# Task Parallel Programming Support for the Single-Chip Cloud Computer

Andreas Prell* and Thomas Rauber

Department of Computer Science
University of Bayreuth, Germany

June 20, 2011

**Abstract**

Task parallel programming has become a popular and effective approach for programming multicore systems. Tasks execute on top of threads, which are no longer exposed but demoted to an implementation detail. This higher level of abstraction makes it easier to identify opportunities for parallelism and leave everything else to the compiler and runtime system. An interesting question is always how to support the task abstraction on a new architecture, and, what is even more important, how to arrive at an efficient implementation. In this paper, we look at Intel's Single-Chip Cloud Computer (SCC) and describe the design and implementation of a tasking environment for the SCC. We focus on the runtime system and compare two different implementations with work-sharing and work-stealing schedulers that take advantage of the processor's on-chip memory in order to achieve scalable performance.

## 1 Introduction

Task parallel programming is a promising approach with the potential to bridge the gap between parallel and mainstream computing. Tasks are implemented as an abstraction on top of threads to allow the programmer to readily express potential parallelism without worrying about the target platform. Consequently, much of the efficiency of using tasks depends on the runtime system support. As processor and thread counts keep growing, runtime system efficiency becomes a more and more important issue.

Novel processor architectures are emerging to address the increasing inefficiencies in conventional chip multiprocessors (CMPs). Important questions are thus: Can we build task abstractions that map well to such architectures? Is it possible to support applications with fine-grained irregular parallelism? And, if so, what about the expected performance?

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

The Single-Chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs to serve as a platform for manycore software research [16, 20]. The SCC is designed to be a message-passing chip where cores communicate through non-cache-coherent shared memory. In this paper, we describe our approach to support task parallel programming on the SCC. We use a task model derived from OpenMP 3.0, which we think strikes a good balance between generality, expressiveness, and potential performance [2, 6]. The central component of the tasking environment is the runtime system, which is responsible for assigning tasks to worker threads and performing load balancing. We outline two different scheduler implementations, based on work-sharing and work-stealing, and compare their performance on three microbenchmarks. The results show that efficient schedulers can be built by taking advantage of the processor's shared on-chip memory.

Much of the remaining "verbosity" of task parallel programming compared to sequential programming can be effectively hidden by a compiler. Language constructs, such as `async` and `wait`, make it straightforward to express potential parallelism without having to specify what exactly makes up a task and how one is created and enqueued. We briefly describe our initial support for task parallelism in a source-to-source compiler and point out further research that we believe will be needed to target future manycore systems.

# 2 The SCC Processor

The SCC was built with the goal of tackling the "coherency wall"—a likely scalability bottleneck in future cache-coherent CMPs with many cores [17, 20]. The SCC has no hardware cache coherence, but instead takes a more scalable approach to data-sharing, effectively making it a cluster-on-a-chip. Processor cores are interconnected by a 2D mesh network that can be used to pass data between cores. One-sided data transfers between the on-chip shared memories allow the implementation of message passing APIs for higher-level programming, such as the synchronous and asynchronous communication functions provided by RCCE and iRCCE [20].

The SCC is a tiled multicore processor [25]. Each tile contains two Pentium-class IA-32 cores (P54C), each with 16 KB L1 cache (instruction and data), 512 KB L2 cache (256 KB per core), 16 KB shared SRAM, called the Message Passing Buffer (MPB), and a Mesh Interface Unit (MIU) that connects to a router for inter-tile communication [16]. There are 24 tiles, organized in a 6x4 array with I/O and memory controllers on the periphery. In addition, each tile provides two globally accessible test-and-set registers (one per core), which can be used to implement locks needed for mutual exclusion. Locks are the only way to achieve synchronization; no atomic operations, such as CAS, are available that work across the cores of the SCC.

The SCC is an interesting platform for software research because it supports both distributed and shared memory spaces. From the point of view of the programmer, the address space is divided into:

- Private off-chip memory for each core (fully cached in L1 and L2)

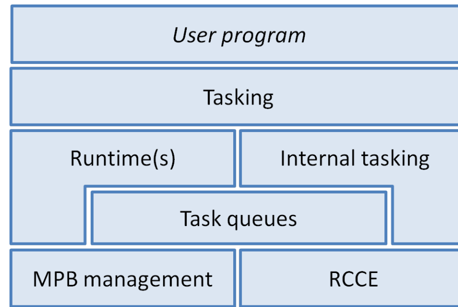



Figure 1: Components of a tasking environment for the SCC processor.

