Adaptive Work-Splitting: Efficient Loop Scheduling for Task Parallel Programming

Ralf Hoffmann, Andreas Prell*, and Thomas Rauber
Department of Computer Science, University of Bayreuth, Germany

January 20, 2011

Abstract

As task-centric programming models become more widely used, we are seeing an increasing number of applications that take advantage of fine-grained parallelism. Programmers are encouraged to expose as much parallelism as possible and let the compiler and runtime system decide what parallelism is useful for a given target architecture. In the case of loop parallelism, it is natural to declare independent iterations as parallel, but doing so calls for a runtime scheduler that is capable of combining iterations and reducing loop scheduling overheads.

In this paper, we build on the idea of using Lazy Binary Splitting (LBS) for scheduling arbitrary loop parallelism in task parallel runtime systems. LBS-based loop scheduling integrates well with existing work-stealing schedulers and eliminates the need for chunking to find a good trade-off between load balance and runtime overheads. We extend LBS with different work-splitting strategies, including guided, adaptive, and distributed splitting, for improved performance on a wide range of parallel loops. Ease of implementation and efficient use of fine-grained parallelism make a case for including work-splitting techniques in future runtime systems.

1 Introduction

The growing importance of parallel programming in mainstream computing is reflected in the number of programming models that seek to balance programmer productivity and application performance. The shift from thread-centric to task-centric execution models brings about a clear separation of responsibilities: the

[1]Email: andreas.prell@uni-bayreuth.de

programmer specifies which computations can be done in parallel, while an underlying runtime system tries to get the most out of this parallelism. In this way, the programmer can focus on the structure of his or her application and let the runtime system perform the actual work of mapping parallel tasks to available processing resources.

In theory, the best performance results from exposing as much parallelism as possible so that the runtime system can pick the parallelism that is useful for a given target architecture. In practice, however, large amounts of parallelism often lead to increased overheads.

Loops with independent iterations are an important source of parallelism in many applications. Task parallel runtime systems may implement parallel loops in terms of sequential loops with parallel function calls, as in:

```
for (i = 0; i < n; i++)
    spawn f(i);
```

Here, **spawn** `f(i)` creates a task for executing iteration `i` of the loop. As a result, this code spawns a total of `n` tasks, corresponding to the `n` iterations of the loop. Two examples illustrate why spawning a task for each iteration might compromise performance and scalability:

- If the amount of work per iteration is small (fine-grained parallelism), the runtime system might not be able to create enough parallelism to keep all workers busy. Limited parallelism also limits scalability.

- If the amount of work per iteration is large enough to not affect scalability, the runtime system might end up creating much more parallelism than can actually be exploited on a given architecture. Too much parallelism means unnecessary runtime overheads, which might hurt performance in addition to consuming large amounts of heap or stack space to store the tasks.

The solution to both problems is to employ a runtime scheduler that creates and schedules the "right amount of parallelism". Such a scheduler must (1) keep parallelism at a level that prevents workers from becoming idle and (2) avoid excessive parallelism that would go unused by adapting the granularity of parallel work at runtime. Lazy Binary Splitting (LBS) is a scheduling algorithm that attempts to do that for loop parallelism [25]. We call LBS a *work-splitting* algorithm because it first combines parallel work and then splits it as needed to maintain load balance.

Central to the idea of work-splitting is a special type of task for representing loop parallelism. We call such a task that refers to a certain iteration range a *loop task*. Thus, whenever a worker encounters a parallel loop, it creates only one loop task instead of one task per iteration. Loop tasks can then be split into sub-tasks so that every idle worker can get some iterations to work on.

LBS-based loop scheduling uses combined work-splitting and work-stealing to distribute the iterations of a parallel loop among worker threads. Because many parallel runtime systems are already based on work-stealing schedulers [16, 17, 22], all we need is additional support for work-splitting. This support is what we focus on in this paper. In summary:

- We make a case for supporting loop tasks in task parallel runtime systems (Section 2). Loop tasks are the primitives upon which parallel loops can be implemented efficiently.
- We propose and implement strategies for scheduling parallel loops by work-splitting (Section 3). We start off with Lazy Binary Splitting (LBS) and develop several extensions to improve load balancing. Because the algorithms are based on LBS, they benefit from the ability to adapt to runtime conditions.
- We evaluate the performance of the work-splitting strategies on a set of parallel loops with balanced and unbalanced iterations, ranging from fine-grained to coarse-grained parallelism (Section 4). For reference, we compare performance to a non-work-splitting scheduler that spawns a task for each iteration, but utilizes more than one worker to do so in parallel. Based on our experiments on a system with two Cell processors, we conclude that a combination of static scheduling and work-splitting is the best strategy for getting good performance across a wide range of parallel loops.

2 The Case for Loop Tasks

Bundling tasks in the process of scheduling is a common optimization technique to reduce scheduling overheads. Examples include the assignment of multiple tasks to a single worker, or stealing more than one task from another worker's queue (steal-half). The tasks themselves need not be related and are only grouped together to facilitate scheduling. Thus, a bundle is essentially a set of tasks, with each task referencing its own code and data.

Structured parallelism in the form of parallel loops can be handled in the same way. Every independent iteration is wrapped up in a separate task and bundling takes care of moving multiple iterations between workers. While this approach may be easy to implement in any task parallel runtime, it has a serious drawback: the parallelism available at runtime is determined by the cost of task creation. If the cost of creating and scheduling a task is small compared to the cost of executing iterations, the scheduler may be able to create enough parallelism for achieving good speedups. However, if task management constitutes a significant

portion of the loop's actual work, only few tasks are available at any given time. Limited parallelism directly translates into limited scalability. In the worst case, only one task may be available, at which point even bundling cannot help to reduce scheduling overheads.

To avoid the task creation bottleneck, schedulers need to have special support for parallel loops. One approach that tries to circumvent the problem is to distribute a loop to a number of workers first to start creating tasks in parallel. In this way, the number of available tasks can be increased and scalability can be improved. But the requirement to create a task for every iteration will make it difficult to get good performance on fine-grained parallelism. We clearly need a better way of expressing loop parallelism; a way that a scheduler can readily and efficiently take advantage of.

Because iterations share code and data environment, we can use a single task to represent a number of iterations. We call such a task T that refers to a certain iteration range $[i, j)$ a *loop task*, denoted by $T(i, j)$. Besides having additional fields for the iteration range, a loop task has the same structure as a non-loop task, and thus, creating a loop task is not much different from creating any other task. This has an important implication for scheduling, which makes a strong case for supporting loop tasks: when a worker encounters a parallel loop, it needs to create only one task instead of one task per iteration. In this way, the iterations of the loop are effectively bundled within a single task. Breaking down the bundle into smaller bundles for load balancing is straightforward; all the scheduler has to do is create a new loop task and split the iteration range into sub-ranges.

This form of bundling is much more efficient than bundling at the level of individual tasks because it allows to execute iterations without extra task creation or scheduling. Combining many small iterations into one larger task helps to control the granularity of parallelism and can drastically reduce runtime overheads.

