Exploiting Fine-Grained Parallelism on Cell Processors

Ralf Hoffmann, Andreas Prell, and Thomas Rauber

Department of Computer Science
University of Bayreuth, Germany
andreas.prell@uni-bayreuth.de

Abstract. Driven by increasing specialization, multicore integration will soon enable large-scale chip multiprocessors (CMPs) with many processing cores. In order to take advantage of increasingly parallel hardware, independent tasks must be expressed at a fine level of granularity to maximize the available parallelism and thus potential speedup. However, the efficiency of this approach depends on the runtime system, which is responsible for managing and distributing the tasks. In this paper, we present a hierarchically distributed task pool for task parallel programming on Cell processors. By storing subsets of the task pool in the local memories of the Synergistic Processing Elements (SPEs), access latency and thus overheads are greatly reduced. Our experiments show that only a worker-centric runtime system that utilizes the SPEs for both task creation and execution is suitable for exploiting fine-grained parallelism.

1 Introduction

With the advent of chip multiprocessors (CMPs), parallel computing is moving into the mainstream. However, despite the proliferation of parallel hardware, writing programs that perform well on a variety of CMPs remains challenging.

Heterogeneous CMPs achieve higher degrees of efficiency than homogeneous CMPs, but are generally harder to program. The Cell Broadband Engine Architecture (CBEA) is the most prominent example [1,2]. It defines two types of cores with different instruction sets and DMA-based "memory flow control" for data movement and synchronization between main memory and software-managed local stores. As a result, exploiting the potential of CBEA-compliant processors requires significant programming effort.

Task parallel programming provides a flexible framework for programming homogeneous as well as heterogeneous CMPs. Parallelism is expressed in terms of independent tasks, which are managed and distributed by a runtime system. Thus, the programmer can concentrate on the task structure of an application, without being aware of how tasks are scheduled for execution. In practice, however, the efficiency of this approach strongly depends on the runtime system and its ability to deal with almost arbitrary workloads. Efficient execution across CMPs with different numbers and types of cores requires the programmer to maximize the available parallelism in an application. This is typically achieved

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 175–186, 2010.
© Springer-Verlag Berlin Heidelberg 2010

by exposing parallelism at a fine level of granularity. The finer the granularity, the greater the potential for parallel speedup, but also the greater the communication and synchronization overheads. In the end, it is the runtime system that determines the degree to which fine-grained parallelism can be exploited. When we speak of fine-grained parallelism, we assume task execution times on the order of 0.1–10μs.

Given the trend towards large-scale CMPs, it becomes more and more important to provide support for fine-grained parallelism. In this work, we investigate task parallel programming with fine-grained tasks on a heterogeneous CMP, using the example of the Cell processor. Both currently available incarnations—the Cell Broadband Engine (Cell/B.E.) and the PowerXCell 8i—comprise one general-purpose Power Processing Element (PPE) and eight specialized coprocessors, the Synergistic Processing Elements (SPE). Although the runtime system we present is tailored to the Cell's local store based memory hierarchy, we believe many concepts will be directly applicable to future CMPs.

In summary, we make the following contributions:

- We describe the implementation of a hierarchically distributed task pool designed to take advantage of CMPs with local memories, such as the Cell processor. Lowering the task scheduling overhead is the first step towards exploiting fine-grained parallelism.
- Our experiments indicate that efficient support for fine-grained parallelism on Cell processors requires task creation by multiple SPEs instead of by the PPE. For this reason, we provide functions to offload the process of task creation to the SPEs. Although seemingly trivial, sequential bottlenecks must be avoided in order to realize the full potential of future CMPs.

2 Distributed Task Pools

Task pools are shared data structures for storing parallel tasks of an application. As long as there are tasks available, a number of threads keep accessing the task pool to remove tasks for execution and to insert new tasks upon creation.

Task pools generalize the concept of work queueing. While a work queue usually implies an order of execution such as LIFO or FIFO, a task pool may not guarantee such an order. To improve scheduling, task pools are often based on the assumption that inserted tasks are free of dependencies and ready to run.

Implementing a task pool can be as simple as setting up a task queue that is shared among a number of threads. While such a basic implementation might suffice for small systems, frequent task pool access and increasing contention will quickly limit scalability. Distributed data structures, such as per-thread task queues, address the scalability issue of centralized implementations, at the cost of requiring additional strategies for load balancing.