- Shared off-chip memory that can be accessed by all cores (uncacheable by default because of lack of cache coherence)
- Shared on-chip memory (of special memory type MPBT, which enables caching in L1, but bypasses L2)

# 3 Tasking on the SCC

Figure 1 shows the components of our tasking environment. At the lowest level, we use some functionality from RCCE [20], as well as our own library routines for accessing MPB memory. Worker threads are based on the RCCE notion of "units of execution" (UE), which are mapped to the cores of the chip. Each UE is assigned a rank from 0 to $N-1$, where $N$ is the number of UEs the program is running on. We use these ranks to define our worker IDs: UE 0 becomes worker 0, UE 1 becomes worker 1, and so on. We designate worker 0 as the master to run the main program, which we call the *root task*. Tasks that are spawned are descendants of root and thus *child tasks*. Note that we could designate any other worker as master, provided that the program is started on more than one UE. The root task refers to the code between the initializing and finalizing calls to `TASKING_INIT()` and `TASKING_EXIT()`. This code is only executed by the master; workers are waiting for spawned tasks and jump to `TASKING_EXIT()` after the master has reached this point and ordered termination.

Figure 2 shows the basic structure of the scheduling loop that every worker is running. Workers are either *working* or *idle*, depending on whether they successfully scheduled a task for execution. Scheduling is a two-step process in all distributed pool implementations[1]. First, a worker tries to get a task from its local queue. If the local queue is empty, the worker needs to check other queues in the system. What this means is determined by the runtime in use. A work-sharing runtime could use a single queue to keep all shared tasks in a central location. In this case, we have only one more queue

[1]We do not consider centralized pools here because of their scalability problems. A centralized pool keeps all tasks in a central location, typically in a single queue, which is accessed by all workers.

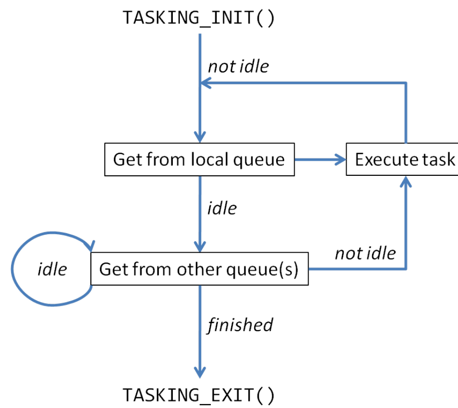



Figure 2: Basic structure of the worker scheduling loop. This structure is the same for all task pool implementations in which tasks are distributed over multiple queues.

to check. If that queue is also empty, the worker stays idle until new tasks arrive. A work-stealing runtime does not need extra queues for load balancing, but instead allows workers to steal tasks from the queues of other workers.

The shared memory on the chip makes it possible to move tasks around without going off-die, which has the potential to improve communication latency by up to a factor of 15 [16]. Before we can use the MPBs as local stores for keeping tasks and related metadata, we have to break with the "symmetric name space" model of RCCE, which restricts the use of the MPBs to message passing [20]. In this model, MPB memory is managed through collective calls, meaning that every worker must perform the same allocations and in the same order with respect to other allocations. Thus, the same buffers exist in every MPB, hence symmetric name space.

To gain full control over MPB memory, we use library routines that work with (ID, offset) tuples. Whenever a worker allocates a block from its MPB, it creates a tuple that stores the worker's own ID and the allocation's byte offset within the worker's MPB. RCCE tells us the starting address of each worker's portion of MPB memory, so ID and offset are sufficient to reconstruct the actual address of an allocation. Put and get operations are implemented using memcpy, in the same way as in RCCE, making sure that the data is actually written to or read from the target MPB instead of the L1 cache. We use synchronization flags modeled after the whole-cache-line flags in RCCE. Compressed single-bit flags and byte flags are currently not supported in our implementation.

An important part of any tasking system are the task queues that store tasks for later execution. To store tasks in MPB memory, we have implemented a double-ended queue (deque) that is similar to the work-stealing deques of [5, 10], but lock-based instead of lock-free (which is the only option on the SCC, see above). Tasks are stored in a circular array of fixed size, with *head* pointing to the oldest task. New tasks are inserted at *tail*. Producer and consumer can execute concurrently most of the time, but

if there is more than one producer that inserts tasks, all accesses require locking.

