this quantitative assessment is the first of its kind for an abstract simulator featuring a loose synchronization mechanism.

We show that SiMany allows to explore a large range of hardware designs and evaluate software implementations 10^3 times faster than existing flexible approaches for 1024-core architectures, while yielding comparable error rates than sampling and statistical simulation for a low number of cores. Section II presents the main time modeling and synchronization concepts. Section III details SiMany's most important implementation traits. Section IV describes the parallel programming model used to allow programs with complex control flow and data structures to take advantage of a varying and high number of cores. Section V presents the experimental methodology. Section VI first presents the relevance assessment for the results that SiMany produces. Then, it reports the performance results for different architecture classes, such as polymorphic architectures, in which cores differ in computing power, and some clustered network patterns. We also investigate the accuracy/speed trade-off related to time drift control. Section VII relates our work to the simulation field and Section VIII concludes the paper.

II. Virtual Timing

The implementation of the notion of time in an abstract discrete-event simulator must realize a delicate trade-off between accuracy and simulation speed. On one hand, an accurate simulation of hardware components involves more processing power and increases the number of synchronizations, resulting in a much longer simulation time. On the other hand, little timing information or infrequent synchronizations would make the simulator too inaccurate to obtain meaningful results for design-space exploration and software development. Moreover, the method for providing timing information must be generic and systematic enough so that an architect can vary the design parameters and still obtain reliable performance information.

Throughout the article, we call the program execution time within SiMany its *virtual time*. The first subsection below presents virtual timing and our spatial synchronization scheme. The second one highlights how SiMany correctly simulates parallel programs despite sometimes processing events out of order.

A. Principles

Timing annotations. Virtual time is computed by the simulator from the timing information that has to be provided for each instruction block. A block is a piece of code directly executed by the local CPU, without any interaction with other components. In a shared-memory program, such a block thus should not contain memory accesses, while in a distributed-memory program, it should not span a remote request. These accesses and requests are indeed timed by the simulator itself, based on the target location and the network model. The annotations can be either derived from profile runs, from a simple processor model or inserted manually. Finally, they can be computed during the execution, for example to allow to attribute approximate timings to coarse program parts at once with very low overhead or to model branch prediction in a probabilistic way. Most blocks, though, are usually statically timed.

Distributed timing. In our simulation framework, each simulated core, as well as each modeled hardware component, maintains its own private virtual time when it is active. All these virtual times would always be the same if they were permanently kept perfectly synchronized. But frequent synchronizations have a cost and are not even necessary for independent events. For this reason, our simulator updates virtual times in a purely *distributed* fashion.

As the simulator executes a code block on a core, it increases its virtual time in accordance with the timing annotations on the code block. Each memory access or remote request is initially stamped with the initiator core's virtual time and is increased by a specific delay as it traverses the architecture's communication components (e.g., CPU to network interface links, routers, wires, ...). If a request requires a reply, the reply message is dated with the request time augmented with a local processing time. When the initiator core finally starts processing the reply, its own local time is updated to that of the reply. To summarize, the sum of all delays induced by all the components traversed is added to a core's virtual time in case of interaction.

Spatial synchronization. Messages themselves are not a reliable enough form of synchronization: Two sets of cores may not communicate for a long time period, resulting in a possibly excessive time drift. On the other hand, a systematic global synchronization of all cores would be very costly. Thus, cores synchronize with their neighbors only, as specified by the interconnect/network topology, a feature we call *spatial synchronization*. On each modification of its virtual time, the core sends a virtual time update message to its immediate neighbors. The latter then update the proxy to its virtual time that they maintain. These messages are *control messages*: They have no architectural existence and they only serve to implement the simulation.

If a core's virtual time is greater than the time of its most late neighbor by a user-chosen constant T, then the core stalls its execution. As soon as this neighbor catches up with it, thus lowering the time drift under T, the core can resume its execution. Figure 1 shows an example of how an active core that is making progress (the one on the left) gradually wakes up the two cores (at its right) that were waiting for it (T is 20). The three cores and their simulation states are represented at three different points in time, from top to bottom. The number at each core's cardinal point indicates the core's view of the corresponding neighbor's virtual time.

In other words, cores are allowed to advance to different virtual times, but they are not allowed to drift from their neighbors by more than T. Note that this local bound guarantee immediately implies a global bound on the drift between any two cores that cannot exceed the diameter of the network graph (i.e., the largest topological distance between two cores) times T.

In the same way that sampling consists in trading some accuracy for speed, allowing a time drift reduces the number of synchronizations and thus speeds up the simulation, at the expense of some accuracy. Indeed, drifts open up the

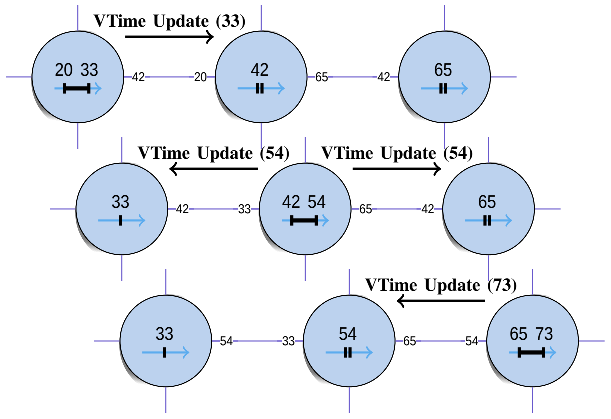

Fig. 1. Spatial synchronization example.

possibility that two messages be received and processed in the opposite order of their virtual time of reception. With $T = 20$, consider the example where two cores A and B, having times 50 and 20, want to send a message to a common neighbor core C. Having a common neighbor, A and B are allowed to drift up to 40 ($2 \times T$), and thus can be simulated concurrently at this point. If the simulator processes A, then B and finally C, A's message can reach C before B's, although its timestamp when arriving at C will be 55 whereas B's message's timestamp will be 25, assuming a common 5-cycle latency to reach a neighbor.

When the simulator processes two messages out of order, it biases their processing's completion time by the same amount of virtual time, which is at most the time needed to process the earliest message, but in opposite directions. However, we recall that a message represents a data access. A large number of them are thus actually exchanged, with out-of-order executions occurring randomly. In the end, as the experimental results will show in Section VI, these errors statistically filter out and do not engage in any visible snowball effect.

Therefore, the T simulation parameter is an accuracy/speed toggle. The smaller the value of T, the more frequent the synchronizations and context switches, the better the accuracy, but the slower the simulation.