In addition, a scheduler may choose to bundle loop and non-loop tasks in applications that exhibit both structured and unstructured parallelism. In fact, the scheduler does not even need to know the type of task until it returns the task for execution. At this point, the scheduler must check whether the task can be executed right away (if it is a non-loop task or a loop task with a single iteration), or whether it might be useful to create parallelism by splitting the loop task.

3 Work-splitting Extensions

Upon returning a loop task, the scheduler must decide whether additional parallelism is needed for load balancing. We consider the code in Figure 1 as the basis for work-splitting.

```
SPLITANDEXECUTE(L)
1  Q = GETLOCALTASKQUEUE()
2  if Q is empty
3       L' = SPLIT(L)
4       push L' onto Q
5  for i ∈ [L.start, L.end)
6       Execute iteration i
7       if Q is empty and L.iterations > 1
8            L' = SPLIT(L)
9            push L' onto Q
```

Figure 1: Pseudocode for the work-splitting subroutine of the scheduler. SPLITANDEXECUTE(L) splits a loop task L if additional parallelism is needed and starts executing the remaining iterations of L. SPLIT(L) creates a new loop task L' by dividing the iteration range $[L.start, L.end)$ of L into a lower range $[L.start, split)$ and an upper range $[split, L.end)$. The value of $split$ depends on the actual splitting strategy.

3.1 Lazy Binary Splitting

The subroutine SPLITANDEXECUTE implements Lazy Binary Splitting (LBS), an algorithm for deciding when to split a loop task to create parallelism [25]. It seems reasonable to assume that more parallelism will be useful when workers become underutilized and spend much of their time waiting for work. But how do we know when that happens? LBS addresses this question with a simple heuristic: if a task queue is empty, it is likely that other workers have already stolen from that queue, or will fail to steal shortly. In both cases, it is beneficial to create tasks and make them available for load balancing. If each worker makes sure to share at least one task in this way, workers that run out of work will have enough freedom in finding a queue to steal from. Thus, if a worker schedules a loop task and subsequently finds its queue empty, it splits off and enqueues one half of its iterations (split-half). Splitting is the basis for creating parallelism to keep workers busy.

As long as the work is balanced and splitting is not required (task queue is not empty), iterations are executed without paying the overhead of task creation and scheduling. Being able to serialize iterations is important to limit parallelism to a level that is useful without incurring too much overhead.

To combine very short iterations, LBS introduces a profitable parallelism threshold (*ppt*), which defines a minimum chunk size for execution. A suitable value for

ppt depends on the work per iteration and the overhead of creating a new task. For our purposes, we assume *ppt* = 1 throughout the paper. Thus, the queue check is repeated after completing every iteration.

3.2 Guided Splitting

The performance of LBS depends on the availability of fine-grained parallelism. Coarse-grained parallelism on the other hand presents a challenge to the algorithm. Consider the following example where a single iteration takes a long time to execute compared to the time required for scheduling: Worker A schedules a loop task $T(0, 16)$, splits it into subtasks $T(0, 8)$ and $T(8, 16)$, and starts executing iteration 0. Worker B succeeds in stealing $T(8, 16)$ from worker A, splits it into $T(8, 12)$ and $T(12, 16)$, and starts executing iteration 8. Because worker A is still executing iteration 0, only $T(12, 16)$ can be stolen. The remaining four iterations can be shared among three other workers C, D, and E, before no more parallelism is available. If there were more than five idle workers, LBS would fail to achieve good load balance. Idle workers would have to wait until either worker A or worker B splits again.

This example is meant to illustrate a potential problem of splitting a loop task into two (equal) halves. Instead of keeping one half of the iterations and leaving the other half for load balancing, worker A should keep a smaller fraction so that the remaining iterations can be balanced without relying on worker A to split again.

Now suppose there are eight workers. Worker A schedules loop task $T(0, 16)$, finds that up to seven workers might be looking for work, and thus enqueues $\frac{7}{8}$ of the iterations, $T(2, 16)$, for load balancing. Worker B succeeds to steal from worker A, finds that $T(2, 16)$ has been split once, and enqueues $\frac{6}{7}$ of the iterations, $T(4, 16)$. Worker C steals from worker B, finds that $T(4, 16)$ has been split twice, enqueues $\frac{5}{6}$ of the iterations, $T(6, 16)$, and so on. Continuing in this way, the iterations are perfectly balanced among the workers.

We call this strategy *guided splitting* because of its similarity to guided scheduling [21]: the fraction of enqueued iterations keeps shrinking until reaching a preset value, corresponding to split-half. Figure 2 shows the pseudocode for guided splitting.

Every loop task L has an additional attribute $splitFor$ that records the split that led to its creation. A value of 5, for example, indicates that L should have enough iterations for at least five more workers. Specifically, a value of $n > 1$ implies that the iteration range of the parent loop task has been split at a ratio of $n : 1$, and that L contains n times the number of iterations of its sibling, or, equivalently, $\frac{n}{n+1}$ of the iterations of its parent. A value of 1 indicates that we have arrived at the base case and all subsequent splits of L will use split-half. The value 0 is reserved for

```
SPLIT-GUIDED(L)
 1  assert(L.iterations > 1)
 2  assert(L.splitFor < NUM-WORKERS)
 3  if L.splitFor == 0
 4       f = NUM-WORKERS
 5  else
 6       f = max(L.splitFor, 2)
 7  if L.iterations < f
 8       f = 2
 9  split = L.start + (L.end − L.start)/f
10  L' = CREATETASK(L)
11  L'.range = [split, L.end)
12  L.range  = [L.start, split)
13  L'.splitFor = f − 1
14  L.splitFor  = 1
15  return L'
```

Figure 2: Pseudocode for guided splitting. SPLIT-GUIDED(L) modifies loop task L by splitting off L' depending on the value of $L.splitFor$. Because L is modified in place, only one task needs to be created. Falling back to split-half if $L.iterations < f$ guarantees that at least one iteration will be left over in L.

newly created loop tasks that have not been split before.

The code in Figure 2 shows that the iteration range of a loop task L is divided according to f, which can take on a value between 2 and the number of workers. L is then reduced to $\frac{1}{f}$ of its original iterations. The remaining iterations are wrapped up into a new loop task L', which is subsequently pushed onto the local queue.

The algorithm falls back to split-half ($f = 2$) if the number of iterations is less than f. This guarantees that L will contain at least one more iteration after splitting.

3.3 Adaptive Splitting

Guided splitting should work well if most of the workers are idle when scheduling a loop task. If most of the workers are busy, however, enqueuing more than half of the iterations through splitting may actually lead to increased overheads. Consider the following worst-case scenario: Worker A schedules a loop task with 1024 iterations. There are 15 other workers, but none of them is idle or attempts

to steal from worker A. Following guided splitting, worker A takes the first 64 iterations and pushes the other iterations onto its queue. After completing its iterations, worker A schedules the previously enqueued loop task, takes the next 64 iterations, and pushes the rest back onto the queue. Continuing in this way, worker A splits 19 more times before completing the last iteration. Using LBS with split-half on the other hand would save us 11 splits, including the associated overhead of creating and scheduling 11 additional tasks.

