Flexible Scheduling and Thread Allocation for Synchronous Parallel Tasks

Christoph W. Kessler and Erik Hansson
IDA, Linköping University, 58183 Linköping, Sweden
{chrke,eriha}@ida.liu.se

Abstract: We describe a task model and dynamic scheduling and resource allocation mechanism for synchronous parallel tasks to be executed on SPMD-programmed synchronous shared-memory MIMD parallel architectures with uniform, unit-time memory access and strict memory consistency, also known in the literature as PRAMs (Parallel Random Access Machines).

Our task model provides a two-tier programming model for PRAMs that flexibly combines SPMD and fork-join parallelism within the same application. It offers flexibility by dynamic scheduling and late resource binding while preserving the PRAM execution properties within each task, the only limitation being that the maximum number of threads that can be assigned to a task is limited to what the underlying architecture provides. In particular, our approach opens for automatic performance tuning at run-time by controlling the thread allocation for tasks based on run-time predictions.

By a prototype implementation of a synchronous parallel task API in the SPMD-based PRAM language Fork and experimental evaluation with example programs on the SBPRAM simulator, we show that a realization of the task model on a SPMD-programmable PRAM machine is feasible with moderate runtime overhead per task.

1 Introduction

During the recent years, computer architectures available on the consumer market have switched from single-core architectures to multi-cores, and it is reasonable to assume that we enter the many-core era in the near future. The reason for this change is that hardware manufacturers try to keep up with the demand of more computation power and at the same time consume less energy. As a consequence, speed-up of legacy, single-threaded computer programs does not come for free any more but requires rewriting to leverage many cores. Even worse is that, even where providing a shared memory abstraction, these new architectures mainly follow NUMA and SMP designs that lack features that could ease parallel programming, such as strong memory consistency or deterministic execution.

To ease the burden for both application programmers and compiler engineers, some architecture projects [PBB+02, For10, WV08] are working towards supporting more powerful, deterministic parallel programming models such as the PRAM model [FW78, KKT01]. The PRAM model is often considered as only a theoretical programming model, but already in the 1990s it has been realized in hardware, albeit not on a single chip, e.g. the SB-PRAM [PBB+02, KKT01]. In a current project by VTT Oulu (Finland) a new architecture

called *Replica* is being developed. It supports both PRAM and NUMA mode, and features massively hardware-multithreaded configurable very long instruction word (VLIW) processor cores with chained functional units and a powerful 2D mesh on-chip combining network providing uniform access to on-chip distributed shared memory. Replica will be realized in hardware and is the successor of the *Total Eclipse* architecture [For10].

PRAMs are instruction-level synchronous MIMD parallel architectures with shared memory and are traditionally programmed in the SPMD execution style using PRAM languages such as Fork [KKT01, KS97a], e [For04] etc. that map the naturally available tight synchronization of the underlying hardware to the expression and statement level, allowing to reduce explicit synchronization in the code while maintaining deterministic parallel execution.[1] While following the SPMD style across the whole machine gives full control over the assignment of computation to execution resources, it becomes cumbersome for more irregular application scenarios that require adaptive resource allocation strategies.

In this work, we show how a flexible MIMD task model allowing multithreaded PRAM tasks, can be realized on top of a SPMD programmed PRAM platform. The data-driven, dynamic scheduling principle of our task model is inspired by current single-threaded task programming models such as StarPU and StarSs. Our work is part of a pre-study for some features of the Replica architecture's runtime system. As the Replica simulator and software toolchain is not completely finished yet, we use the similar SBPRAM simulator and Fork toolchain [KKT01] for the prototype implementation and evaluation.

2 Principle

We are given a PRAM with p hardware threads and with a low-level programming model based on SPMD execution style, i.e., all p threads execute `main` from the beginning and the hardware itself does not provide for dynamic creation and deletion of additional threads[2]. Hence, a software layer on top of a low-level programming environment will be responsible for providing a task-based programming model. In the following, we propose a task-based programming model with non-preemptive dynamic scheduling, where the tasks can be serial or PRAM-style synchronous parallel computations and thus might require one or several threads of the underlying PRAM machine for execution.