Non-connected sets of active cores. As mentioned before, a core only propagates its own virtual time to its neighbors when it changes. Indeed, regularly sending time messages, as done in cycle-level simulators, would degrade simulation speed. However, it may happen that the set of active cores in the network is non-connected. In that case, time drift control does not spread to the whole network. Local control indeed relies on the neighbors' virtual time information but idle cores do not have a virtual time of their own and do not produce any time update messages. An example of this problem is presented in Figure 2, where advances of cores in both sets, on the right and left sides, are not propagated through the idle cores. As a result, their times drift by more than the diameter of the network times T.

A solution to overcome this problem is to have idle cores maintain a *shadow* virtual time which is the minimum virtual time of all their neighbors plus T, as if they were executing code and had advanced to the maximum virtual time allowed by the local time window before stalling. Like working cores,

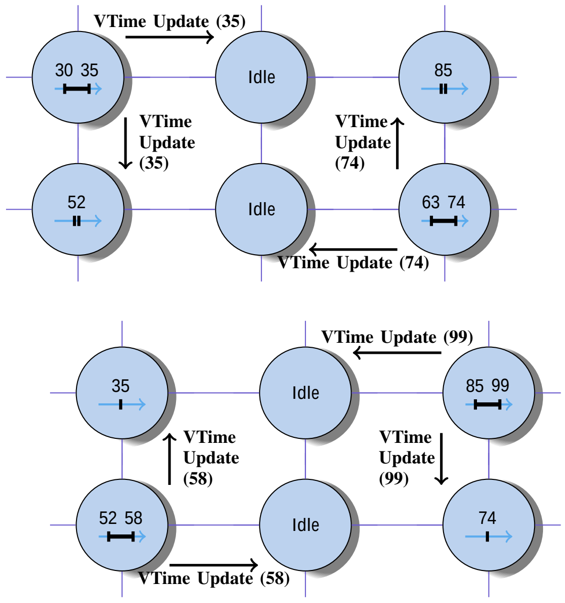

Fig. 2. The non-connected sets problem.

they only propagate their time to their neighbors when their shadow virtual time changes. Thanks to this technique, all cores within the network remain connected with respect to virtual timing, which effectively enforces the expected global time drift.

Time drift of dynamically created tasks. Many programming models (e.g., Cilk [5] or TBB [17]) provide primitives to spawn tasks dynamically. Such a task is sent to another core, as specified by the simulated run-time system and architecture, by emitting a task creation message that is stamped with the time of the initiating task, as any other message. While this message travels to its target core in the network, active cores are concurrently making progress depending on the order in which the simulator executes them and on the spatial synchronization mechanism. But, since the latter only takes into account cores, not messages, cores may well drift ahead of the not-yet-running new task's timestamp by much more than the authorized global bound.

Figure 3 shows such a situation. The left core spawns a new task at virtual time 20, through a task spawn message, but then continues to run. Since the other cores in the network are idle, the spatial synchronization mechanism does not prevent it from reaching timestamp 90, which is a lead of more than T (here, 20) over the new task's start time.

To keep this drift under control, a core tracks the birth times of the tasks it spawned and takes them into account when computing its current drift, as if the new tasks had started execution on one of its neighbors. When a spawned task arrives at its final destination, the run-time system has to send a control message to the core running the parent task to inform it that it can discard the corresponding birth date.

B. Ensuring Correct Simulation

Program execution correctness. In spite of some messages being processed out of order, program execution correctness is preserved. A well-written program's outcome is correct independently of how its threads are scheduled. In

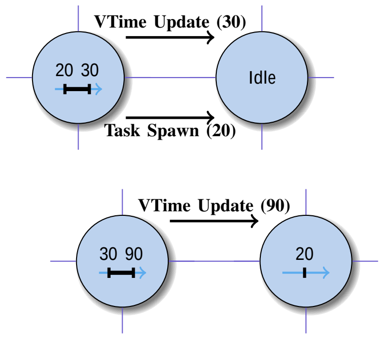

Fig. 3. Time drift of a dynamically created task.

shared-memory, in particular, output must be correct for all potential orders in which different threads can acquire a given lock. Only the lock acquisitions within the same thread have to be performed in order so as to avoid deadlocks. In distributed-memory, a program must be prepared to process incoming messages in any order that is allowed by the programming model/run-time system.

Our network implementation enforces that a core receives all messages coming from another given core in the order the latter sent them. Only messages incoming from different cores can be processed out of order. This solves the shared-memory lock order reversal problem and is compatible with traditional distributed-memory programming models that assume this property, such as MPI [10].

Locks and critical sections. A task running on a core can stall at any time because of spatial synchronization. This can cause deadlocks if this task is holding a lock or was interrupted in the middle of a critical section. Indeed, a second task may then try to acquire the same lock or enter the same critical section and will then block until the first one wakes up again and releases its resources. But if the second task is very late, i.e., if its virtual time is far behind that of the first task, spatial synchronization will prevent the first task from making enough progress to release its resources.

Figure 4 illustrates such a situation in the simple case of two neighbors. The core on the right acquired a lock at virtual time 35 and then reached time 45 at which point spatial synchronization suspended it, being ahead of the left node's time (20) by more than T (equal to 20 in this example). The left node then sends a lock request at time 22 and blocks until it receives an acknowledgement. Note that both cores involved in a deadlock need not be neighbors but can be located at distant places in the network.

Avoiding such deadlocks without the help of a global monitor can be done by temporarily allowing the core holding a lock to execute until it has released its resources. Waiving the time synchronization could potentially result in more time drift than the aforementioned global bound. However, locks or critical sections are used precisely to serialize resource access. Thus, contention essentially occurs when attempting to acquire a lock rather than in the ensuing critical section, except in the rare cases of deeply nested locking. As an example, in our dwarf benchmarks, tasks simply do not interact with others while holding a lock. In practice, the additional time drift thus has no effect on performance. We additionally devised a scheme that can handle the infrequent cases but, due to lack

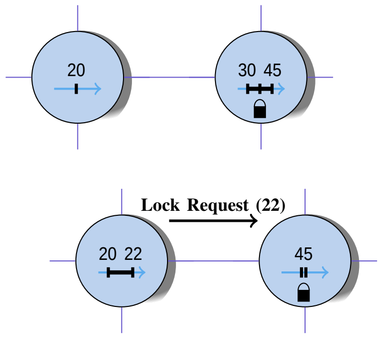

Fig. 4. Deadlock between two tasks competing for a lock.

of space and because we did not need to implement it, we will not detail it in this paper.