As an alternative, we could allocate one deque per tile instead of one deque per worker [19]. Having two locks per deque, we could allow producer and consumer to execute concurrently, even in the case of multiple producers. However, if a deque is always shared by two workers, inserting a task (a frequent operation!) becomes less efficient.

# 4 Runtime Systems

The runtime system, which implements the scheduler, is the central and performance determining part of the tasking environment. The scheduler is responsible for assigning tasks to worker threads and performing load balancing. It must operate as efficiently as possible in order to provide applications with scalable performance. There are two general approaches to scheduling task parallelism: work-sharing and work-stealing.

## 4.1 Work-Sharing Scheduling

Work-sharing is based on the idea of redistributing tasks to underutilized workers. A trivial implementation of a work-sharing scheduler uses a single shared queue, for example one of our deques. Our implementation is slightly different; we use both private off-chip and shared on-chip memory for storing tasks.

Workers keep their tasks in private memory while the load is balanced and they are not required to share. Tasks are stored in LIFO queues in the private memories of the workers. Since a queue is completely private and other workers have no way to access it, we can use a conventional list implementation without the need for locking. In addition, we allocate a deque in the MPB of the master thread for work-sharing. Workers that have enough tasks are responsible for moving some of their work to the deque when they find that there are only few tasks left. Workers that run out of tasks can turn to the deque to pick up new work. Note: It would be better to use a two-lock deque for work-sharing, or an entirely different data structure that supports a higher degree of concurrency (worth the effort?).

## 4.2 Work-Stealing Scheduling

In work-stealing scheduling, idle workers take the initiative to find new work. Every worker has a local deque on which it operates. Whenever a worker finds its deque empty, it attempts to steal a task from a deque of another worker, which, in the ideal case, is not disrupted by the steal. The order in which a thief visits the deques of its victims depends on the victim selection strategy. We currently support two different strategies: randomized and latency-oriented work-stealing. In randomized work-stealing, victims are chosen uniformly at random [7]. In latency-oriented work-stealing, victims are selected based on their distance in terms of (on-chip) network hops. A thief always tries to steal from the same tile, before it checks for tasks on neighboring tiles and then on tiles farther away, in increasing order of distance. The maximum number of hops on the SCC is 8, see for example [20].

## 4.3 Synchronization

Synchronization is required to coordinate the execution of tasks. We have adopted a task model similar to that of OpenMP 3.0 [2, 6] with two primary synchronization constructs, discussed in turn in the following sections.

### 4.3.1 Task Barrier

A *taskbarrier* waits for the completion of all pending tasks. This form of synchronization is coarse-grained and can only be invoked from the master thread within the root task. The call to `TASKING_EXIT()` includes a barrier, thus making sure that all tasks that were spawned during the execution of the program have completed before telling the worker threads to terminate.

A *taskbarrier* marks a global synchronization point—a point in the program where all tasks are required to complete before execution can continue past that point. This is not a barrier in the usual sense. Only the master checks into the barrier. Workers keep running their scheduling loop and eventually become idle after finishing all tasks, at which point the master is allowed to return from the barrier.

### 4.3.2 Task Wait

A *taskwait* waits for the completion of all direct children of the current task. This form of synchronization allows fine-grained control over parent-child dependencies. Note that this is not a *deep taskwait*: tasks may outlive their ancestors without proper synchronization at the parent-child level.

### 4.3.3 Future

To synchronize with particular child tasks, we provide simple explicit *futures*. Futures were first introduced in functional programming languages, such as Multilisp, a parallel dialect of Scheme [15], and are gradually making their way into mainstream concurrent programming [9, 18]. Essentially, a future is a placeholder for an asynchronous computation (that is, a placeholder for the result of a task). Forcing a future means waiting for the result to arrive, rather than waiting for the task to complete. A worker is free to pick up other tasks while a future is pending.

### 4.3.4 Implementation

The actual implementation of the synchronization constructs depends on the runtime in use. A runtime must provide the functions `RT_barrier()`, `RT_taskwait()`, and `RT_force_future()`, which are called from the tasking code. In the following discussion, we focus on the work-stealing runtime. Listings 1-3 outline the basic structure of our implementation.