Thread pool A program that uses synchronous parallel tasks or asynchronous sequential tasks (or both) should in the beginning have a single thread initialize the task system by calling `init_tasksystem();` and then send a subset of the available hardware threads as *worker threads* into a central shared *thread pool* TP, where they wait for work. A thread joins the thread pool by calling `join_threadpool()`. If no tasks have been created yet at this time, at least one thread should continue execution to create work for the others, and may still join the thread pool later, thereby becoming an additional worker thread.

[1] The strict memory consistency model of PRAMs is the strongest possible shared memory consistency model, it is even stronger than sequential consistency.

[2] In the following, thread means hardware thread (also known as *virtual processor*) unless otherwise stated.

```
typedef struct vector {
 int vid;     // unique ID for debugging purposes
 int state;   // 0 = data not ready, 1 = data valid
 void *pdata; // address of the wrapped payload data array
 int n_elems; // number of elements
 int type;    // element type field, refers to type table
 struct sptaskdescriptor *consumers[MAXCONSUMERSPERVECTOR];
 int n_consumers; // number of registered consumer tasks
} Vector;
```

Figure 1: Implementation of the `Vector` container in the C-based PRAM language Fork.

The program terminates successfully (by each thread calling `exit(0)`) if all p worker threads are waiting idle in the thread pool and there is no task left in the task queue (see below). A global counter holding the current number of threads waiting idle in TP can easily be maintained using atomic prefix-add operations.

Containers A container is a wrapper data structure that encapsulates aggregate user data such as an array together with metadata such as information about its size, type and state (invalid/ready), and manages memory access information such as where its most recent contents is currently located if there are multiple kinds of memory in the system. In particular, containers can, on such systems, provide consistent access to data on request (i.e., a call to the container's flush operation) by enforcing a write-back to the default memory location. On a PRAM this latter feature is actually not required, while it can be useful on a NUMA system as it provides an object-based distributed shared memory.

The most common container, and the only one that we support by now, is `Vector`, inspired by the corresponding container type in the C++ STL. For the C-based PRAM language Fork, our `Vector` is internally defined as shown in Figure 1. However, following a modular design style, the application programmer (API user) should not access these fields directly but use predefined access functions and macros instead, some of which will be described in the following.

In C++, `Vector` is generic in the element data type. In the C-based Fork language, we have to represent the element type explicitly using a `type` field. The `consumers` are those task instances (see later) that take this operand container with access mode "in".

The function `Vector *new_Vector(void *array, int length, int type );` allocates a new Vector container for `array` of `length` elements. The payload data `array` is not copied, only a pointer to it is stored in the container. Hence, it is possible that multiple containers point to the same payload data. This is a way to avoid unnecessary copying; it is the programmer's responsibility that this sharing of payload data does not lead to data races. An example will be given at the end of this section.

The state of a vector `v` can be set as follows. A task is blocked until all its argument containers are in ready state. Tasks waiting for an argument container to become valid are registered in the container so they can be notified. `setREADY(v)` sets `v` to state valid; this operation is used for containers holding input data to a task-based computation.

```
typedef struct sptaskdescriptor {
 int tid;  // Rank in Frozen Queue FQ
 void (* func)( int argc, Vector **argv, Vector *ret );
 sync void (* sfunc)( sh int argc, sh Vector **argv, sh Vector *ret );
 int argc;  // number of arguments
 Vector **args;  // dynam. allocated shared array of argument containers
 Vector *retvalue;  // container that holds the return value
 int minnthreads, maxnthreads;  // lower and upper bound for #threads
 int *shmem;  // pointer to shmem, initially 0
 int shmemsize;
 int nthreads; // actual number of threads running this task as a group
} sptask;
```

Figure 2: Data structure for a SP-task descriptor in C/Fork. One such entry exists for each task in the global shared heap memory.