Deadlock avoidance. The spatial synchronization mechanism does not cause by itself any deadlock in the simulator's execution because, at any given point in time, the task with lower virtual time can always make progress, unless the system is deadlocked for another reason. Indeed, this task is either about to perform a local computation, a task spawn or a remote or shared-memory access. Among these possibilities, only an access to a locked resource may lead to the task being blocked but in this case, thanks to how locks and critical sections are handled, the other task holding the resource will run until it can release it, eventually allowing progress of the task having lower virtual time.

III. Simulator Implementation

Implementation Efficiency. Unlike most simulators, SiMany does not perform instruction set (ISA) emulation. It does not even do dynamic binary translation, as used in fast simulators like QEMU [3] or Graphite [23]. Instead, apart from network simulation and task management, it relies on *native execution*. To achieve this, it is currently necessary to modify the programs to be simulated, both to integrate the timing annotations and to explicitly use the simulator and run-time system's APIs to spawn tasks, send messages or access remote data. To ease the simulation of very large programs or to enable it for closed source ones, though, it is possible to apply these modifications automatically thanks to static binary translation or JIT compilation.

The code running on a given core is simulated in a dedicated thread. SiMany runs a program in a single system process and uses non-preemptive userland scheduling to execute the threads corresponding to cores. Thus, it does not currently exploit multiple cores or processors on the host machine. Communications between cores through the interconnect are also handled purely in userland through shared-memory lock-free circular buffers. The only system calls used are for context switching. These techniques bring two important benefits. First, we measured that using shared-memory for communications is at least 10 times faster than using traditional kernel mechanisms (pipes or sockets). Second, by bypassing kernel scheduling, SiMany avoids the overhead of the associated scheduling latency, i.e., the average delay between reception of data and wake-up of the thread going to process them. Busy-waiting would not be a viable alternative because of the system calls overhead and since the number of threads is proportional to the simulated architecture's number of cores, which exceeds by far the available physical cores on the host machine.

Architecture Variability. SiMany can be configured to explore a wide range of many-core architecture. The number of cores and their computing power are tunable, enabling the exploration of differently sized homogeneous but also heterogeneous architectures, such as polymorphic architectures. Different memory organizations are possible, from a single shared memory with uniform latency to fully distributed banks with or without hardware coherence.

Network topology is specified in a configuration file as an adjacency matrix that gives the connections between the cores. The latency and bandwith of individual links are also independently tunable. Consequently, SiMany can handle arbitrary network organizations, such as clustered or hierarchical ones. Various other fine-grain parameters, such as the size of message chunks, the time needed to process them or the routing penalty, are tunable as well.

Finally, although the simulator does not simulate a core's ISA nor its microarchitecture in detail, it is possible to reproduce some of the impact they have on performance through timing annotations. The effects of the implementation and number of functional units for a class of instructions can be mimicked by varying the timing attributed to instruction blocks that use them. In our experiments, as an example, we used a model where the instructions of a class are attributed a fixed cost, but we made the distinction between floating-point and integer additions, multiplications and divisions. We also introduced annotations to model a branch prediction scheme that succeeds at least 90% of the time and assumes a pipeline depth of 5, as will be explained in Section V.

IV. Programming Model

Rationale. Evaluating program behavior on a many-core architecture requires a careful choice of the programming model and the hardware organization that can sustain such high scalability. We consider both shared-memory and distributed-memory architecture, as described in Section V. As for the software part, task-oriented programming models, such as Cilk [5] or Intel TBB [17], are growing in popularity. Indeed, they relieve the programmer from doing the actual thread management (work dispatch), instead presenting him with a simple method to express parallelism (task spawning). Besides regular programs, they can also accommodate ones with complex control flow and data structures, thanks to their inherent load-balancing properties. As a result, they are used to parallelize a broad range of programs.

However, their efficiency is sensitive to how well the task granularity expressed by the programmer matches the features of the architecture (number of cores, their performance, communication latency and bandwidth, ...). The overhead of spawning an excessive number of fine-grain tasks decreases the benefit of parallelization, whereas too coarse-grain tasks will not be numerous enough to keep all cores busy. Moreover, only Cilk features a distributed-memory version [32], but it has a simple notion of local/remote processors and does work stealing remotely only when local sources of tasks are depleted. For these reasons, we rather used a programming model in the spirit of TBB that solves this problem through conditional spawning [29]. We paid special care to our run-time dispatching spawned tasks to neighboring cores only, avoiding any communication with far away cores. Tasks can however progressively migrate to other cores if the local ones are overloaded.

Semantics and Messages. The run-time system generates messages to drive task dispatching and, when using distributed-memory, object movements upon program requests.

When a program attempts a task spawn, it calls a special primitive, *probe*, that triggers a resource check. In order to minimize network traffic, the run-time system maintains proxies to neighbors' occupation status. Only if some proxy indicates that a neighbor is likely to have a free slot in its task queue does the run-time system actually send a reservation message (PROBE). The neighbor then responds, accepting or denying the reservation (PROBE_ACK or PROBE_NACK message). If the probe succeeds, the program then has to call the *spawn* primitive, which actually sends the neighbor the new task (TASK_SPAWN message). This neighbor finally broadcasts its new task queue's state to its own neighbors. When the probe is denied, no task is spawned and the program executes the code of the task sequentially.

Coarse synchronization is expressed thanks to task grouping. Each time a task terminates, it decrements the active tasks counter of its group. By calling the *join* primitive, a task can wait for all other active tasks in a group to finish. Its execution context is then saved until it receives a notification (JOINER_REQUEST) from the last active task in the group, which has just decremented the associated task counter to 1.

When using distributed memory, shared data are stored in objects, called *cells*, bearing similarity to C structures. Program access them by dereferencing *links*, generalized pointers that can reference cells be they stored locally or remotely. The run-time system automatically sends messages (DATA_REQUEST and DATA_RESPONSE) to retrieve remote cell content when requested and locks the cell for the access duration.

V. Experimental Methodology

Architecture Configuration. All the architectures we simulate comprise 32-bit PowerPC 405 cores with a scalar 5-stage pipeline. Each core has a private L1 cache with 1-cycle latency. The associated cache model is simple and pessimistic: Data do not stay in the cache accross function boundaries of the executed program. The instructions in the ISA are grouped by classes, including unconditional branches, conditional ones, common integer arithmetic, integer multiply, simple floating-point arithmetic and floating-point multiply and divide. All the instructions within a given class share a single time value. Branch prediction is handled specially. Where its outcome is known with certainty at compilation time (e.g., for unconditional branches and loop constructs), its effect are included in the timing annotations and a 5-cycle penalty is applied to the mispredicted branch. For the other cases, a probabilistic branch predictor with a 90% success rate is assumed.