To avoid unnecessary overheads, we need an algorithm that favors split-half over guided splitting when the load is already well-balanced. The idea here is simple: measure the load imbalance by counting the number of idle workers P_i and split accordingly[1]. If we find out that more than one worker is idle, we enqueue a larger fraction of the iteration range, similar to what we did when using guided splitting. Otherwise, if at most one worker is idle, we assume that split-half is good enough for load balancing. We call this strategy *adaptive splitting* because it can change its behavior depending on the workers' *idleness*, which reflects the load balance (or imbalance) at runtime. Implementing adaptive splitting requires scheduler extensions beyond the work-splitting subroutine. In the following, we explore two implementations; one using global state, the other using local, per-worker information for quantifying idleness.

Computing Idleness The easiest way to compute P_i is to have a global variable that is incremented whenever a worker fails to get some work and decremented whenever a worker manages to find some new work. Figure 3 shows the pseudocode for adaptive splitting based on using such a global variable.

The counter $Global$ is updated atomically by all P workers to keep track of P_i. Initially, $Global$ is set to P, accounting for the fact that every worker starts out idle at the beginning of its execution. Likewise, after completing all pending tasks, $Global$ is reset to P. Note that, whenever the value of $Global$ is read in SPLIT-ADAPT-G, at least one worker must be busy—namely the one that wants to split—and thus P_i can be at most $P - 1$. The general rule is to enqueue $\frac{P_i}{P_i+1}$ of the remaining N iterations, unless $P_i = 0$ or $N \leq P_i$, in which case the algorithm falls back to using split-half.

While SPLIT-ADAPT-G can base its choice of how to split on an accurate measure of idleness, updates to a global variable are a source of contention, which can have a negative effect on scalability. We would rather prefer to trade off some accuracy for improved scalability. For this reason, we explore a second implementation of adaptive splitting, based on using local, per-worker idleness information.

[1] Alternatively, one could also count the number of busy workers P_b and derive the number of idle workers from $P_i = P - P_b$.

```
SPLIT-ADAPT-G(L)
 1  assert(L.iterations > 1)
 2  let Global refer to the global counter for P_i
 3  idle = ATOMICREAD(Global)
 4  f = max(idle + 1, 2)
 5  if L.iterations < f
 6      f = 2
 7  split = L.start + (L.end − L.start)/f
 8  L' = CREATETASK(L)
 9  L'.range = [split, L.end)
10  L.range  = [L.start, split)
11  return L'
```

Figure 3: Pseudocode for adaptive splitting based on using a global variable for counting the number of idle workers P_i. To maintain a proper count, the scheduler must signal whenever a worker transitions from busy to idle or vice versa (pseudocode not shown).

Estimating Idleness Instead of maintaining one global counter to keep track of P_i, every worker gets its own counter to *estimate* P_i independently of other workers. The key idea is the following: whenever a worker runs out of work, it increments the counters belonging to other workers to signal that it is looking for work. By doing so, it can affect the number of iterations that will be made available through work-splitting.

Figure 4 shows the pseudocode for adaptive splitting based on using local counters. The variable *Local* can have a value between 0 and $P - 1$. A value of n indicates that n workers have become idle since the last split. Recall that a worker is idle when it has to wait for another worker to create a task that it can steal. Thus, if worker A fails to steal from worker B because B's queue is empty, A increments B's counter if it hasn't done so already since B's last split. Reading a value of n, however, does not necessarily mean that P_i still equals n. If worker A had incremented B's idle estimate and subsequently found some work, A would have to decrement B's idle estimate to signal that it is no longer idle. In fact, this could mean decrementing up to $P - 1$ counters. To avoid this bookkeeping overhead, we choose to reset the value of a local counter only after splitting (line 16), and thus accept the possibility that idleness may be overestimated.

Independent counters alone are not always sufficient to make reasonable splits. In particular, consider the following example: when the runtime system starts up,

```
SPLIT-ADAPT-L(L)
 1  assert(L.iterations > 1)
 2  assert(L.splitFor < NUM-WORKERS)
 3  let Local refer to the local counter for P_i
 4  idle = ATOMICREAD(Local)
 5  if L.splitFor > 0
 6       f = min(idle, L.splitFor − 1)
 7  f = max(f + 1, 2)
 8  if L.iterations < f
 9       f = 2
10  split = L.start + (L.end − L.start)/f
11  L' = CREATETASK(L)
12  L'.range = [split, L.end)
13  L.range  = [L.start, split)
14  L'.splitFor = f − 1
15  L.splitFor  = 1
16  RESET(Local)
17  return L'
```

Figure 4: Pseudocode for adaptive splitting based on using one counter per worker for estimating P_i. If worker A fails to steal from worker B, A increments B's counter if it hasn't done so since B's last "read and reset" (pseudocode not shown).

every worker must assume that every other worker is idle, i.e., it estimates that $P_i = P$. If one worker now schedules a loop task, it will enqueue $\frac{P-1}{P}$ of the iterations. Despite the fact that one worker starts executing iterations, every other worker continues to assume that $P_i = P$ until it had the chance to reset its local counter at the end of SPLIT-ADAPT-L. As a result, up to $P - 1$ subsequent splits will each enqueue $\frac{P-1}{P}$ of the remaining iterations, instead of adapting the fraction to account for a decreasing number of iterations. To avoid such overestimations of P_i, we reuse the loop task attribute *splitFor* to bound the fraction of enqueued iterations for every loop task L that has been split before. Note that, once $L.splitFor == 1$, subsequent splits of L will use split-half, regardless of the value of $Local$.

Comparison Both implementations of adaptive splitting can fall back to using split-half when the load is well-balanced. But only SPLIT-ADAPT-G is truly adaptive in the sense that splitting is solely based on the computed idleness at runtime.

In SPLIT-ADAPT-L, once a loop task L has been split, subsequent splits of L are bounded by the previous splits, regardless of the supposed number of idle workers. This is in line with our implementation of guided splitting, except that we managed to remove the implicit assumption that all workers are idle when a parallel loop is encountered.

Figure 5 compares the two implementations in terms of the splits performed in two applications—matrix multiplication and sparse LU decomposition—which are detailed in Sections 4.5 and 4.6. For each split, we determined the fraction of iterations enqueued by SPLIT-ADAPT-L versus SPLIT-ADAPT-G.

The left histograms in Figures 5(a) and (b) show how the splits would look like if SPLIT-ADAPT-L used an unbounded estimate of idleness[2]. We can see by the left-skew that SPLIT-ADAPT-L tends to overestimate idleness compared to SPLIT-ADAPT-G, which is directly reflected in the number of iterations that are enqueued.

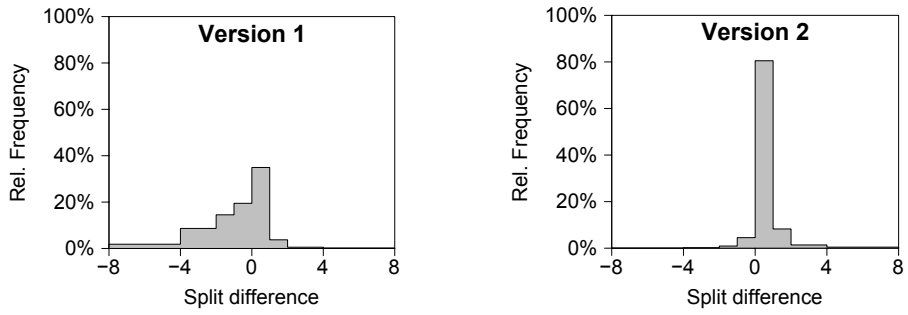
The right histograms in Figures 5(a) and (b) show the actual splits performed by our final algorithms. Using the loop task attribute $splitFor$ as an upper bound for idleness helps to approach the behavior of SPLIT-ADAPT-G. For the matrix multiplication, 80.5% of the splits are identical and 12.8% are within a difference of 1. For the less regular LU decomposition, 66.4% of the splits are identical and 21.5% are within a difference of 1. Overall, we see that our implementations of adaptive splitting end up making similar decisions, despite using different approaches to quantify idleness.