`setREADYandpromote(v)` additionally notifies the consumer tasks that depend on `v`, and promotes these to READY state where `v` was the last awaited argument.

Tasks can also return data in a container, which usually is then used as input to subsequent, data-dependent tasks. Also in this case, the consumers will be notified and promoted to READY state as applicable. Depending on the type of return data, one of the following three versions of YIELD should be used:

- `YIELD_VALUE(ret,type,value)` copies scalar base-`type` data to (the first element of) a (pre-allocated) payload array in a (pre-allocated) `Vector ret`.
- `YIELD_PTR(ret,ptr)` replaces the payload array field in the `Vector ret` by the new array pointed to by `ptr`.
- `YIELD_VOID(ret)` is a variant of YIELD with no assigned return value. This is useful for in-place updates of the payload array; we will later see an example of this.

SP-functions and SP-tasks Synchronous parallel functions (SP-functions) are executed by the calling group of hardware threads in lock-step mode, hence the execution will be deterministic (assuming that the resolution of possible concurrent write access conflicts is deterministic, too).

We define *synchronous parallel tasks* (*SP-tasks*) as instantiations (invocations) of such synchronous parallel functions by a group of threads. The special case of invocations of SP-tasks by a single-thread group is the ordinary sequential task model known from classical scheduling theory. SP-tasks are a special case of *malleable tasks*, which can be executed by an arbitrary number of threads but are internally not necessarily synchronous.

Tasks are, at runtime, represented by a *task descriptor*, a data structure defined in Figure 2, which contains the key parameters of a task, such as the SP-function to be called, the argument vector and return value, minimum and maximum specified thread allocation (or default values if unspecified), and also some non-public administrative entries such as the task state. The `shmemsize` field holds the size of the shared memory block `shmem` to be

allocated to the task before execution; it must be 0 for an asynchronous task and > 0 for a synchronous task to accommodate its group stack and heap.

These task properties are set upon creation (see below) or derived automatically; it is not intended to change them during execution (e.g., no reallocation of its shared memory segment while the task is running). In future work we may add some get functions or macros to allow for querying of certain task properties.

Creating new tasks and SP-tasks At the run-time system programming level, task descriptors for asynchronous and synchronous tasks are created explicitly by the constructors

```
sptask *new_task ( void (*foo)(int, Vector **, Vector *),
                   int argc, Vector **args, Vector *ret );

sptask *new_stask ( sync void (*foo)(sh int, sh Vector **, sh Vector *),
                    int argc, Vector **args, Vector *ret,
                    int minp, int maxp, int shmemsize );
```

which take a function name and its arguments. The static type checking of synchronicity and sharity in Fork requires different constructors for synchronous and asynchronous tasks. `minp` and `maxp` specify the minimum and maximum number of threads to be used for this task. The implementation enforces at runtime that the value for `minp` is at least 1, and that of `maxp` is automatically truncated to the maximum available number of workers if it is too large. Hence, it is safe (but possibly not most efficient) to oversize `maxp`.

A task (synchronous or asynchronous) t can be spawned explicitly by a spawn operation: `spawn_task(`t`)` creates a task descriptor with the parameters given by t and enqueues it to a central scheduler for execution concurrently with the continuation of the spawning thread; control returns thus immediately to the spawning thread.

Lifecycle, scheduling and synchronization of tasks During its lifecycle, a task's state changes from new to ready to running to terminated. When spawned, created tasks receive a unique task ID (`tid` field) and are sent to a *frozen queue* FQ of tasks that are not yet data ready. Once all its input arguments (containers) are in ready state, a task is promoted to ready state and enqueued in a central shared task queue TQ, from where idle worker threads fetch new work for execution. All synchronization between SP-tasks is data driven.