For scalability and architecture exploration, we use two different types of architectures. The first type is a shared-memory architecture in which all cores, besides their private L1 cache, access the shared memory banks with a common low latency (10 cycles). The delays induced by cache coherence effects are not taken into account. The purpose of this optimistic architecture model is to study inherent program scalability.

The second architecture type features distributed-memory without cache coherence. In this mode, the run-time system manages shared data (see Section IV). A L2 cache with 10-cycle latency is added to each core. The base link traversal latency between two cores is set to 1 cycle and the bandwidth to 128 bytes per cycle. The purpose of this architecture type is to present results using realistic and currently common hardware parameters while anticipating short to mid-term improvements in network bandwidth. Its completely distributed design is one of the possible choices for future many-core architectures to avoid the overhead of systematic hardware cache coherence.

Cycle-Level Parameters. In order to validate the results obtained by SiMany, we compare them to those obtained on a hybrid cycle-level/system-level simulator based on the UNISIM framework [2] up to 64 cores. This simulator models architectures of the shared-memory type described above, except that cache coherence effects are fully simulated and L1 caches are split into separate instruction and data caches. To perform a fair comparison, we decided to enable the timings of cache coherence effects in SiMany during the validation, instead of deactivating cache coherence in the UNISIM-based simulator. This enables us to compare how close SiMany and other simulation approaches are to accurate simulators.

Architecture Exploration. In addition to uniform 8, 64, 256 and 1024 cores 2D meshes, we simulate our benchmarks on clustered architectures with the same number of cores but split into 4 or 8 clusters. The latency of network links inside a cluster is set to 4 times the base latency (4 cycles). The intra-cluster latency, on the other hand, is set to half a cycle. We also simulate our benchmarks on polymorphic architectures where one core out of two is twice slower than base cores and the other faster by a factor of 3/2. These latter architectures thus have exactly the same cumulated computing power as the uniform ones.

Virtual Timing Parameters. The reference value for the maximum local drift parameter T is 100 cycles and this value is used for all experiments, except when studying the effect of varying T on speed and accuracy.

The run-time system advances virtual time on its own to account for task management. Starting a task on a core has an overhead of 10 cycles in addition to the time to receive the spawn message containing the task arguments. A context switch to a joining task resuming execution costs 15 cycles. Finally, for distributed-memory architectures, access to remote data is handled by the run-time system through messages timed by the simulator. The requested data are stored in the initiating core's L2 cache, where they can be accessed with the usual 10-cycle latency.

Benchmarks. Our choice of benchmarks follows the dwarf approach's philosophy advocated by researchers at Berkeley [1], which proposes a set of kernels deemed representative of large classes that encompass current and future parallel programs. Porting a full suite comprising numerous benchmarks to a task-oriented environment and making it support both shared and distributed-memory architectures would have consumed an exorbitant amount of time. We thus decided to focus on benchmarks that include a wide range of computation and communication patterns. Most of them are notoriously difficult to parallelize because of their complex control flow and/or data structures. We argue that, in the context of the advent of many-core in general purpose computing, studying the scalability of irregular benchmarks is as relevant as that of niche scientific applications.

We present two parallel versions of `Quicksort`. The shared-memory version works on arrays and spawns a new task to handle one of the sub-arrays after each pivot step. The distributed version is an adaptation to lists, in order to avoid the transfer of whole sub-arrays to remote processing nodes. Pivot steps are distributed and they gradually construct a binary search tree. Browsing the list in order is then tantamount to traversing the constructed binary tree. For both versions, we use 50 random arrays/lists with 100,000 elements.

A graph's `Connected Components` computation is a common graph algorithm, which is for example used in image processing. Since the graph topology is not known in advance, depth-first searches are launched from lots of nodes in parallel, resulting in contention when nodes belonging to the same component are being tagged repeatedly, although the conditional spawning mitigates this issue. Parallelizing this algorithm has already been studied but solutions were proposed in the context of shared-memory machines with large structures being shared between nodes [9, 16]. We run this algorithm on 50 random graphs with 1000 nodes and 2000 edges.

We have also parallelized `Dijkstra`'s shortest paths algorithm, used for routing and navigation purposes, as described in [29]. It bears some similarity with the connected components algorithm except that already explored paths may have to be explored again when reached with a lower value of the current distance computed. On the other hand, a task encountering an already explored path close to the optimal can terminate quickly and free a core so that it can be reused for more interesting paths. Again, sophisticated variants have already been studied [11] but they require frequent global synchronizations, which we want to avoid. We run the algorithm on 50 random graphs of 2000 nodes having 3000 edges on average.

`Barnes-Hut` is the well known N-body simulation algorithm. It partitions space by building a hierarchical tree in which each internal node represents the center of mass of all the bodies in the underlying subtree. In a second phase, the force on each body B is computed by traversing the tree starting at the root. This computation is independent of that of other bodies and can be performed in parallel. The resulting communication patterns are highly irregular [15]. Only the scalability of the second phase is reported, assuming that the built tree has been broadcasted to all cores before it starts. We use 4 data sets with 128 bodies and 4 data sets with 200 bodies.

`SpMxV` is the sparse matrix-vector multiply algorithm. Matrices are specified in a row-oriented format alike to the Harwell-Boeing format. We use 30 matrices coming from a freely available sparse matrix collection [6] and 60 randomly-generated matrices of size $10^6 \times 10^6$, half of them having an average of 50 non-null coefficients per row and the other half 100 of them. For the validation experiments, only the first group of 30 matrices is used because of excessive cycle-level simulation time.

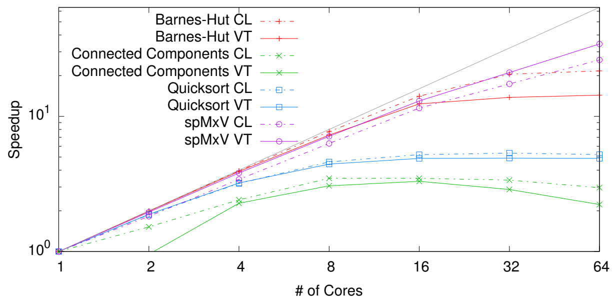

Fig. 5. Regular 2D Mesh Speedups Cycle-Level Comparison.