2.1 Task Pool Runtime System

The task pool runtime is implemented as a library, which provides an API for managing the task pool, running SPE threads, performing task pool operations,

and synchronizing execution after parallel sections. In addition to the common approach of creating tasks in a PPE thread, we include functions to delegate task creation to a number of SPEs, intended for those cases in which the performance of the PPE presents a bottleneck to application scalability.

Each SPE executes a basic self-scheduling loop that invokes user-defined task functions after removing tasks from the task pool. In this way, the user can focus on the implementation of tasks, rather than writing complete executables for the SPEs. To support workloads with nested parallelism, new tasks can be spawned from currently running tasks, without tying execution to any particular SPE.

2.2 Design and Implementation

To match the Cell's memory hierarchy, we define a hierarchical task pool organization consisting of two separate storage domains: a *local storage domain*, providing fast access to a small set of tasks, and a *shared storage domain*, collecting all remaining tasks outside of local storage.

From an implementation viewpoint, the hierarchical task pool may be thought of as two disjoint task pools, one per storage domain, with an interface for moving tasks between the task pools. An SPE thread initiates task movement in two cases: (1) there is no local task left when trying to schedule a task for execution, or (2) there is no local storage space left when trying to insert a new task. To reduce the frequency of these events, tasks should be moved in bundles. However, choosing a large bundle size may lead to increased load balancing activity, up to the point where load balancing overheads outweigh any savings. For this reason, we only try to maximize the bundle size within fixed bounds.

Shared Task Storage. Data structures for storing tasks in main memory should meet the following requirements: (1) Allow concurrent access by multiple processing elements. (2) Provide potentially unbounded storage space that can grow and shrink as needed. (3) Facilitate movement of tasks between storage domains.

Concurrent access by multiple processing elements is usually achieved by using a distributed task pool. In our previous work, we have described the implementation of a distributed task pool based on a set of double-ended queues (deques) [3]. We noted that dequeuing a single task involved up to 11 small DMA transfers, which added significant latency to the overall operation. As a consequence of these overheads, fine-grained parallelism remained hard to exploit.

To increase the efficiency of DMA transfers, we now allocate blocks of contiguous tasks and arrange the blocks in a circular linked list. Similar in concept to distributed task queues, blocks are locked and accessed independently, allowing concurrent access to distinct blocks.

Each SPE is assigned a separate block on which to operate. Only if an SPE finds its block empty or already locked when searching for tasks, it follows the pointer to the next block, effectively attempting to steal a task from another SPE's block. Thus, load balancing is built into the data structure, rather than being implemented as part of the scheduling algorithm.

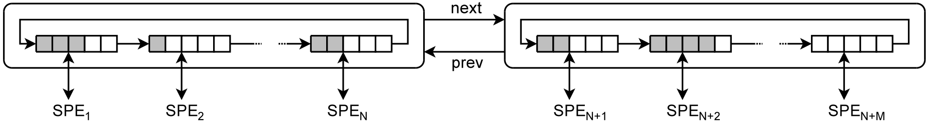


Fig. 1. Task storage in main memory. The basic data structure is a circular linked list of task blocks, each of which is locked and accessed independently. If more than one list is allocated, the lists are in turn linked together.

The resulting data structure including our extension to support multiple lists is illustrated in Fig. 1. Individual task blocks may be resized by requesting help from the PPE, using the standard PPE-assisted library call mechanism. The number of blocks in a list remains constant after allocation and equals the number of SPEs associated with that list. In the case of more than one list, as shown in the example of Fig. 1, SPEs follow the pointer to the head of the next list if they fail to get a task from their current list.

Local Task Storage. Tasks in the local storage domain are naturally distributed across the local stores of the SPEs. Given the limited size of local storage, we can only reserve space for a small set of tasks. In our current implementation, we assume a maximum of ten tasks per local store.

With DMA performance in mind, we adopt the task queue shown in Fig. 2. The queue is split into private and public segments for local-only and shared access, respectively. Access to the public segment is protected by a lock. The private segment is accessed without synchronization.