When the master enters a *taskbarrier*, it doesn't simply wait but helps make progress when there is work to do. It first completes all local work and then participates in work-stealing to help execute the remaining tasks. Because a task can always create new tasks, the master must go back to checking its deque after a successful steal. When the

```
void RT_barrier(void)
{
  Task task;

empty_local_queue:
  while (MPB_task_pop(queue, &task))
    execute(&task);

  while (idle_workers() != num_workers-1) {
    if (steal(&task)) {
      execute(&task);
      goto empty_local_queue;
    }
  }
}
```

Listing 1: Implementation of *taskbarrier* in work-stealing runtime.

task pool is empty and all workers have signaled that they are idle, the master returns from the barrier.

When a worker enters a *taskwait*, it first searches for child tasks that are still in the local deque. If one or more child tasks were stolen and are still in progress, the worker tries to find other useful work by going off stealing. A *taskwait* has an associated counter to keep track of the number of children a task is waiting for. The counter is allocated from the MPB of the worker that runs the parent (the joining task) and atomically updated whenever a child is created or terminates[2]. Once the *taskwait* completes by reading a count of zero, the memory can be freed or otherwise marked as reusable.

Unrestricted stealing within *taskwait* can have a negative effect on the workers' stack size as stolen tasks execute on top of non-empty stacks. Stack space is not the only problem. Deep recursion and many pending *taskwaits* can quickly exhaust MPB memory. To avoid overflows, the runtime must take care not to allocate more MPB memory than is available. Thus, if the MPB pressure becomes too large, new tasks are executed sequentially. Tasks are created again once enough memory has been reclaimed.

When a worker forces a *future*, it first checks to see if the result of the future is already computed, and if so, it just returns the result. A future object is allocated from the MPB of the worker that creates the future. It has an embedded synchronization flag that is set once the result is ready. If the worker finds that the result is not ready, it tries to resolve the future by looking at its current children. If executing them doesn't resolve the future, it means the child representing the future was stolen. We proceed with leapfrogging and try to steal from the worker that has started to evaluate the future [26]. This form of depth-restricted work-stealing may limit the available parallelism and impact performance, but helps to keep a bound on memory consumption [14].

[2] In the current implementation, a task counter is a 64 byte object (int + flag, both padded to cache lines). Note: We should be able to at least halve the required memory if we reliably fool the write combine buffer.

```
void RT_taskwait(void *num_children)
{
  Task task;

  while (MPB_task_pop_child(queue, &task))
    execute(&task);

  while (MPB_read((MPB_int *)num_children) > 0) {
    // Unrestricted work-stealing
    if (steal(&task))
      execute(&task);
  }
}
```

Listing 2: Implementation of *taskwait* in work-stealing runtime.

```
void *RT_force_future(void *future)
{
  Task task;
  int victim;

  if (MPB_flag_is_set(MPB_flag_ptr(future))
    return future;

  while (MPB_task_pop_child(queue, &task)) {
    execute(&task);
    if (MPB_flag_is_set(MPB_flag_ptr(future)))
      return future;
  }

  victim = get_thief(future);

  while (MPB_flag_is_unset(MPB_flag_ptr(future)) {
    if (leapfrog(victim, &task))
      execute(&task);
  }

  return future;
}
```

Listing 3: Implementation of *futures* in work-stealing runtime.

# 5 Preliminary Experimental Results

We compare performance on three microbenchmarks designed to evaluate the scalability and efficiency of the runtime systems:

- Simple Producer-Consumer (SPC) benchmark: A single producer (worker ID 0) spawns $n$ consumer tasks. Consumer tasks perform some computation for time $t$. This benchmark allows us to test how many active consumers a single producer can tolerate.
- Bouncing Producer-Consumer (BPC) benchmark [13]: A variation of the producer-consumer benchmark with two kinds of tasks, producer and consumer tasks. Each producer task creates another producer task followed by $n$ consumer tasks, until a depth of $d$ is reached. Producer tasks do nothing besides spawning tasks. Consumer tasks perform some computation for time $t$. This benchmark stresses the ability of the runtime scheduler to find work and perform load balancing.
- Fibonacci-like tree-recursive benchmark: Modeled after the tree-recursive process of computing Fibonacci numbers. Each task $n \geq 2$ creates two child tasks $n-1$ and $n-2$ and waits for their completion. Leaf tasks $n < 2$ that end the recursion compute for time $t$. This benchmark includes the runtime overhead that is incurred to keep track of parent-child dependencies.

The SCC is used in the default configuration, with cores clocked at 533 MHz, routers at 800 MHz, and DDR3-800 memory. Work-sharing and work-stealing runtimes are configured to use a deque size of 10, that is, at most 10 tasks are stored per deque and MPB[3]. The work-stealing runtime uses the default randomized victim selection.