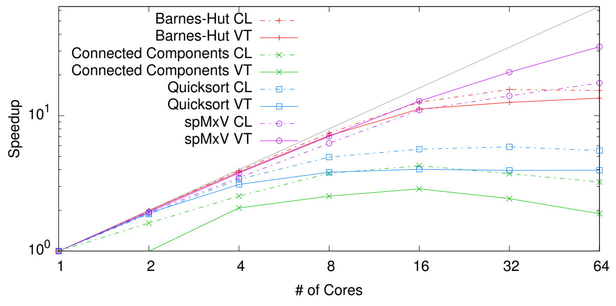

Fig. 6. Polymorphic 2D Mesh Speedups Cycle-Level Comparison.

Finally, we use a tree traversal algorithm that updates all objects within an `Octree` structure. This scenario is typically used in gaming or for graphics generation. We ran the experiments with 50 randomly generated octrees of depth 6.

VI. Evaluation and Architecture Exploration

Simulator Validation. The virtual time speedups obtained by SiMany and by the UNISIM-based simulator for shared-memory architectures with cache coherence are compared in Figure 5 for uniform 2D meshes and in Figure 6 for polymorphic ones. In the legend, CL stands for Cycle-Level and designates the results from the UNISIM-based simulator. VT stands for Virtual Time and indicates results obtained with SiMany. Both axis employ a logarithmic scale, which tends to emphazise scalability differences at the bottom of the graphs. One can see that, for every benchmark, SiMany correctly captures the speedup evolution as the number of cores increases.

SiMany's results are quantitatively close to the reference ones. The geometric mean of the errors with respect to the cycle-level simulator results for uniform meshes are 8.8% for 16 cores, 18.8% for 32 cores and 22.9% for 64 cores. The error for 16 cores is comparable to that obtained by other simulation techniques, such as interval simulation, sampling or statistical simulation on 2 to 8 cores. For 4 and 8 cores, the error is smaller but close to the 16 cores' one. These results confirm our intuition that simulating the smallest details of the microarchitecture is less important to many-core simulation than it is for single-core or dual-core simulation.

For polymorphic meshes, the errors are 22.2% for 16 cores, 30.3% for 32 cores and 33.4% for 64 cores. The higher errors for these meshes are due to slightly different imple-

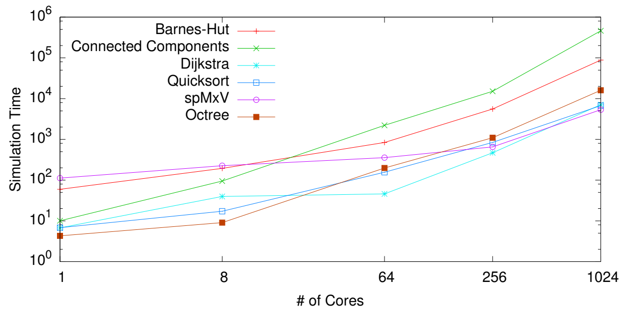

Fig. 7. Average Normalized Simulation Time.

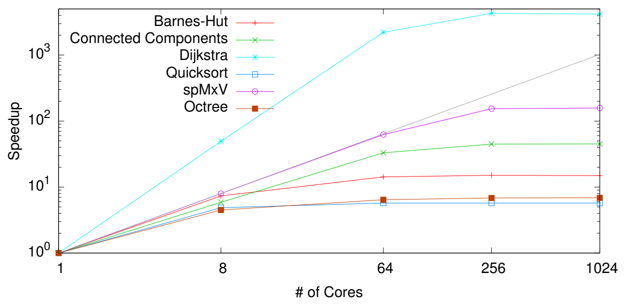

Fig. 8. Regular 2D Mesh Speedups (Shared-Memory).

mentations of the polymorphic architectures. In the UNISIM-based simulator, the L1 cache speed is the same for all cores, whereas in SiMany it is proportional to the core speed. For this reason, the cycle-level curves in Figure 6 are slightly offset upwards compared to those of Figure 5, whereas the virtual timing curves practically do not change, the polymorphic architectures having the same computing power as the uniform ones for a given number of cores.

We observe that, for both architectures, the error increases at a much slower pace than the number of cores. More importantly, the trends exhibited by the benchmarks on the reference simulator are fully reproduced in SiMany. For `Barnes-Hut`, the speedup is close to ideal until 16 cores, the point of diminishing return after which the curves start to flatten. For `Connected Components`, scalability reaches a peak at 16 cores and then decreases rapidly, because of the high contention on graph nodes as their tags are changed. `SpMxV` scales well up to 64 cores.

Simulation Speed. Figure 7 shows the overall simulation time for every benchmark in all architecture configurations, normalized to native execution on a single-core machine. The time required to simulate most of the benchmarks on 1024-core architectures is on the order of 10^4 compared to native execution.

The higher simulation time for `Barnes-Hut` and `Connected Components` are due to the distributed-memory simulations. We recall that, on these architectures, the run-time system is responsible for handling shared data. It actually implements data access as an exclusive operation, requiring data transfer to the core that needs them, whether the access is a read or a write. Algorithms that frequently access shared data will thus cause a high number of messages to be exchanged. On this respect, `Quicksort`, `SpMxV` and `Octree`, which exhibit no or little data dependencies, are much more representative of the simulator intrinsic behavior and performance. A regression shows that the average simulation time increases as a square law with a small coefficient.

SiMany is considerably faster than previous approaches. The best sampling and statistical simulation techniques so far [25, 31] can produce a 10^4 simulation time speedup for 4 cores relative to cycle-level simulation, whose best normalized simulation time is currently around 10^6, giving a net simulation time of 10^2. SiMany simulates our benchmarks on 4 cores with a normalized simulation time of 26, at the expense of some accuracy for this low number of cores. The recently proposed Graphite simulator [23] reaches a normalized simulation time of 1751 in average to simulate a 32-core architecture using 8 host cores, which, scaled to a single-core, gives a simulation time of around 14000. SiMany's corresponding normalized simulation time is 112 with only one host core being used.

Speedups on Regular 2D meshes. Figure 8 presents the benchmark virtual time speedups on our shared-memory architecture type, whereas Figure 9 presents results obtained on distributed-memory architectures (please read Section V for details).