3.4 Distributed Splitting

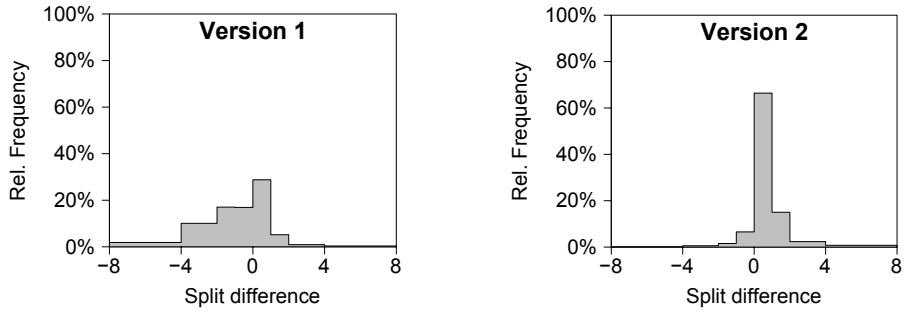
All work-splitting algorithms that we have discussed so far rely on work-stealing to distribute the work in the system. Sharing the iterations of a loop task among P workers involves at least P splits and $P-1$ steals. But having up to $P-1$ workers contend for iterations can slow down the distribution of work considerably.

Ideally, we would like to have a work-splitting scheduler that performs as few splits and steals as possible to distribute the work among idle workers. In other words, we would like to be able to "feed" all idle workers simultaneously. We attempt to approach this ideal behavior by combining work-splitting with a static scheduling step. Consider the following example. Worker A schedules a loop task with 64 iterations and decides to share them with three other workers. Thus, worker A assigns 16 iterations to each of the three workers and keeps the remaining iterations for itself. After this initial distribution, all four workers can start splitting and executing their iteration ranges.

[2]Remove lines 5, 6, 14, and 15 from SPLIT-ADAPT-L.



(a) Block matrix multiplication



(b) Block LU decomposition

Figure 5: Relative frequency histograms showing the difference between two implementations of adaptive splitting—SPLIT-ADAPT-G and SPLIT-ADAPT-L—in terms of the splits performed. The difference between two splits is calculated as $f_{global} - f_{local}$, where f_{global} and f_{local} correspond to the final values of f in Figures 3 and 4, respectively. Thus, a difference of less than 0 means that SPLIT-ADAPT-L has enqueued a larger fraction of the remaining iterations as a result of overestimating idleness compared to SPLIT-ADAPT-G. To see the effect of using the loop task attribute $splitFor$ as an upper bound for f_{local}, we distinguish between two versions of SPLIT-ADAPT-L. Version 2, our final algorithm, comes close to SPLIT-ADAPT-G, whereas Version 1, a variant without bounding f_{local}, tends to overestimate idleness. The number of workers in these experiments is 16.

We call this strategy *distributed splitting* because loop tasks can be additionally distributed in an attempt to reduce the frequency of steals. It should be emphasized that distributing a loop task is an option rather than a requirement. In fact, assigning busy workers even more work in advance may be counterproductive. In the worst case, an otherwise idle worker could give away $\frac{P-1}{P}$ of its iterations to $P-1$ workers, keeping only $\frac{1}{P}$ for itself. Such a decision would increase the number of subsequent steals, and would thus achieve the opposite of what we intended.

To make effective use of distributed splitting, we have to combine it with adaptive splitting to take advantage of the idleness information maintained by the latter strategy. If all workers seem to be idle, we can safely distribute the iterations of a loop task. Otherwise, we proceed with adaptive splitting. Alternatively, we could establish a threshold for P_i above which we consider it safe to distribute. Note that we never know *which* particular worker is idle, so when we choose to distribute, we divide the iterations evenly among all workers.

3.5 Scheduler Integration

We have extended our existing work-stealing scheduler for the Cell processor [10] with the proposed support for work-splitting. Figure 6 gives an overview of the scheduler's operation.

Tasks are stored in a distributed task pool, which is built around the idea of keeping tasks close to the SPEs. Each SPE takes advantage of an array-based task queue in its local store. If an SPE finds its queue empty, it first searches a pool of remaining tasks outside of local storage (lines 5-6), before trying to steal from another SPE's local queue (line 8).

To allow for stealing, the queues are split into a private and a public part, one for local, lock-free access by the queue owner, the other for shared and synchronized access by all threads. Depending on the number of tasks, the queue owner is responsible for sharing at least one task via the public segment. In this way, loop tasks that are enqueued after splitting are guaranteed to be available for load balancing. Work-stealing follows the steal-half policy and does not distinguish between loop and non-loop tasks. Only after removing a task from the task pool, the scheduler makes sure to take the appropriate action by inspecting the task (line 11).

4 Experimental Results

In order to compare the performance of the different schedulers, we first analyze their behavior in the context of single loops generated by a small benchmark ap-

```
SCHEDULETASK()
 1  T = DEQUEUEPRIVATE()
 2  if T == NIL
 3      T = DEQUEUEPUBLIC()
 4      if T == NIL
 5          B = GETTASKBLOCK()
 6          T = SWAPIN(B)
 7          if T == NIL
 8              T = STEAL()
 9              if T == NIL
10                  return FALSE
11  if T.type == LOOP and T.iterations > 1
12      SPLITANDEXECUTE(T)
13  else
14      EXECUTE(T)
15  return TRUE
```

Figure 6: Internal structure of our extended work-stealing scheduler for the SPEs of a Cell processor. Because local storage is a limited resource, tasks can be moved to main memory and back into local storage when needed (typically done in bundles of tasks or task blocks). If the scheduler got a loop task with more than one iteration, it proceeds with work-splitting.

plication, before moving on to real-world application kernels with iterative loop parallelism. The synthetic benchmark allows us to simulate loops with the following characteristics:

- Fine-grained parallelism: many independent iterations, each iteration executes in a very small amount of time (1μs)
- Coarse-grained parallelism: only few independent iterations, each iteration requires a long time to complete (10ms)
- Nested parallelism: sufficiently many independent iterations, iterations are of varying length (between 0.1ms and 1ms for the outer loop and 10μs for the inner loop)

Our initial tests on real-world applications include two important kernels from linear algebra with different task structures:

- Block matrix multiplication: contains sufficient parallelism in the form of parallel loops with balanced iterations
- Block LU decomposition of a sparse matrix: is characterized by decreasing parallelism over the course of the computation, contains parallel loops with unbalanced iterations

We ran our tests on an IBM BladeCenter QS22 with two PowerXCell 8i processors for up to 16 Synergistic Processing Elements (SPEs). Programming support is provided by the IBM Cell SDK version 3.1 [1] on top of Fedora 9 with Linux kernel 2.6.25-14. The reported execution times are the average of 20 program runs. Table 1 summarizes the schedulers used in the evaluation.