From the task queue TQ, idle threads fetch their next task for execution. An asynchronous task will be assigned to exactly one thread. For synchronous tasks, at least `minnthreads` and at most `maxnthreads` idle threads will be collected, barrier-synchronized and assigned as a synchronous group to the execution of the task's SP-function. Once the task terminates, the task status will be changed to `TERMINATED`.

For now, we implemented for SP-tasks the thread assignment policy *FIFO-FLEX*, i.e., the oldest task waiting in TQ will be assigned threads first, and dispatched as soon as at least `minnthreads` have been assigned; as multiple threads can become idle (almost) simultaneously, it is possible that, implementation defined, more threads, up to `maxnthreads` in total, could be allocated when the task starts execution. Further available threads will be reassigned to the next task(s). The current implementation is blocking, i.e., only one

```
#include <fork.h>
#include "forktasks.h"

#define N_A 2048  // (max) array size
sh int A[N_A];
sh Vector *s, *r;
// ... some minor details omitted

void main( void )
{
 ... // read / initialize array A
 if ($==0)
    init_tasksystem();
 barrier;
 if ($==0) {
    sptask *t;
    s = new_Vector( A, N_A );
    r = new_Vector( A, N_A ); // in-place
    t = new_stask( msort, 1, &s, r, 1, 1, 1000 );
    setREADY( s );
    spawn_task( t );  // spawn the initial task (msort)
 }
 barrier;
 join_threadpool();
 // once all work has been done, the workers return here
}
```

Figure 3: Mergesort example, the main program. The hardware thread (PRAM processor) with rank 0 initializes the task system and creates two vector containers `s` and `r` that both share the same payload array `A` of size `N_A`, for in-place sorting by the SP-task `msort` that takes `s` as input operand and `r` as output operand. After this task has been spawned, all hardware threads join the thread pool where they are assigned work. The code for the SP-tasks `msort` and `merge` can be found in Figures 4 and 5, respectively.

task can be assigned and dispatched at a time. In future extensions of this work, additional thread assignment policies such as smallest-task first or best-fit could be tried. Adaptive thread allocation as in [EKC06, KL08] could be tried as well.

Example Figures 3, 4 and 5 show an implementation of recursive parallel mergesort with explicitly parallel tasks. For mergesort there are two types of tasks required: `msort`, recursive mergesort tasks that form the divide step in the recursion tree, which create new subtasks with their containers in each instance (see Figure 4), and `merge`, the tasks forming the combine step, merging two subsolutions into one (see Figure 5). The instances of these tasks are connected by data flow edges via container objects. See the figure captions for further explanation of the code. The values used in the `new_stask()` calls for `minp` (1) and `maxp` (1) are motivated by the fact that `msort` tasks themselves do not perform much work but unfold the tree of `merge` tasks where almost all of the computational work is done. A `merge` task of size n with a (not work-optimal) fully parallel implementation can use up to $M(n) = n$ threads. In fact, this value M is a performance tuning parameter.

```
sync void msort ( sh int argc, sh Vector **argp, sh Vector *ret )
  // invariant: the Vector's are allocated by caller
{
 seq {
  Vector src = (Vector *)argp[0];  // container passed in
  int *arr = ((int *)(src->pdata));  // payload array
  int n = src->n_elems;
  if (n<=1) {
     // may call qsort(arr,n) here if threshold > 1
     YIELD_VOID( ret );  // return data in place
  }
  else {
     sptask *t1, *t2, *t3;
     Vector **s = (Vector**)shmalloc(2*sizeof(Vector*));
     Vector **r = (Vector**)shmalloc(2*sizeof(Vector*));
     s[0] = new_Vector( arr, n/2 );
     s[1] = new_Vector( &(arr[n/2]), n-n/2 );
     r[0] = new_Vector( arr, n/2 );
     r[1] = new_Vector( &(arr[n/2]), n-n/2 );
     t1 = new_stask( msort, 1, s, r[0], 1, 1, 1000 );
     t2 = new_stask( msort, 1, s+1, r[1], 1, 1, 1000 );
     setREADY( s[0] );
     setREADY( s[1] );
     spawn_task( t1 );
     spawn_task( t2 );
     // synchronization on r1 and r2 is automatic by scheduler
     t3 = new_stask( merge, 2, r, ret, 1, M(n), 1000 );
     spawn_task( t3 );  // delegates the writing of ret
}}}
```