On the optimistic shared-memory architectures, `Dijkstra` performs best and exhibits super-linear speedups (up to 4282), which is purely due to the algorithm and the datasets used. Indeed, more cores enable more parallel tasks which increases the probability to tag nodes optimally, which, in our algorithm, leads to giving up non-interesting paths quicker. `SpMxV` scales well up to 64 cores and then suddenly tops, essentially because of the size of the datasets we used. Performance of `Quicksort` may be surprisingly bad. In fact, the theoretical maximum speedup reachable by `Quicksort` is $\log_2(n)/2$ for balanced arrays of n elements. We used arrays of 100,000 elements, in which case the ideal speedup is about 8.30. In practice, we reached no more than 5.72. It is interesting to note that, for most benchmarks, going from 256 to 1024 cores does not make a significant difference and sometimes even lowers performance a bit.

On the realistic distributed-memory architectures, `Quicksort`'s and `SpMxV`'s results do not significantly change, because they cause little data movement and no contention on cells. Unsurprisingly, the performance of data-contended benchmarks, `Dijkstra` and `Connected Components`, collapses. `Connected Components`'s performance actually degrades above 8 cores, despite the run-time system's load-balancing property.

Simulation time/accuracy trade-off. We hereby study the practical effects of varying the maximum local drift parameter T. In addition to the baseline, experiments were performed with values of T of 50, 500 and 1000 for the shared-memory architecture type. As explained in Section II, increasing T means relaxing the spatial synchronization, allowing to simulate each task for a longer time window, which increases temporal and spatial locality and potentially causes more messages to be processed out of order.

Figure 10 shows the average virtual speedup variation for each benchmark as T varies, with $T = 100$ as the baseline. Only speedup values for 64 to 1024 cores are considered

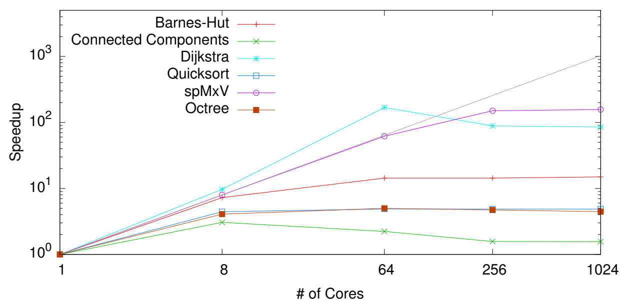

Fig. 9. Regular 2D Mesh Speedups (Distributed-Memory).

in the averages, since they are the part of interest of the benchmarks' scalability profiles. Figure 11 similarly shows the average simulation time variation for the different values of T. ± signs before numbers indicate that variations have been observed in both directions.

Lowering T to 50 increases simulation time for most benchmarks (26.7% on average), as expected. The only exception is `Octree` whose execution is slightly faster for 1024 cores (−5%). Speedup variation is only a few percents for each benchmark. Raising T to 1000, on the other hand, speeds up simulation by an average 2.38 factor (3.67 when considering only 1024-core architectures).

Figure 10 shows that only `Dijkstra` and `Connected Components`, whose algorithms are highly dependent on the tasks' timings, exhibit speedup variations of more than a few percents. They perform worse because simulation of cores is less intermixed when T is high, which decreases the probability of exploring a good path quickly and thus increases the amount of work to perform. By contrast, the performance of even relatively contended cases (`Barnes-Hut` and `Octree`) does not vary much (respectively −0.4% and −1.1% for 1024 cores). Regular benchmarks practically do not exhibit any variation. An increase of T is always beneficial for performance, but must remain limited for algorithms or applications running significantly different code paths as the relative timings of the different tasks change.

Clustered Architectures. Figure 12 presents speedup results on clustered architectures comprising 4 clusters. Data-contended benchmarks' performance varies the most. For low numbers of cores, clusters are small and the inter-cluster latency dominates. Results in this case are better on the regular meshes. This situation reverses as the number of cores grows. The average turning point for all benchmarks is around 78 cores, with however large disparities between them (from 15 for `Barnes-Hut` to 140 for `Connected Components`). Virtual execution time on 1024 cores decreases by 28.7% for `Connected Components` and by 25.6% for `Dijkstra`, whereas it practically does not change for `Quicksort` (−2.2%) and `SpMxV` (−0.1%). These results are coherent with the fact that, in the former benchmarks, tasks continuously exchange vertex data, whereas in the latter ones, tasks do not communicate much and sensitivity to network latency is low. We also did experiments with 8 clusters and obtained results featuring the same trend.

Polymorphic Architecture. Figure 13 presents results on polymorphic architectures having the same theoretical computing power than the uniform mesh comprising the same number of cores (see Section V). The `Dijkstra`'s and `SpMxV`'s performances decrease slightly. The decline is higher for the other benchmarks (−18.8% on average for 256 and 1024 cores). The run-time system, which is not particularly tuned for such architectures, has a harder time at balancing the load because the slower cores cannot spawn tasks at the same rate as faster cores.

VII. Related Work

Virtual Timing and Discrete-Event Simulators. The virtual time concept was initially introduced under the denomination of *logical clocks* by Lamport in a seminal article about the ordering of events in a distributed system [22]. Jefferson later introduced the *virtual time* term and described the Time Warp mechanism [19], a *discrete-event* distributed simulator which allows all processors in a system to be simulated mostly independently. The simulators of this kind are called *optimistic*: Messages are processed as soon as they arrive, although some other messages with comparatively lower virtual time may come afterwards. In order to enforce an execution of events in the order of their virtual time, optimistic discrete-event simulators must be able to rollback a message's execution. The other traditional approach is the *conservative* one, which was first explored by Chandy and Misra [7] at the same period. Such simulators do not simulate a processor until they can determine that it will not receive any message that could influence its next execution step.

A huge number of simulators have been implemented according to one of these two paradigms. Of the early simulators, we only mention the emblematic Wisconsin Wind Tunnel proposal [33]. It was one of the first discrete-event simulator to feature direct execution of instrumented code. WWT however used a conservative quantum-based synchronization technique tied to the CM-5 architecture. The second release of WWT improved portability using active messages, but the synchronization was still quantum-based and global. It was finally shown to be a significant performance bottleneck beyond 8 processors.

BigSim [40] is an optimistic simulator introduced in the context of the Charm++ programming model. Programs written with the latter constrain the possible message patterns and interaction orderings that can occur in the simulator, reducing dependency violations and allowing to solve the remaining ones by virtual timing adjustments instead of full rollbacks. Our approach shares with BigSim the requirement to annotate instruction blocks to compute the simulation time. However, BigSim uses a simpler network model that completely neglects contention. In contrast, we do model contention on individual links.