For reference, we include the performance of Cell Superscalar (CellSs), a task-based programming environment and corresponding runtime system for the Cell processor [5, 20]. CellSs relies on the Power Processing Element (PPE) to create and distribute tasks to the SPEs. Thus, truly nested parallelism involving task creation on SPE side is not supported. Matrix multiplication and LU decomposition codes are adapted from the CellSs distribution.

Performance will largely depend on two (conflicting) factors: runtime overheads and load balance. Runtime overheads can be reduced by combining many small tasks into fewer large tasks. This "serialization" of parallel work is critically important to avoid generating large amounts of parallelism that would go unused. On the other hand, aggressive serialization may affect load balance by removing too much parallelism. For this reason, we expect to see the best performance from schedulers that serialize as much tasks/iterations as possible, while still maintaining good load balance.

Table 1: Summary of the work-splitting and non-work-splitting schedulers used in the evaluation.

Scheduler	Description
Plain	Lazy Binary Splitting (LBS) with split-half
Guided	LBS with guided splitting
Adapt-G	LBS with adaptive splitting based on global idle count
Adapt-L	LBS with adaptive splitting based on per-worker idle estimates
Distrib	LBS with distributed adaptive splitting based on per-worker idle estimates
Offload	Scheduler that "offloads" task creation to half of the workers, the other half starts stealing
CellSs	Cell Superscalar 2.1, centralized scheduler, all tasks are created by a master thread

4.1 Fine-grained Loop Parallelism

Fine-grained parallelism occurs in loops with large numbers of independent but very short iterations. Without increasing the granularity at runtime, this parallelism will be hard to exploit due to the cost of scheduling. We consider the following loop, which translates into a single loop task:

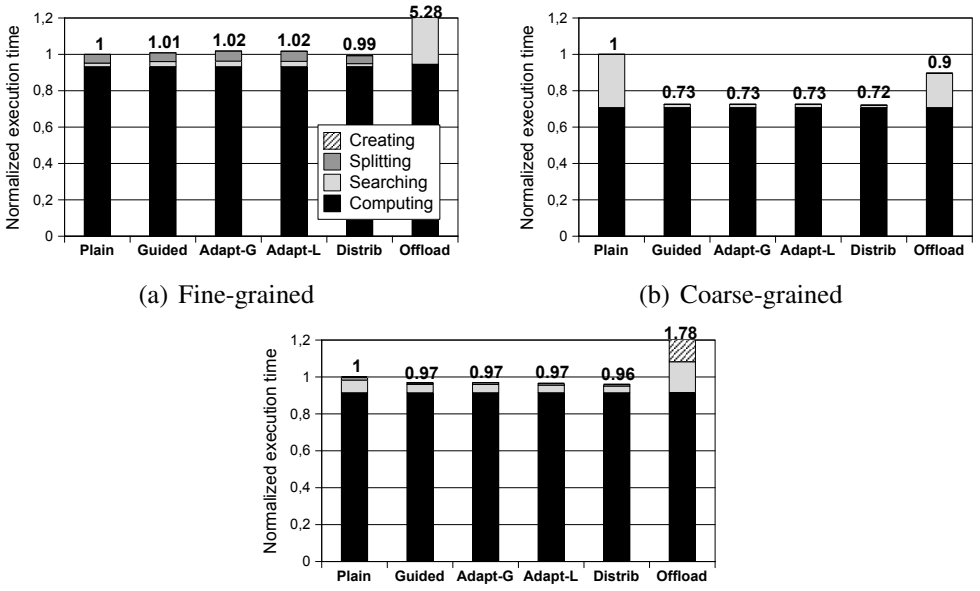
```
for (i = 0; i < 1000000; i++)
    compute(1);
```

The body of the loop, represented by `compute`, is parameterized to run for a given number of microseconds. In this case, an iteration executes in 1μs, which corresponds to 3200 SPU clock cycles[3].

Figure 7(a) shows the resulting execution times, broken down into different components to identify parallel overheads. The scheduling component (Searching) includes the time spent waiting for tasks to become available (idleness) and is thus an indication of the achieved load balance. For the work-splitting schedulers, we combine the time spent splitting and creating tasks into one component (Splitting).

All work-splitting strategies deliver roughly the same performance, which shows that LBS is a good choice for scheduling fine-grained parallelism. We can see that creating a task for each iteration generates massive overhead, resulting in only 19% of the performance achieved by work-splitting. Clearly, serializing iterations is required for efficient execution.

[3]The typical SPU clock frequency is 3.2 GHz.



(a) Fine-grained

(b) Coarse-grained

(c) Nested

Figure 7: Normalized execution time of loops with (a) fine-grained, (b) coarse-grained, and (c) nested parallelism, using different schedulers on two Cell processors with 16 SPEs. In (a), the task creation overhead of Offload is significant but not visible. The normalized execution time of 5.28 breaks down into 0.95 for computing, 2.24 for searching/scheduling, and 2.08 for creating.

Looking more closely at the execution times, we see that guided and adaptive splitting incur slightly more overhead than split-half. While the latter manages to serialize 99.4% of the iterations, creating around 5600 tasks, Adapt-L, for example, creates around 9800 tasks. As a result of creating more tasks, scheduling overheads increase as well. Distributed splitting succeeds in reducing overheads by serializing 99.6% of the iterations. The effect on overall performance remains hardly noticeable. All work-splitting schedulers perform within 1-3% of each other.

4.2 Coarse-grained Loop Parallelism

Coarse-grained parallelism is characterized by iterations that take a long time to complete. Unlike fine-grained parallelism, where slight variations in load balance may not necessarily affect performance, coarse-grained parallelism depends on equal distribution of work, or otherwise, scalability will be limited. Here we consider the following loop:

```
for (i = 0; i < 64; i++)
    compute(10000);
```

The computations are scaled up so that each iteration runs for 10ms. 64 iterations should represent enough parallelism to keep 16 workers occupied. With the given granularity, we expect to see the best performance if every worker executes the same number of iterations, namely four.

Figure 7(b) compares the execution times. All schedulers achieve significant improvements over plain LBS—up to 39% on average. The breakdown explains why. Plain LBS spends a large fraction of its execution time searching for tasks. This indicates difficulties in balancing the iterations. In fact, we count roughly 33 000 attempts to steal a task, of which only 36 succeed on average. In comparison, all other schedulers except Offload attempt 1000 steals or less. Again, distributed splitting has the lowest overhead of all work-splitting strategies.

4.3 Nested Loop Parallelism

So far, we have only considered parallel loops with balanced iterations. Such loops could be easily scheduled without making decisions at runtime. We now pay attention to unbalanced iterations in the form of a nested loop:

```
for (i = 0; i < 64; i++) {
    compute(r * 100);
    for (j = i * 4; j < 1024; j++)
        compute(10);
}
```

Each iteration of the outer loop performs some work and creates another loop task, with the trip count depending on the value of `i`. The outer loop is rather coarse-grained, computing between 0.1ms and 1ms (`r` is a random integer in the range of 1 and 10), while the inner loop has fine-grained iterations of 10μs each.