Depending on the number of tasks in the queue, the local SPE adjusts the segment boundary to share at least one task via the public segment. Adapting the segments by shifting the boundary requires exclusive access to the queue, including the public segment. If the queue is empty or there is only one task left to share, the queue is public by default (2a). When inserting new tasks, the private segment may grow up to a defined maximum size; beyond that size, tasks remain in the public segment (2b–d). Similarly, when searching for a task, the private segment is checked first before accessing the public segment (2e–f). Unless the public segment is empty, there is no need to shrink the private segment (2g). Given that other SPEs are allowed to remove tasks from a public segment, periodic checks are required to determine whether the public segment is empty, in which case the boundary is shifted left to share another task (2h–j).

Dynamic Load Balancing. Task queues with private and public segments provide the basis for load balancing within the local storage domain. If an SPE fails to return a task from both local and shared storage, it may attempt to steal a task from another SPE's local queue.

Task stealing transfers a number of tasks between two local stores without involving main memory. We have implemented a two-level stealing protocol for a system containing two Cell processors. At first, task stealing is restricted to

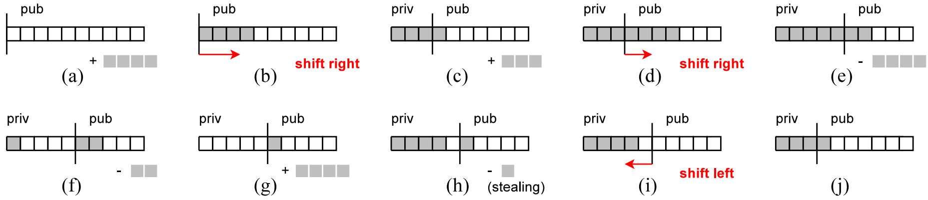


Fig. 2. Task storage in local memory. The basic data structure is a bounded queue, implemented as an array of tasks. The queue owner is responsible for partitioning the queue dynamically into private and public segments. In this example, the private segment may contain up to five tasks.

finding a local victim, i.e., a victim located on the same chip. SPE_i starts off with peeking at the public segment of its logical neighbor $\text{SPE}_{(i\%N)+1}$, where N is the number of local SPEs, and $1 \leq i \leq N$. Note that two logically adjacent SPEs need not be physically adjacent, though we ensure that they are allocated on the same chip. To reduce the frequency of stealing attempts, a thief tries to steal more than one task at a time—up to half the number of tasks in a given public segment (steal-half policy). If the public segment is already locked or there is no task left to steal, the neighbor of the current victim becomes the next victim, and the procedure is repeated until either a task is found or the thief returns to its own queue. In the latter case, task stealing continues with trying to find a remote victim, i.e., a victim located off-chip. Again, each SPE in question is checked once.

Task Pool Access Sequence. Our current task pool implementation is based on the assumption that tasks are typically executed by the SPEs. Therefore, the PPE is used to insert but not to remove tasks. In the following, we focus on the sequence in which SPEs access the task pool. For the sake of clarity, we omit the functions for adapting the local queue segments.

Get Task Figure 3(a) shows the basic sequence for removing a task from the task pool. Tasks are always taken from the local queue, but if the queue is empty, the shared storage pool must be searched for a task bundle to swap in. The size of the bundle is ultimately limited by the size of the local queue. Our strategy does not attempt to find the largest possible bundle by inspecting each block in the list, but instead, tries to minimize contention by transferring the first bundle found; even if the bundle is really a single task. Before the lock is released and other SPEs might begin to steal, one of the transferred tasks is reserved for execution by the local SPE.

Put Task Figure 3(b) shows the sequence for inserting a task into the task pool. The task is always inserted locally, but if the local queue is full, a number of tasks must be swapped out to the shared storage pool first. For reasons of locality, it makes sense not to clear the entire queue before inserting the new task. Tasks

```
void *get_task()
{
    void *task = remove_private();
    if (task) return task;
    lock(public);
    task = remove_public();
    if (task) {
        unlock(public);
        return task;
    }
    // Local store queue is empty
    swap_in_from_mm();
    task = remove();
    unlock(public);
    if (task) return task;
    // Nothing left in main memory
    return steal_task();
}
```

(a) Get task

```
void put_task(void *task)
{
    bool ins = insert_private(task);
    if (ins) return;
    lock(public);
    ins = insert_public(task);
    if (ins) {
        unlock(public);
        return;
    }
    // Local store queue is full
    swap_out_to_mm();
    insert(task);
    unlock(public);
}
```

(b) Put task

Fig. 3. Simplified sequence for accessing the task pool (SPE code)

that are moved from local to shared storage are inserted into an SPE's primary block. If the block is already full, the SPE calls back to its controlling PPE thread, which doubles the size of the block and resumes SPE execution.