Sampling and Statistical Simulation. We already mentioned most of the relevant work in the introduction and defer to it for the single-core part. For multi-cores, the first approaches proposed to study the *co-phases* exhibited by multiple programs or threads running on top of SMTs and CMPs [4, 20]. Phases are already used for single-cores (e.g., by SimPoint [35]) to find a set of representative instruction intervals whose simulation will yield similar metrics than the execution of the whole program. The main problem when trying to apply this approach to multi-core simulation is

T	Barnes-Hut	Connected Components	Dijkstra	Quicksort	SpMxV	Octree
50	±3.8%	−1.6%	1.6%	0 %	±0.3%	−5.4%
500	±1.3%	−17.8%	−5.1%	0 %	0.1%	±3.1%
1000	±2.3%	−38.6%	−16.1%	0 %	±0.6%	±3.5%

Fig. 10. Average Virtual Time Speedup Variations with T (Baseline: T = 100).

T	Barnes-Hut	Connected Components	Dijkstra	Quicksort	SpMxV	Octree
50	32 %	18.9%	17.4%	13.2%	35.5%	±5.4%
500	−64.4%	−40.3%	−43.3%	−23.7%	−65.2%	−33.4%
1000	−74.4%	−56 %	−55.4%	−33.8%	−72.2%	−40.4%

Fig. 11. Average Simulation Time Variations with T (Baseline: T = 100).

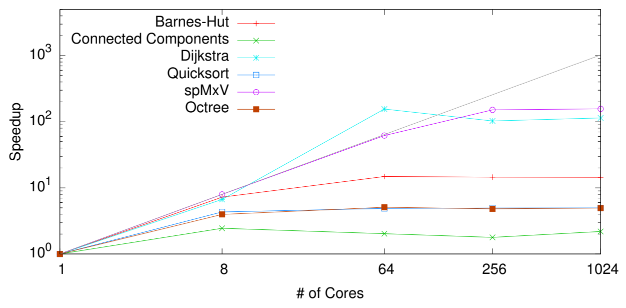

Fig. 12. Clustered 2D Mesh Speedups with 4 Clusters (Distributed-Memory).

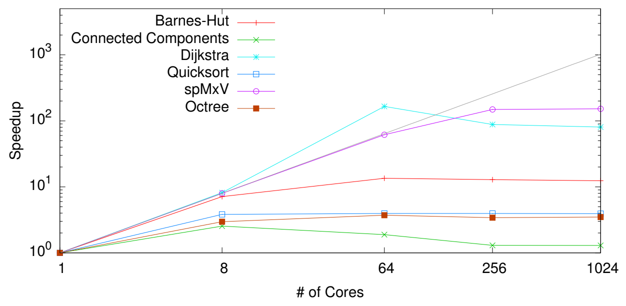

Fig. 13. Polymorphic 2D Mesh Speedups (Distributed-Memory).

that the different threads/programs may go through a large number of phase combinations (co-phases) during a single run. Experimentally, the number of co-phases grows exponentially, which does not allow to reach speedups over cycle-level simulation larger than 10^2 for 2 cores.

More recent work mitigated this problem up to 16 cores [25] by relaxing the similarity constraint used to group co-phases, while keeping the error low (under 5% on average). Perelman et al. [31] developed an alternative in which individual threads' instruction intervals are clustered into the same set of phases, instead of a separate set per thread. This approach uses the same number of overall phases (between 5 and 10) as was previously used for a single thread in experiments up to 4 cores. It remains to be seen if it can support more cores without increasing linearly the number of phases. Unfortunately, both these approaches do not fundamentally solve the exponentially growing nature of the co-phases, which does not bode well for their scalability.

Sampling and statistical simulation techniques rely on gathering some architecture-dependent metrics and statistics on the first program's run in order to accelerate subsequent simulations. They are thus constrained in the architectural variability that they can support. It is yet unclear whether they can tolerate significant variations in the number of cores or the interconnect topology. Our approach, which simulates the whole program, does not suffer from this problem.

Loose Synchronization. Several recent simulators or frameworks feature some form of loose synchronization, by which individual components can be simulated independently even in the case where this violates the constraint that events have to be processed in their virtual time order.

The SystemC TLM 2.0 standard [27] complements some of its modeling styles with *temporal decoupling*, a technique that allows a component to be simulated during a fixed amount of cycles in a row, called a *quantum*, without synchronization with other components, provided that execution stays correct. This is typically the case when a processor continues its virtual execution as long as it operates solely on its private L1 cache.

SlackSim [8] is a CMP cycle-accurate simulator derived from SimpleScalar and designed to run Pthread applications. It provides a range of synchronization mechanisms, from global barriers at every cycle to unbound slack where threads are simulated completely independently without any synchronization. Between these two extremes, one can select a WWT-like quantum-based synchronization or a bounded slack scheme, where all threads are allowed to go ahead of the current global time up to a fixed amount of cycles. This is similar to the Optimistic Time Windows mechanism [36], although for SlackSim the global time window corresponds to the allowed imprecision. This can also be seen as an extension of the temporal decoupling practice of TLM 2.0.

Graphite [23] is a simulation framework with lax synchronization. It proposes a new synchronization technique called *LaxP2P* that goes further SlackSim's ones. Cores are allowed to make progress without reference to a global time. Instead, they periodically check if they are too much ahead of another randomly chosen core by a configurable fixed slack parameter. Early cores are put to sleep for a duration computed based on the local clock advance speed observed so far. The intent is that they wake up only after the other cores have caught up.

Compared to spatial synchronization, this technique does not provide a fixed guarantee about time drift. Also, cores have to communicate with the reference core they randomly choose, which can be any other core, causing communication overhead in the whole network. Provided that local time is checked often against enough referees, LaxP2P can be stricter than spatial synchronization, because other cores than the direct neighbors are considered. This last property, however, does not translate into more accurate results, since messages between two cores necessarily pass by a chain of direct neighbors, which is exactly the topology used by spatial synchronization. With an appropriate tuning of the maximum allowed local time drift T, it is not necessary to check for drifts between remote cores, because they cannot influence each other in less than the minimum latency for a message to reach one from the other. This would mean taking T equal or less than the minimal link latency between two cores. In practice, the occurence and impact of a larger drift between remote cores is insignificant, as is highlighted by the validation and the experiments with varying T we performed.

Finally, Graphite is largely slower than our simulator. It needs 8 cores to produce speedups that are 15 times smaller, without bringing more accurate results. SiMany only requires a single core to run, making it a more practical tool for architecture exploration.