Figure 7(c) reveals only a small performance difference between the work-splitting schedulers. Because all variants are based on LBS, their behavior is identical when there is enough parallelism such that the task queues are not empty. In that case, splitting is not immediately required and more iterations can be serialized. The slight improvements over plain LBS can be attributed to reduced splitting and more efficient load balancing. Once again, distributed splitting achieves the best performance.

Unlike the work-splitting schedulers, Offload creates a task for each of the 33 344 independent iterations. Because of the fine-grained inner loop, task creation adds significant overhead.

Figure 8 summarizes benchmark performance by showing speedups over single SPE execution with work-splitting. Note that this baseline closely matches

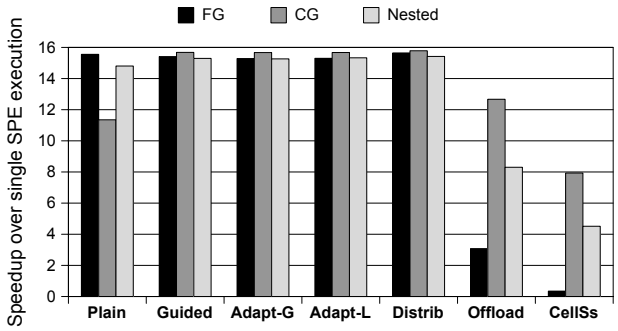


Figure 8: Performance of different schedulers on fine-grained (FG), coarse-grained (CG), and nested (Nested) loop parallelism, on two Cell processors with 16 SPEs.

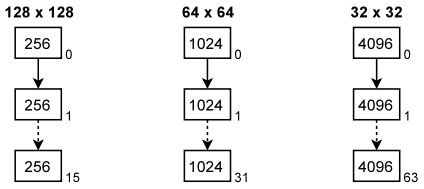
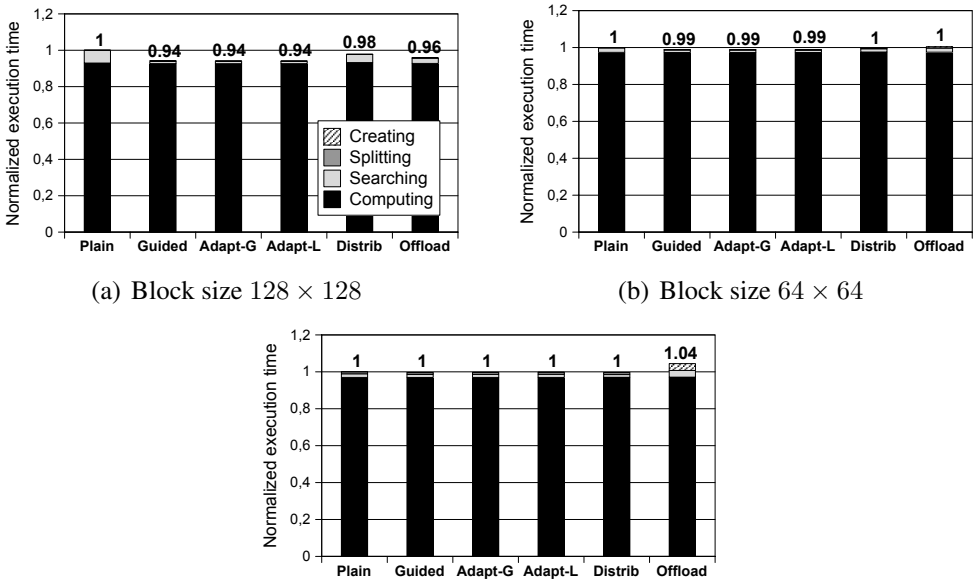


Figure 9: Task structure of the block matrix multiplication, given two 2048×2048 matrices and blocks of size 128×128, 64×64, and 32×32, respectively. Boxes represent loop tasks with the displayed numbers of iterations.

strictly sequential execution because a single SPE never splits but serializes all iterations of a loop. Distributed splitting achieves the best performance in all three cases. Creating a task for each iteration can only be considered an option if parallelism is sufficiently coarse-grained. Compared to CellSs, Offload can improve performance by creating tasks in parallel. Lacking granularity control, however, fine-grained parallelism is very hard to exploit; much more so for CellSs.

4.4 Block Matrix Multiplication

Figure 9 shows the task structure of a block matrix multiplication after interchanging the outermost and innermost loops and combining the two inner loops into a single loop task. Thus, for each iteration of the sequential outer loop, we create a loop task with $(D/B)^2$ iterations, where D is the dimension of the matrices and B is the block size for the multiplication. In our experiments, we fix the dimension and vary the block size to obtain different levels of granularity. With $D = 2048$



(a) Block size 128×128

(b) Block size 64×64

(c) Block size 32×32

Figure 10: Normalized execution time of block matrix multiplication using different schedulers on two Cell processors with 16 SPEs.

and $B = 128$, 64, or 32, we get the iteration counts shown in Figure 9. Edges connecting loop tasks indicate dependencies between iterations, which are easily resolved by synchronizing execution after every loop task.

Using any one of these block sizes guarantees sufficient parallelism, although with varying granularity. Task lengths range from coarse-grained 2.8ms for blocks of size 128 to relatively fine-grained 60μs for blocks of size 32. Multiplying two blocks of size 64 lies in between at 390μs[4].

Figure 10 shows the execution times for the different block sizes. There are several things to observe: (1) Alternative splitting strategies can do slightly better than split-half in the case of coarse-grained parallelism. (2) Split-half works well for fine-grained parallelism, so there is no need to use other strategies, such as adaptive splitting, in that case. (3) When task creation and scheduling overheads are negligible compared to the amount of actual work, there is little advantage in serializing tasks, and schedulers like Offload can deliver comparable performance.

Figure 10(a) shows that the slight performance improvement over plain LBS comes from less scheduling overhead, indicating better load balancing. With blocks of size 128, Plain serializes 52% of the iterations, leaving 48% that are created as tasks. Out of these tasks, 70% are scheduled by work-stealing. Adapt-

[4]Task lengths refer to SPE execution times.

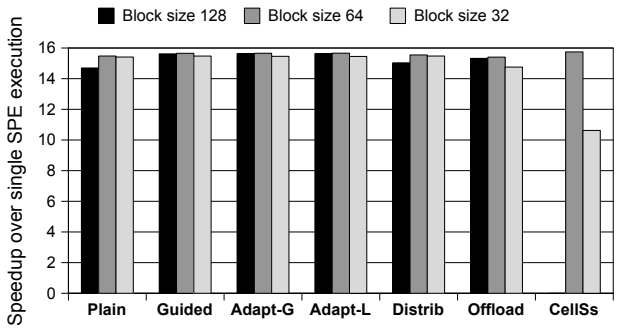


Figure 11: Block matrix multiplication: performance of different schedulers on two Cell processors with 16 SPEs.

L for example improves both serialization and work-stealing frequencies to 69% and 19%. Fewer splitting and stealing translates to less overhead, improving overall performance, provided that the load is balanced.