3 Experimental Results

We evaluate the performance and scalability of our task pool implementation in two steps. First, we compare different task pool variants, using synthetic workloads generated by a small benchmark application. Second, we present results from a set of three applications with task parallelism: a matrix multiplication and LU decomposition and a particle simulation based on the Linked-Cell method [4]. For these applications, we compare performance with Cell Superscalar (CellSs) 2.1, which was shown to achieve good scalability for workloads with tasks in the 50μs range [5,6]. Matrix multiplication and decomposition codes are taken from the examples distributed with CellSs.

We performed runtime experiments on an IBM BladeCenter QS22 with two PowerXCell 8i processors and 8 GB of DDR2 memory per processor. The system is running Fedora 9 with Linux kernel 2.6.25-14. Programming support is provided by the IBM Cell SDK version 3.1 [7]. The speedups presented in the following subsections are based on average runtimes from ten repetitions. Task pool parameters are summarized in Table 1.

3.1 Synthetic Application

We consider two synthetic workloads with the following runtime characteristics:

- *Static*(f, n): Create n tasks of size f. All tasks are identical and require the same amount of computation.
- *Dynamic*(f, n): Create n initial tasks of size f. Depending on the task, up to two child tasks may be created. Base tasks are identical to those of the static workload.

Table 1. Distributed task pool parameters used in the evaluation

Shared storage parameters	
Initial block size	10
Max. task bundle size (into LS)	10
Max. task bundle size (out of LS)	8

Local storage parameters	
Queue size	10
Max. size of private segment	5
Task stealing policy	steal-half

Task size is a linear function of the workload parameter f such that a base task of size $f = 1$ is executed in around 500 clock cycles on an SPE.

Figure 4 shows the relative performance of selected task pools, based on executing synthetic workloads $Static(f, 10^5)$ and $Dynamic(f, 26)$. In the case of the dynamic workload, $n = 26$ initial tasks result in a total of 143 992 tasks to be created and executed. Task sizes range from very to moderately fine-grained.

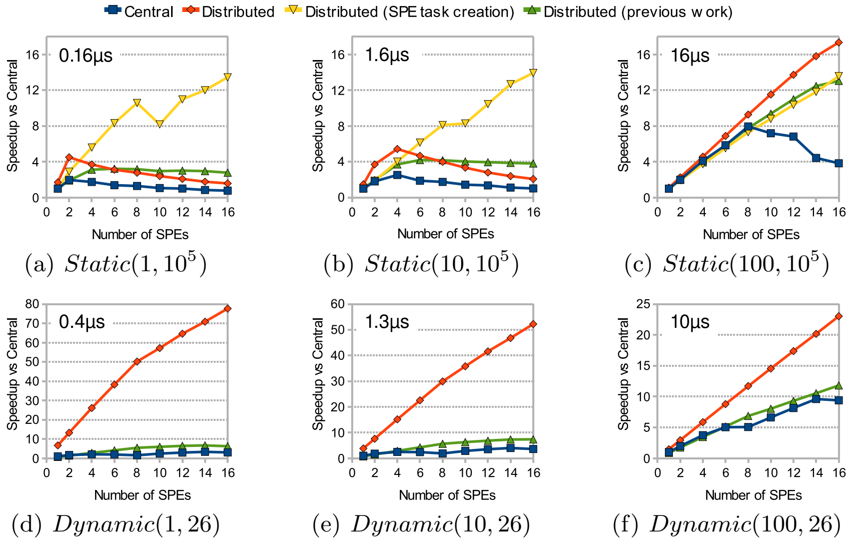
To evaluate the benefits of local task storage, we compare speedups with a central task pool that uses shared storage only. Such a task pool is simply configured without the local store queues described in the previous section. Lacking the corresponding data structures, tasks are scheduled one at a time without taking advantage of bundling. In addition, we include the results of our previous task pool implementation, reported in [3]. Speedups are calculated relative to the central task pool running with a single SPE.