VIII. Conclusion And Future Work

In this article, we presented an abstract discrete-event simulator, SiMany, that can simulate thousands of cores with a considerable speedup (10^2 or more) over existing flexible approaches. It combines spatial synchronization, which is, to our knowledge, the first completely distributed and local synchronization approach, with abstract modeling and native execution to achieve unprecedented speed while ensuring realistic simulation. By comparing the results of SiMany with those of a cycle-level simulator up to 64 cores and by verifying that several expected behavior variations for well-known benchmarks actually occur, we showed that the main trends are successfully captured. The analysis of accuracy supports our claim that simulating the microarchitecture's innermost details of processors is less important to many-core than to single-core simulation, in an architecture exploration context. We also demonstrated how SiMany can be used to quickly explore different kinds of architectures, such as polymorphic cores or clustered networks with shared or distributed memory, and study the behavior of software on them.

An increasing number of researchers are taking the path of heterogeneous architectures to take advantage of cheap silicon. We believe that the results we obtained for the polymorphic and clustered architectures could be improved substantially with specific scheduling policies that would take into account the latency and computing power disparity among cores. Also, more data should be gathered about forecasts from discrete-event simulators starting from around a hundred of cores, the approximate threshold from which cycle-accurate simulations would take so much time that they cannot practically be conducted. These data could come from other simulator implementations or from actual many-core hardware, e.g., made from FPGAs. They would help the community quantify the accuracy of results for a high number of cores. Finally, we are convinced that the spatial synchronization scheme we presented in this paper is especially suited to parallel simulation because cores can be simulated independently within their local time window. A preliminary study indicates that, at least from networks with 64 cores, there are enough cores verifying these conditions to keep all cores of current multi-core host machines busy. Whether this property can be leveraged to improve SiMany's performance remains to be investigated.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[2] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007. ISSN 1556-6056. doi: http://dx.doi.org/10.1109/L-CA.2007.12.

[3] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS*, pages 45–56, 2004.

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995. URL citeseer.ist.psu.edu/blumofe95cilk.html.

[6] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: A web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman & Hall, 1997.

[7] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, 1979. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1979.230182.

[8] Jianwei Chen, Murali Annavaram, and Michel Dubois. Exploiting simulation slack to improve parallel simulation speed. *Parallel Processing, International Conference on*, 0:371–378, 2009. ISSN 0190-3918. doi: http://doi.ieeecomputersociety.org/10.1109/ICPP.2009.50.

[9] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358628.358650.

[10] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard.

[11] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 722–731, London, UK, 1998. Springer-Verlag. ISBN 3-540-64827-5.

[12] Lieven Eeckhout, Sebastien Nussbaum, James E. Smith, and Koen De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2003.1240210.

[13] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th International Symposium on High-Performance Computing Architecture*, 01 2010.

[14] Tony Givargis and Frank Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11):1317–1327, 2002.

[15] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 439–448, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-6605-6.

[16] Yijie Han and Robert A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990. URL `citeseer.ist.psu.edu/han90efficient.html`.

[17] Intel. Intel threading building blocks. URL: http://www.threadingbuildingblocks.org/, .

[18] Intel. Intel tera-scale computing research program. URL: http://techresearch.intel.com/articles/Tera-Scale/1421.htm, .

[19] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/3916.3988.

[20] Joshua L. Kihm and Daniel A. Connors. Statistical simulation of multithreaded architectures. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 67–74, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2458-3. doi: http://dx.doi.org/10.1109/MASCOT.2005.69.

[21] Tim Kogel, Malte Doerper, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Serge Goossens. A modular simulation framework for architectural exploration of on-chip interconnection networks. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 7–12, New York, NY, USA, 2003. ACM. ISBN 1-58113-742-7. doi: http://doi.acm.org/10.1145/944645.944648.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359545.359563.

[23] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th International Symposium on High-Performance Computing Architecture*, 01 2010.

[24] Matteo Monchiero, Jung Ho Ahn, Ayose Falcon, Daniel Ortega, and Paolo Faraboschi. How to simulate 1000 cores. Technical Report HPL-2008-190, Hewlett Packard Laboratories, November 9 2008. URL `http://www.hpl.hp.com/techreports/2008/HPL-2008-190.pdf`.

[25] Jeffrey Namkung, Dohyung Kim, Rajesh Gupta, Igor Kozintsev, Jean-Yves Bouget, and Carole Dulong. Phase guided sampling for efficient parallel application simulation. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 187–192, New York, NY, USA, 2006. ACM. ISBN 1-59593-370-0. doi: http://doi.acm.org/10.1145/1176254.1176301.

[26] Sébastien Nussbaum and James E. Smith. Modeling superscalar processors via statistical simulation. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8.

[27] Open SystemC Initiative (OSCI). TLM 2.0 Whitepaper.

[28] Mark Oskin, Frederic T. Chong, and Matthew K. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*, pages 71–82, 2000.

[29] P. Palatin, Y. Lhuillier, and O. Temam. Capsule : Hardware-assisted parallel execution of component-based programs. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006*, Orlando, Florida, december 2006.

[30] David A. Penry, Daniel Fay, David Hodgdon, Ryan Wells, Graham Schelle, David I. August, and Dan Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *HPCA*, pages 29–40, 2006.

[31] Erez Perelman, Marzia Polito, Jean yves Bouguet, John Sampson, Brad Calder, and Carole Dulong. Detecting phases in parallel applications on shared memory architectures. In *In International Parallel and Distributed Processing Symposium*, pages 25–29, 2006.

[32] Keith Harold Randall. *Cilk: Efficient multithreaded computing*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1998.

[33] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60, 1993. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/166962.166979.

[34] Ali Sayinta, Gorkem Canverdi, Marc Pauwels, Amer Alshawa, and Wim Dehaene. A mixed abstraction level co-simulation case study using SystemC for system on chip verification. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20095, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2-2.

[35] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. doi: http://doi.acm.org/10.1145/605397.605403.

[36] Lisa M. Sokol, Jon B. Weissman, and Paula A. Mutchler. MTW: An empirical performance study. In *WSC '91: Proceedings of the 23rd Winter simulation conference*, pages 557–563, Washington, DC, USA, 1991. IEEE Computer Society. ISBN 0-7803-0181-1.

[37] Tilera. Tilera website. URL: http://www.tilera.com/.

[38] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006. URL `http://dblp.uni-trier.de/db/journals/micro/micro26.html#WenischWFAFH06`.

[39]