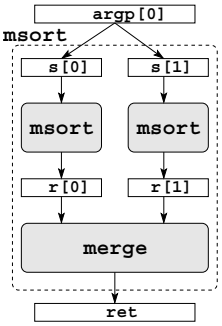

Figure 4: Mergesort example (cont.), code for the `msort` (mergesort) tasks. A `msort` task takes 1 argument, passed in `argp[0]`: the vector to be sorted. It returns the sorted vector in `ret`, with the same payload data for in-place sorting. In the `else` branch, a new level of the task graph is unfolded. Fresh vector container objects (`s[0]`, `s[1]`, `r[0]`, `r[1]`) are dynamically allocated for all intermediate operands of created subtasks, see the illustration, while the payload array space can be reused, thus avoiding copying and memory management. The data flow dependencies between the `msort` subtasks `t1`, `t2` and the `merge` subtask `t3` are given explicitly by the container references.

3 Implementation Details

A prototype of a runtime system and API has been implemented in Fork for the SBPRAM, for which we use the cycle-accurate instruction-level simulator `pramsim`.

Our implementation uses for the general case (*mixed-mode parallelism*, i.e. allowing both synchronous parallel and asynchronous sequential tasks to occur in the same application) two central, blocking, shared task queues as main data structure for the frozen queue and for the ready queue, respectively, which are implemented as bounded buffers of size $O(maxT)$ where $maxT$ is the maximum number of tasks that could be active simultaneously; this parameter can be adapted if necessary. The queue implementations make extensive use of the SBPRAM's nonblocking, constant-time multiprefix-add operations.

```
sync void merge ( sh int argc, sh Vector **argp, sh Vector *ret )
  // invariant: the Vector's are allocated by caller
{
 sh Vector *s1 = (Vector *)argp[0],  *s2 = (Vector *)argp[1];
 sh int *arr1 = (int *)(s1->pdata),  *arr2 = (int *)(s2->pdata);
 sh int n1 = s1->n_elems,            n2 = s2->n_elems;
 /* ... merge arr1 and arr2 in place, code omitted */
 seq
  YIELD_VOID( ret );  // as ret also points to arr1
}
```

Figure 5: Mergesort example (cont.), code for the `merge` tasks. Merge tasks take two input parameters, namely two vector containers passed in `argp`, pointing to two adjacent subarrays in an array where they are to be merged in-place. The result vector container points to the first subarray head (`arr1`). Once the subarrays have been merged, the `ret` vector container is advanced to ready state by `YIELD_VOID`.

Each new synchronous parallel task is, upon dispatch, allocated a new shared stack segment from global shared heap memory, which it keeps during its lifetime and releases upon termination. The code for startup and finalization of parallel tasks is (as already for ordinary Fork programs) written in SBPRAM assembler because some hardware thread (PRAM processor) registers for addressing the new shared stack segment must be saved and set up resp. restored properly.

For mixed-mode parallel applications, dispatch of data-ready tasks is, in the current prototype, serialized because the implementation needs to make sure that lower and upper bounds for allocating available threads to all ready tasks in the *FIFO-FLEX* scheduler are properly addressed as stated by each individual parallel task. Simpler (non-individual) task allocation policies or less fair dispatch schemes might allow for a more efficient, non-serializing implementation, which is a subject for future extension.

For programs that use asynchronous tasks only, we have an alternative implementation with lower overhead and a completely parallel (and non-blocking) task queue.