Figures 4(a) and (b) show that fine-grained parallelism drastically limits the scalability of all task pools that rely on the PPE to create tasks. This is to be expected, since the PPE cannot create and insert tasks fast enough to keep all SPEs busy. The key to break this sequential bottleneck is to involve the SPEs in the task creation. Because SPEs can further take advantage of their local stores, we see significant improvements in terms of scalability, resulting in a performance advantage of 8.5× and 6.7× over the same task pool using PPE task creation. In this and the following experiments, the tasks to be created are evenly distributed among half of the SPEs, in order to overlap task creation and execution.

Without task creation support from the SPEs, our new implementation is prone to parallel slowdowns, as is apparent in Fig. 4(a) and (b). While the deques of our previous task pool allow for concurrency between enqueue and dequeue operations, the block list requires exclusive access to a given block when inserting or removing tasks. Thus, with each additional SPE accessing the block list, lock contention increases and as a result task creation performance of the PPE degrades.

Figure 4(c) shows that tasks of size $f = 100$ ($16\mu s$) are already large enough to achieve good scalability with all distributed task pools, regardless of task creation. The central task pool scales equally well up to eight SPEs, but suffers from increasing contention when using threads allocated on a second Cell processor.

Figures 4(d) and (e) show that fine-grained nested parallelism is a perfect fit for the hierarchically distributed task pool, with performance improvements of 25.5× and 14.5× over the central task pool (using 16 SPEs). Even for larger tasks, the new implementation maintains a significant advantage, as can be seen in Fig. 4(f).



(a) $Static(1, 10^5)$ (b) $Static(10, 10^5)$ (c) $Static(100, 10^5)$

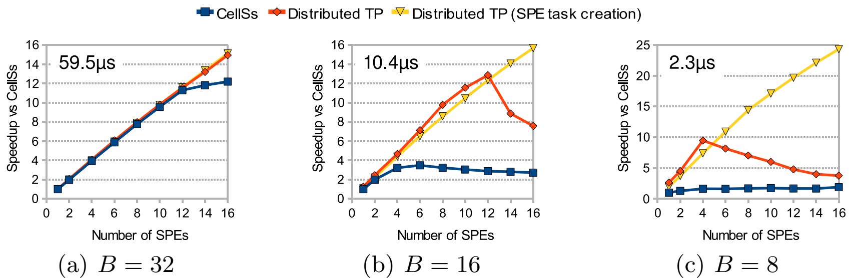
(d) $Dynamic(1, 26)$ (e) $Dynamic(10, 26)$ (f) $Dynamic(100, 26)$

Fig. 4. Speedups for the synthetic application with static and dynamic workloads (f, n), where f is the task size factor and n is the number of tasks. The resulting average task size is shown in the upper left of each figure. Speedups are calculated relative to the central task pool (shared storage only) using one SPE.

3.2 Matrix Multiplication

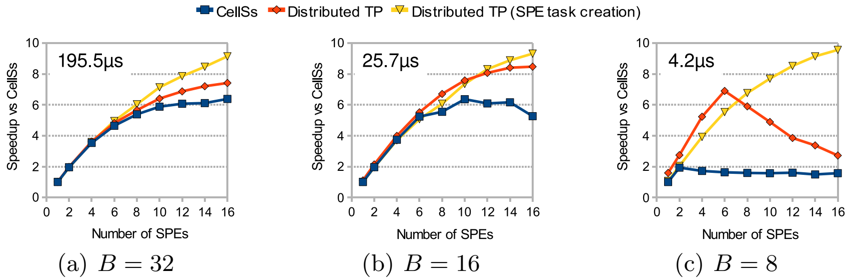
The matrix multiplication workload is characterized by a single type of task, namely the block-wise multiplication of the input matrices. Figure 5 shows the effect of decreasing task size on the scalability of the implementations. Speedups are calculated relative to CellSs using a single SPE.

CellSs scales reasonably well for blocks of size 32. However, if we further decrease the block size towards fine-grained parallelism, we clearly see the limitations of CellSs. Likewise, requiring the PPE to create all tasks drastically limits the scalability of our task pool implementation. Because the shared



(a) $B = 32$ (b) $B = 16$ (c) $B = 8$

Fig. 5. Speedups for the multiplication of two 1024×1024 matrices. The multiplication is carried out in blocks of size $B \times B$. Speedups are relative to CellSs using one SPE.