With blocks of size 32, all work-splitting schedulers manage to serialize 85-90% of the iterations, meaning that only 10-15% are actually created as tasks. Despite its overheads, Offload performs within 4% of the work-splitting schedulers.

Figure 11 summarizes the performance of the different schedulers. Among the work-splitting variants, there is no single best strategy that should be preferred over others. CellSs achieves performance on par with our best schedulers when using blocks of size 64, but starts to fall behind when using smaller blocks. In this case, centralized scheduling, as implemented in CellSs, introduces a sequential bottleneck that limits the degree to which fine-grained parallelism can be exploited. Because we did not succeed in using a block size of 128 with the CellSs runtime, the corresponding numbers are missing in Figure 11 (and also in Figure 14).

4.5 Block LU Decomposition

The task-based implementation of block LU decomposition contains four different types of tasks: diagonal block factorization, panel factorization, row computation, and trailing sub-matrix update. Panel factorization and row update can be performed in parallel after factorizing a given diagonal block. As the algorithm proceeds, the size of the trailing sub-matrix is reduced, which leads to fewer parallelism in each subsequent iteration.

Figure 12 shows the task structure of decomposing a 2048×2048 matrix, using block sizes of 128×128, 64×64, and 32×32. Because of the sparsity

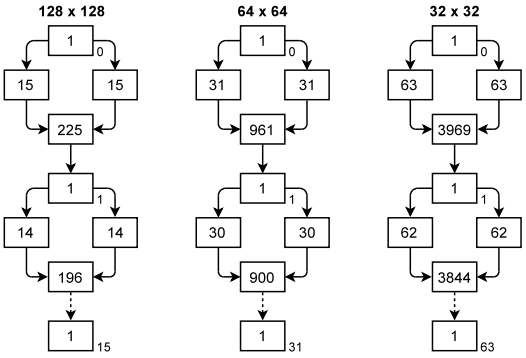
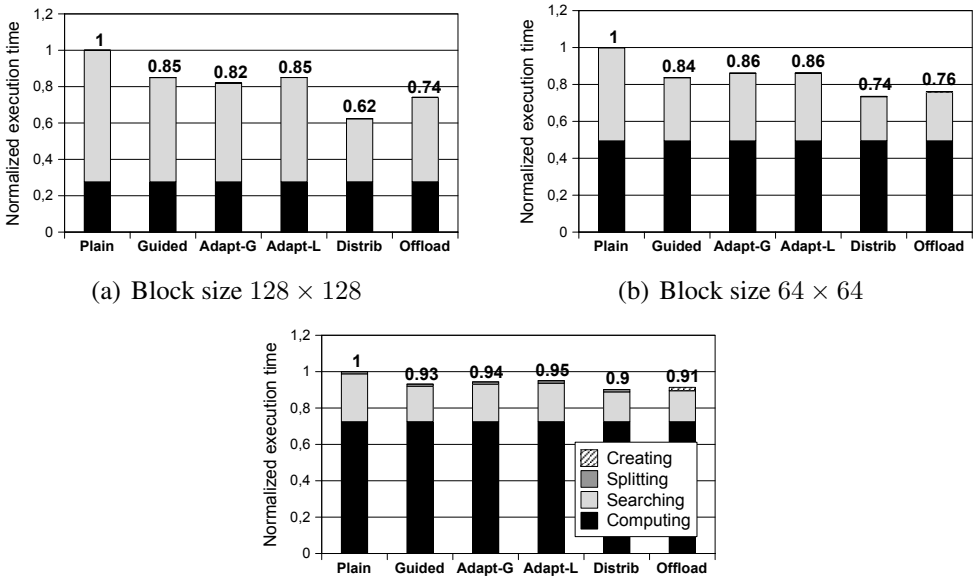


Figure 12: Task structure of the block LU decomposition, given a 2048×2048 matrix and blocks of size 128×128, 64×64, and 32×32, respectively. Boxes represent loop tasks with the displayed numbers of iterations. Within an iteration of the algorithm, each of the four parallel loops describes a different type of task—diagonal block factorization, panel factorization, row computation, and trailing sub-matrix update.

of the input matrix (some blocks are zero-valued), the actual numbers of tasks to be executed will be smaller than the numbers of iterations displayed in Figure 12. Using a block size of 128 results in limited coarse-grained parallelism on the order of 15ms per task. Thus, increased load imbalance, especially in the final iterations of the algorithm, motivates a decomposition at a finer granularity. In fact, better performance is achieved when using blocks of size 64 or 32. In these two cases, task lengths are on average 1ms and 190μs. Note that factorizing a diagonal block is a single task, expressed in terms of a loop task with a single iteration.

Figure 13 compares the resulting execution times. We identify the following trends: (1) In the case of limited parallelism (loop tasks with few, possibly unbalanced iterations), distributed splitting facilitates an equal distribution of work, which is difficult to achieve otherwise. Distrib improves performance by up to 60% compared to plain LBS. Because the available parallelism quickly becomes a bottleneck, giving up on serialization is the second best choice here: Offload is up to 35% better than plain LBS. (2) The performance of LBS depends on the amount of exploitable parallelism. Good load balance is only guaranteed when there is sufficient parallelism. Otherwise, split-half might not be the best strategy to reduce load imbalance. (3) Because a large fraction of the execution time is spent stealing, Adapt-L, which required an extended work-stealing implementation, shows higher scheduling overheads than Adapt-G.

Across all block sizes, Distrib spends the least time searching for tasks. With blocks of size 128, Distrib serializes 64% of the iterations, leaving 36% of which



(a) Block size 128×128

(b) Block size 64×64

(c) Block size 32×32

Figure 13: Normalized execution time of block LU decomposition using different schedulers on two Cell processors with 16 SPEs.

49% are stolen. On the other hand, plain LBS creates 14% more parallelism than Distrib and yet schedules 82% of this parallelism by work-stealing. Thus, Distrib achieves much better load balance than plain LBS, resulting in the substantial performance advantage of 60%.

Reducing the block size leads to fewer problems with load imbalance. With blocks of size 32, plain LBS manages to serialize 77% of the iterations. Distrib can further reduce overheads by serializing 83% of the iterations, while also stealing fewer tasks.

Figure 14 summarizes the performance of the schedulers. CellSs achieves comparable performance, but becomes less efficient when dealing with limited parallelism.

5 Related Work

Lazy Binary Splitting The Lazy Binary Splitting (LBS) algorithm for scheduling nested parallelism on shared-memory multiprocessors has been introduced in [25]. LBS in turn builds on existing work-splitting schedulers as implemented in Intel's Threading Building Blocks (TBB) [22] and Cilk++ [17]. Unlike these

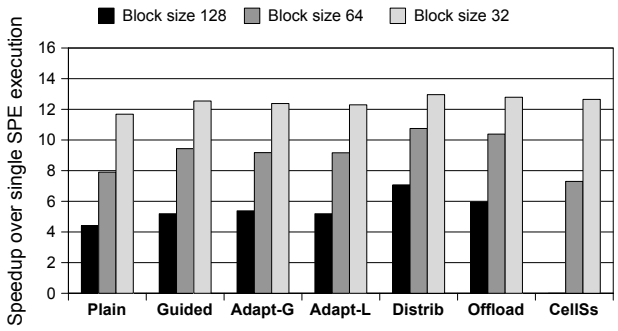


Figure 14: Block LU decomposition: performance of different schedulers on two Cell processors with 16 SPEs.

schedulers, which split a loop repeatedly depending on some predetermined grain size, LBS is able to postpone splitting until additional parallelism is actually useful. While the original algorithm works well for fine-grained parallelism, more elaborate splitting strategies are needed to handle coarser-grained parallelism efficiently, as demonstrated in this work.