4 Experimental Evaluation

Sequential tasks only We use Fibonacci (the computation of the Nth Fibonacci number by the well-known recursive algorithm) as a very simple example that contains almost no computation, hence it reflects very well the overhead that is incurred by the task management system. Table 1 (left) shows runtime results taken on the SBPRAM simulator (given in thousand SBPRAM clock cycles) for $N = 17$ with different numbers of SBPRAM processors and for the two implementations of the shared task queue data structures: (i) blocking get_task and nonblocking insert, and (ii) completely non-blocking. Fib(17) recursively unfolds 7751 tasks in total, and creates 10335 operand containers. The task queue buffers were dimensioned with 8K entries each. The average overhead per task is about 2000 clock cycles with the nonblocking task queue and only slightly higher with

Table 1: Test runs for Fibonacci number calculation (left) and Mergesort (right), all times are in thousand SBPRAM clock cycles.

Overall execution time for computing the 17th Fibonacci number, creating 7751 tasks.

Hardw. Thr.	Time w. Blocking TQ (i)	Time with Non-blocking TQ (ii)
1	16086	14148
2	8158	7090
4	4466	3579
8	2839	1842
16	2209	1068
32	2120	926
64	2099	897
128	2080	893
256	2056	892

Overall execution time for Mergesort of 2048 integers, creating 6143 tasks and 8191 vectors. M denotes the choice for the upper bound `maxp` for `merge` tasks of size N.

HW Thr.	Time (blocking TQ)				
	$M = N/2$	$= N/4$	$= N/8$	$M = 1$	seq.
1	25033	25037	25039	24935	*18912*
2	13615	13118	12855	12933	*9686*
4	7189	6764	6532	6992	*5016*
8	4158	3730	3796	4312	*2990*
16	2572	2320	2425	3265	*2244*
32	2116	2006	*2000*	3104	2166
64	1896	1858	*1829*	3077	2147
128	1799	*1777*	1793	3065	2135

the blocking one. Note that program execution (and timing) on SBPRAM is completely deterministic, therefore a single test run per scenario is sufficient for the measurements.

One fundamental problem that this example reveals is that Fibonacci creates the tasks in a LIFO way, i.e., the earliest-created task is executed last of all, hence the maximum number of tasks that (could be) simultaneously alive almost equals the overall number of tasks, requiring an equally large dimensioning of the task queue data structures to avoid overflow and possibly also (too) many containers that are alive simultaneously. Similar behavior will be encountered with many divide-and-conquer algorithms, too. Where space becomes a critical resource, recursive programs thus might need to be reformulated in order to limit the amount of simultaneously alive tasks and containers.

Mergesort Our second example is the parallel mergesort program as shown above. Mergesort (`msort`) tasks are set up to use exactly 1 worker thread, and `merge` tasks use at least one and up to N workers for merging of size-N vectors. Results are shown in Table 1 (right) for a Mergesort of 2048 integers and different PRAM sizes. As 6143 tasks are generated, the average granularity is approximately 3000 instructions per task with serial ("seq.") and 4000 with parallel merging, including the dispatch overhead of about 2000 instructions. This makes also clear that the granularity is too fine for most of the tasks, as the overhead dominates. Coarsening the task granularity, e.g. by replacing spawning of light-weight `msort` tasks with inlined computation, is a way of tuning performance; this can be an issue for future work on auto-tuning optimizations.

We experimented with different choices for the upper limit M of the number of threads for `merge` calls using fully parallel merging, which is not work-optimal. M is a tuning parameter; we found empirically that e.g. $M = N/4$ and $M = N/8$ work better than $M = N$, $M = N/2$, $M = \log_2 N$ or $M = 1$. As expected, these do basically not differ in the case of a single worker thread. Using a sequential merge routine (work-optimal) leads to lower cost for small machine sizes but does not scale beyond 16 threads.