(a) $B = 32$ (b) $B = 16$ (c) $B = 8$

Fig. 6. Speedups for the LU decomposition of a 1024×1024 matrix. The decomposition is carried out in blocks of size $B \times B$. Speedups are relative to CellSs using one SPE.

storage pool is contended by both PPE and SPE threads, we face similar slowdowns as described above. For scalable execution of tasks in the low microsecond range, task creation must be offloaded from the PPE to the SPEs. Using 16 SPEs (eight SPEs for task creation), the distributed task pool achieves up to 12.8× the performance of CellSs.

3.3 LU Decomposition

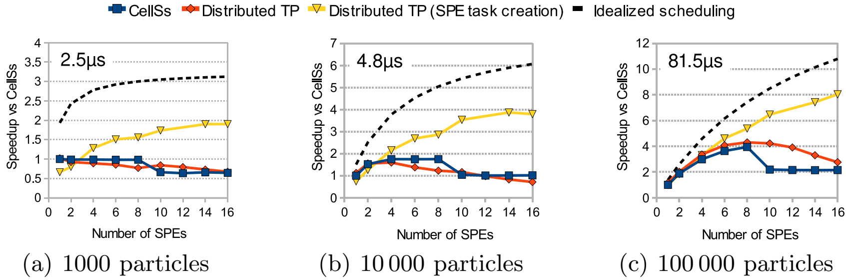
Compared to the matrix multiplication, the LU decomposition exhibits a much more complex task structure. The workload is characterized by four different types of tasks and decreasing parallelism with each iteration of the algorithm. Mapping the task graph onto the task pool requires stepwise scheduling and barrier synchronization.

Figure 6 shows speedups relative to CellSs, based on the same block sizes as in the matrix multiplication example. Due to the complex task dependencies and the sparsity of the input matrix, we cannot expect to see linear speedups up to 16 SPEs. Using a block size greater than 32 results in a small number of coarse-grained tasks. Limited parallelism, especially in the final iterations of the algorithm, motivates a decomposition at a finer granularity. In fact, the best performance is achieved when using blocks of size 32 or 16. Once again, fine-grained parallelism can only be exploited by freeing the PPE from the burden of task creation. In that case, we see an improvement of up to 6× over CellSs.

3.4 Linked-Cell Particle Simulation

The particle simulation code is based on the Linked-Cell method for approximating short-range pair potentials, such as the Lennard-Jones potential in molecular dynamics [4]. To save computational time, the Lennard-Jones potential is truncated at a cutoff distance r_c, whose value is used to subdivide the simulation space into cells of equal length. Force calculations can then be limited to include interactions with particles in the same and in neighboring cells.

For our runtime tests, we use a 2D simulation box with reflective boundary conditions. The box is subdivided into 40 000 cells, and particles are randomly



(a) 1000 particles (b) 10 000 particles (c) 100 000 particles

Fig. 7. Speedups for the Linked-Cell application with three different workloads based on the number of particles to simulate. Idealized scheduling assumes zero overhead for task management, scheduling, and load balancing. Speedups are relative to CellSs using one SPE.

distributed, with the added constraint of forming a larger cluster. The workload consists of two types of tasks—force calculation and time integration—which update the particles of a given cell. At the end of each time step, particles that have crossed cell boundaries are copied to their new enclosing cells. This task is performed sequentially by the PPE.

We focus on the execution of three workloads, based on the number of particles to simulate: 1000, 10 000, and 100 000. The potential for parallel speedup increases with the number of particles, but at the same time, clustering of particles leads to increased task size variability. Thus, efficient execution depends on load balancing at runtime.

Figure 7 shows the results of running the simulation for ten time steps. Once again, speedups are calculated relative to CellSs using a single SPE. Maximum theoretical speedups are indicated by the dashed line, representing parallel execution of force calculation and time integration without any task related overheads.

In the case of only 1000 particles, there is little parallelism to exploit. Although SPE task creation leads to some improvement, 16 SPEs end up delivering roughly the same performance as a single SPE executing the workload in sequence. In this regard, it is important to note that tasks are only created for non-empty cells, requiring that each cell be checked for that condition. Whereas the PPE can simply load a value from memory, an SPE has to issue a DMA command. Even with atomic DMA operations, the additional overhead of reading a cell's content is on the order of a few hundred clock cycles.

The amount of useful parallelism increases with the number of particles to simulate. Using 16 SPEs, the task pool delivers 62% and 75% of the ideal performance, when simulating 10 000 and 100 000 particles, respectively. In contrast, CellSs achieves only 16% and 20% of the theoretical peak.