Dynamic Loop Scheduling and Chunking Extensive research has gone into dynamic loop scheduling and chunking techniques. OpenMP for example lets the programmer choose among different scheduling policies, including static, dynamic, and guided scheduling of chunks to worker threads [2]. Much like task bundling, chunking was designed to reduce scheduling overheads by executing iterations of a loop in sequential chunks. As such, the iterations of a chunk are executed by the worker to which they are assigned. Large chunk sizes may effectively reduce overheads, but likely at the expense of load balance.

Load balance can be improved by setting up a work queue and fetching iterations from that queue. This is known as self-scheduling, a technique that is also central to the idea of tasking. Combining self-scheduling and chunking has led to many more algorithms, such as guided self-scheduling [21], in which a fraction of the remaining iterations is assigned to every idle worker, or trapezoid self-scheduling [26], in which the chunk size is linearly decreased over the course of the computation. In a similar manner, factoring keeps decreasing the chunk size, but does so only after scheduling a batch of chunks of the same size [12]. Weighted factoring [11] and adaptive weighted factoring [4] are variants of factoring for use in heterogeneous systems.

Granularity Control Work on controlling the granularity of tasks is motivated by the observation that the ideal parallel runtime, i.e., the runtime with the lowest possible overhead, maximizes task granularity while maintaining load balance. In other words, too fine-grained parallelism should be avoided because of its large runtime overheads. In addition to enforcing a minimum task size at compile-time, runtime decisions are often indispensable for efficient granularity control [19]. Load-based inlining [14], lazy task creation [18], and adaptive cut-off [9] are serialization strategies that decide on a per-task basis whether a task should be spawned or not. Tasks that are inlined or cut-off are executed as part of their parent tasks, thereby increasing the granularity of the parent tasks. In addition, tasks may be created lazily even after inlining (from the continuation of the parent task) if load balancing demands it.

Parallel Programming Languages The High Productivity Computing Systems (HPCS) languages Chapel [8, 6], Fortress [3], and X10 [24, 7] share a common approach of expressing parallelism in terms of tasks rather than threads. Chapel, for example, is designed from the ground up with task parallelism in mind. Even data parallel constructs, such as the `forall` statement, are implemented on top of tasks, using simple chunking to reduce scheduling overheads. The number of tasks to execute a `forall` loop is currently controlled by three configuration variables, which can be set at program startup. Work-splitting provides an alternative to chunking that eliminates the need for manual parameter tuning. Thus, it has the potential to improve performance as well as programmability.

Efficient Parallel Runtime Systems Work-splitting is well suited for scheduling fine-grained loop parallelism. As task parallel programming becomes more widely used, computer architects propose hardware support to boost the performance of software runtime systems. Carbon introduces (bounded) hardware task queues and per-core prefetchers to hide the latency of accessing the task queues [15]. Tasks that overflow the hardware structures are swapped out to data structures in main memory. In addition to regular tasks, Carbon supports loop tasks, which are split into $\lceil N/S \rceil$ tasks, where N is the number of iterations and S is the chunk size. If we choose a chunk size of one, as we did in this work, Carbon creates a task for each iteration.

Rigel is a scalable accelerator architecture that supports task parallel programming via a low-level work queuing API [13]. Unlike Carbon, Rigel does not provide hardware structures for storing tasks. Task management is done entirely in software, built on top of efficient atomic operations and global synchronization primitives. Rigel's low-level programming interface includes a function to enqueue a parallel loop in a single operation, which is similar to enqueuing a loop

task, but the mechanism that is used to schedule iterations is left to the runtime.

Asynchronous Direct Messages (ADM) extend a shared-memory CMP with fast messaging between threads, enabling the implementation of efficient message-passing schedulers that bypass the cache hierarchy [23]. As opposed to Carbon, which fixes the splitting of loop tasks in hardware, ADM-based runtimes may fully benefit from flexible work-splitting in software. In particular, we think that ADM provides just the kind of primitives that would help accelerate implementations of distributed splitting.

6 Conclusions

In this paper, we have built on the idea of using loop tasks and Lazy Binary Splitting (LBS) as a basis for efficient loop scheduling in task parallel runtime systems. Starting with the LBS algorithm, we have developed several scheduler extensions—guided, adaptive, and distributed splitting—with the goal of improved performance on a wide range of parallel loops, including those for which LBS failed to achieve good load balance. Our experiments confirm that LBS is a simple yet efficient algorithm for scheduling fine-grained parallelism. In that case, some of our implementations, in particular those of adaptive splitting, start to show their bookkeeping overheads, which we hope to reduce in the future.

While schedulers that lack support for work-splitting can deliver comparable performance on coarse-grained parallelism, achieving good performance on fine-grained parallelism requires serialization of tasks at runtime to reduce overheads as much as possible. Based on our experiments, we conclude that a combination of static scheduling and subsequent work-splitting, i.e., distributed splitting, is the best candidate for efficient loop scheduling: it can significantly outperform LBS on coarse-grained parallelism, while it achieves comparable performance on fine-grained parallelism.

Acknowledgments

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

References

[1] IBM Software Development Kit (SDK) for Multicore Acceleration Version 3.1. `http://www.ibm.com/developerworks/power/cell`.

[2] OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification, Version 1.0. http://projectfortress.sun.com, March 2008.

[4] I. Banicescu, V. Velusamy, and J. Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Cluster Computing*, 6(3):215–226, July 2003.

[5] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, November 2006.

[6] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21:291–312, August 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, October 2005.

[8] Cray Inc. Chapel Language Specification, Version 0.796. http://chapel.cray.com, October 2010.

[9] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11.

[10] R. Hoffmann, A. Prell, and T. Rauber. Exploiting Fine-Grained Parallelism on Cell Processors. In *Euro-Par '10: Proceedings of the 16th international Euro-Par conference on Parallel Processing*, 2010.

[11] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328, 1996.

[12] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[13] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 140–151, 2009.

[14] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90, 1989.

[15] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, June 2007.

[16] D. Leijen, W. Schulte, and S. Burkhardt. The Design of a Task Parallel Library. In *OOPSLA '09: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–242.

[17] C. E. Leiserson. The Cilk++ Concurrency Platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, 2009.

[18] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 185–197, 1990.

[19] J. E. Moreira, D. Schouten, and C. D. Polychronopoulos. The Performance Impact of Granularity Control and Functional Parallelism. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 581–597, London, UK, 1996. Springer-Verlag.

[20] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, September 2007.

[21] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[22] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[23] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *ASPLOS '10: Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 311–322, 2010.

[24] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification, Version 2.1. `http://x10.codehaus.org`, December 2010.

[25] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy Binary-Splitting: A Run-Time Adaptive Work-Stealing Scheduler. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, 2010.

[26] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.