5 Related work

The synchronous parallel task concept is inspired by the join statement of Fork [KS97b, KKT01]. The main difference is that join is intended to implement synchronous parallel critical sections, so there will, at any time, be at most one instantiation of any synchronous parallel function (join body) running, while here several instances of the same SP-function could run simultaneously on disjoint thread subsets. The concept of parallel critical sections is motivated by the need of protecting certain code sections against race conditions caused by unsynchronized concurrent updates. While strict sequentialization using mutex locks is an option, the deterministic synchronous execution of PRAM systems opens for another more scalable way of avoiding race conditions. The join construct was demonstrated in Fork for parallel heap memory allocation and accelerated I/O processing [KKT01], operations that otherwise require mutual exclusion of individual threads.

The optimization of thread allocation to synchronous tasks was solved by Eriksson et al. [EKC06] for the special case where subtasks generated by recursive calls in parallel divide-and-conquer computations were executed in-line, either in parallel on disjoint thread subgroups or in serial by the entire thread group. Execution time of tasks is predicted from closed formulas that depend on problem and group size, and that are calibrated from timing data on the target machine (here, SBPRAM). Here, we generalize over this work by decoupling the subtask execution from the caller task, adding more flexibility and possibly sacrificing predictability.

StarPU [ATN10] is a run-time system for single-threaded and multi-threaded (but non-PRAM) tasks that can execute on different kinds of execution units such as CPU cores, GPUs or other programmable accelerators. In contrast to our model, StarPU does not support nested or recursive tasks. StarPU tasks are serial[3] and run on a single CPU or single GPU; support for multi-CPU OpenMP tasks is an issue of ongoing work. StarPU keeps track of each task's recent execution time history depending on input sizes, such that future decisions can be based on preditions made from collected history data.

StarSs (Star-superscalar) [PBAL09] is a family of languages and runtime systems implemented for different kinds of parallel target platforms, such as CellSs, GPUSs, OMPSs, ClusterSs. Similarly as StarPU, the StarSs model extends sequential computing by discovering and scheduling data-ready sequential tasks, which are defined by invocations of specific user functions, at run-time to some available execution unit, such as an idle CPU core, a GPU or a Cell SPU. While StarPU uses a specific API, StarSs uses language extensions to mark up task functions with their input and output parameters.

Wimmer and Träff [WT11b, WT11a] have considered multiple-thread allocation in a work-stealing scenario on distributed task queues, in order to gather a set of several threads for executing a parallel (but non-PRAM) algorithm. They use the concept of *mixed-mode parallelism* to support both task-based algorithms, such as divide and conquer, and SPMD (single program multiple data) algorithms in the same application where one task can

[3] GPU tasks in StarPU are of course internally massively parallel as they are run on many or all cores of a GPU, but to the task scheduler they look like an ordinary serial task, and the entire GPU is treated as a single resource for scheduling.

spawn other tasks. Their approach is based on classical work-stealing with independently working processors with their own queues where communication is only done when they are out of work. Wimmer and Träff organize processor groups in a binary tree topology. At level 0, each processor is in its own group; on higher levels they work together in groups of groups, called teams. Creating a team is done by work stealing in a deterministic way by visiting so-called partners on each level until work is found. The teams are needed to execute parallel tasks that require more than one thread. A team can "live" longer than a task, e.g. be used to execute tasks that need at most the number of threads available in the team. The implementation uses standard lock free data structures. Apart from not being limited to group sizes that are powers of two, a main difference from their work is that we can afford the luxury of having a central shared work queue without time penalty since we have a Combining CRCW PRAM architecture.

6 Conclusion

We have introduced and evaluated a task model for flexible dynamic scheduling and resource allocation mechanism for synchronous parallel tasks executing on a PRAM architecture. Our proof-of-concept prototype implementation shows that we can realize it with a low runtime overhead per task. It provides the option of dynamic task scheduling and thread allocation on top of a SPMD-programmed PRAM machine that was mostly designed for single-task applications. It combines the flexibility of task-based runtime systems with the power of SPMD-controlled, naturally synchronized PRAM execution within the SP-tasks.