4 Related Work

Efficient support for fine-grained parallelism requires runtime systems with low overheads. Besides bundling tasks in the process of scheduling, runtime systems

may decide to increase the granularity of tasks depending on the current load and available parallelism. Strategies such as lazy task creation [8] and task cut-off [9] limit the number of tasks and the associated overhead of task creation, while still preserving enough parallelism for efficient execution. However, not all applications lend themselves to combining tasks dynamically at runtime.

In a recent scalability analysis of CellSs, Rico et al. compare the speedups for a number of applications with different task sizes [10]. The authors conclude that, in some of their applications, the task creation overhead of the PPE limits scalability to less than 16 SPEs. To reduce this overhead and thereby improve scalability, the authors suggest a number of architectural enhancements to the PPE, such as out-of-order execution, wider instruction issue, and larger caches. All in all, task creation could be accelerated by up to 50%. In the presence of fine-grained parallelism, however, task creation must be lightweight *and* scalable, which we believe is best achieved by creating tasks in parallel.

Regardless of optimizations, runtime systems introduce additional overhead, which limits their applicability to tasks of a certain minimum size. Kumar et al.make a case for exploiting fine-grained parallelism on future CMPs and propose hardware support for accelerating task queue operations [11,12]. Their proposed design, Carbon, adds a set of hardware task queues, which implement a scheduling policy based on work stealing, and per-core task prefetchers to hide the latency of accessing the task queues. On a set of benchmark applications with fine-grained parallelism from the field of Recognition, Mining, and Synthesis (RMS), the authors report up to 109% performance improvement over optimized software implementations. While hardware support for task scheduling and load balancing has great potential for exploiting large-scale CMPs, limited flexibility in terms of scheduling policies will not obviate the need for sophisticated software implementations.

5 Conclusions

In this paper, we have presented a hierarchically distributed task pool that provides a basis for efficient task parallel programming on Cell processors. The task pool is divided into local and shared storage domains, which are mapped to the Cell's local stores and main memory. Task movement between the storage domains is facilitated by DMA-based operations. Although we make extensive use of Cell-specific communication and synchronization constructs, the concepts are general enough to be of interest for other platforms.

Based on our experiments, we conclude that scalable execution on the Cell processor requires SPE-centric runtime system support. In particular, the current practice of relying on the PPE to create tasks is not suitable for exploiting fine-grained parallelism. Instead, tasks should be created by a number of SPEs, in order to maximize the available parallelism at any given time.

Acknowledgments

We thank the Forschungszentrum Jülich for providing access to their Cell blades. This work is supported by the Deutsche Forschungsgemeinschaft (DFG).

References

1. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM J. Res. Dev. 49(4/5) (2005)
2. Johns, C.R., Brokenshire, D.A.: Introduction to the Cell Broadband Engine Architecture. IBM J. Res. Dev. 51(5) (2007)
3. Hoffmann, R., Prell, A., Rauber, T.: Dynamic Task Scheduling and Load Balancing on Cell Processors. In: Proc. of the 18th Euromicro Intl. Conference on Parallel, Distributed and Network-Based Processing (2010)
4. Griebel, M., Knapek, S., Zumbusch, G.: Numerical Simulation in Molecular Dynamics, 1st edn. Springer, Heidelberg (September 2007)
5. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a Programming Model for the Cell BE Architecture. In: Proc. of the 2006 ACM/IEEE conference on Supercomputing (2006)
6. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM J. Res. Dev. 51(5) (2007)
7. IBM: IBM Software Development Kit (SDK) for Multicore Acceleration Version 3.1, http://www.ibm.com/developerworks/power/cell
8. Mohr, E., Kranz, D.A., Halstead Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In: Proc. of the 1990 ACM conference on LISP and functional programming (1990)
9. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: Proc. of the 2008 ACM/IEEE conference on Supercomputing (2008)
10. Rico, A., Ramirez, A., Valero, M.: Available task-level parallelism on the Cell BE. Scientific Programming 17, 59–76 (2009)
11. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In: Proc. of the 34th Intl. Symposium on Computer Architecture (2007)
12. Kumar, S., Hughes, C.J., Nguyen, A.: Architectural Support for Fine-Grained Parallelism on Multi-core Architectures. Intel Technology Journal 11(3) (2007)