Future work will consider high-level programming support that avoids low-level coding of calls to the run-time system API. Possible approaches include (1) high-level language constructs such as `spawn`, (2) a library of skeleton functions for frequently occuring parallel algorithmic design patterns such as *parallel divide-and-conquer*, or (3) graphical programming languages for specifying task graphs with parallel tasks (similarly to the illustration in Fig. 4) from which Fork source code can be generated automatically [KSF10]. Experiments with further thread allocation strategies and, in particular, static and dynamic autotuning of thread allocation will be considered in future work. Finally, porting our Fork-based prototype implementation to the new VTT Replica architecture and system software can be done as soon as the complete toolchain is available.

Acknowledgments This research is funded by VTT, project REPLICA, and by SeRC. We thank the anonymous reviewers for their helpful comments.

References

[ATN10] Cedric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *Proc. HPPC-2009,*

in Euro-Par 2009 Workshops, volume 6043 of *Lecture Notes in Computer Science*, pages 56–65. Springer Berlin / Heidelberg, 2010.

[EKC06] Mattias Eriksson, Christoph Kessler, and Mikhail Chalabine. Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments. In *Proc. 8th Workshop on Parallel Systems and Algorithms (PASA 2006), Frankfurt am Main, Germany, GI Lecture Notes in Informatics (LNI), vol. P-81*, pages 313–322, March 2006.

[For04] M. Forsell. Designing NOCs with a parallel extension of C. In *FDL'04*, pages 463–475, 2004.

[For10] M. Forsell. TOTAL ECLIPSE – An Efficient Architectural Realization of The Parallel Random Access Machine. *Parallel and Distributed Comput., Ed. A. Ros, IN-TECH, Wien*, pages 39–64, 2010.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Symposium on the Theory of Computation (STOC)*, pages 114–118, 1978.

[KKT01] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM Programming*. Wiley Interscience, 2001.

[KL08] Christoph Kessler and Welf Löwe. A Framework for Performance-Aware Composition of Explicitly Parallel Components. In *Proc. ParCo-2007 conference, Jülich/Aachen, Germany, Sep. 2007. In C. Bischof et al. (eds.): Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing Series, Volume 15, IOS Press*, pages 227–234, February 2008.

[KS97a] Christoph W. Keßler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. J. of Par. Programming*, 25(1):17–50, February 1997.

[KS97b] Christoph W. Keßler and Helmut Seidl. Language Support for Synchronous Parallel Critical Sections. In *Proc. APDC'97 Int. Conf. on Advances in Parallel and Distributed Computing, Shanghai, China*. IEEE CS press, March 1997.

[KSF10] Christoph W. Kessler, Wladimir Schamai, and Peter Fritzson. Platform-independent modeling of explicitly parallel programs. In *Proc. PARS'10: 23rd PARS-Workshop on parallel Systems and Algorithms, Hannover, Germany, Feb. 2010. In: M. Beigl and F. Cazorla-Almeida (Eds.): ARCS'10 Workshop Proceedings*, pages 83–93. VDE-Verlag Berlin/Offenbach, Germany, February 2010.

[PBAL09] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming With StarSs. *IJHPCA*, 23(3):284–299, 2009.

[PBB+02] Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real PRAM Programming. In *Proc. Euro-Par'02*, August 2002.

[WT11a] Martin Wimmer and Jesper Larsson Träff. Work-stealing for mixed-mode parallelism by deterministic team-building. In *23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, pages 105–115, 2011.

[WT11b] Martin Wimmer and Jesper Larsson Träff. A work-stealing framework for mixed-mode parallel applications. In *16th Int. Worksh. on Multi-threaded Architectures and Applications (MTAAP) at Int. Par. and Distr. Processing Symp. (IPDPS 2011)*, 2011.

[WV08] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proc. ACM Computing Frontiers, Ischia, Italy*, May 2008.