In our experiments, we focus on scaling rather than raw performance. We choose three different task sizes $t$, roughly representing fine, medium, and coarse-grained parallelism. Scalable performance under varying workloads will be critical to support applications with dynamic parallelism.

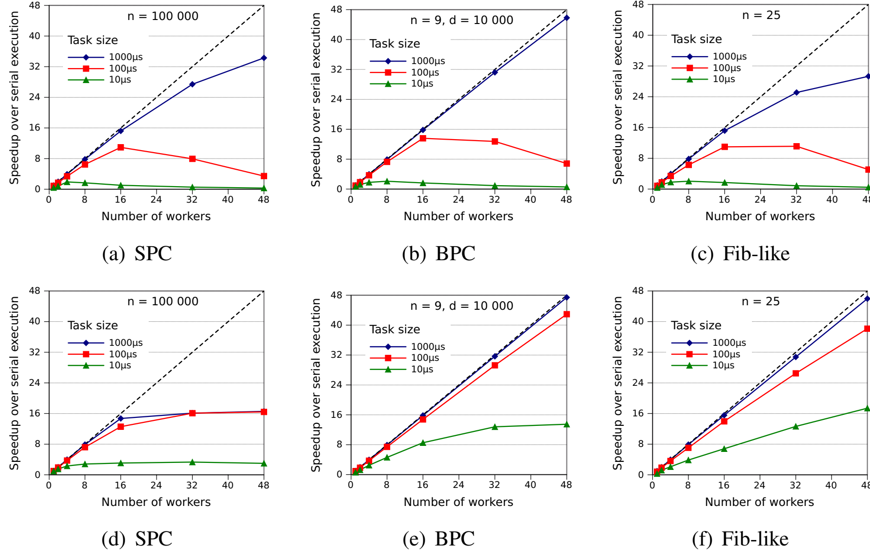
## 5.1 Work-Sharing Scheduling

The top row of Figure 3 shows the results for the work-sharing runtime. Work-sharing is only an option for relatively coarse-grained parallelism. The shared deque in the MPB of the master thread is a significant bottleneck that quickly limits scalability when workers are frequently searching for tasks. In fact, contention for the deque can become so high that performance starts to degrade. We would have to manage contention explicitly, for example by backing off before trying to access the deque again, in order to maintain the same level of performance. For very fine-grained workloads, work-sharing is not practical.

## 5.2 Work-Stealing Scheduling

The bottom row of Figure 3 shows the results for the work-stealing runtime. Work-stealing delivers good performance on medium to fine-grained parallelism. Very fine-

[3] A deque of 10 tasks takes up 1504 KB, which is almost 20% of the available MPB memory of a worker.




(a) SPC (b) BPC (c) Fib-like

(d) SPC (e) BPC (f) Fib-like

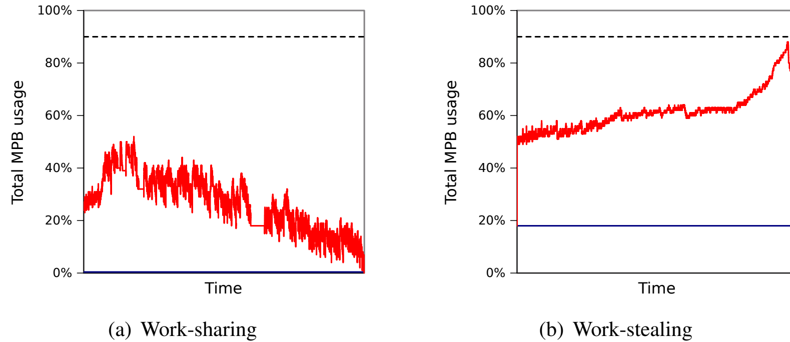
Figure 3: Performance of work-sharing (top row) and work-stealing (bottom row) runtime schedulers on SPC, BPC, and tree-recursive benchmarks. The numbers of tasks are $100\,000$ for SPC and BPC and $242\,784$ for Fib-like.

grained parallelism on the order of a few microseconds per task is hard to exploit efficiently due to the cost of task creation and scheduling. Efficiency drops to around 30% in BPC and tree-recursive benchmarks with tasks of $10\mu$s.

We see only one case where work-stealing is outperformed by work-sharing: on the SPC benchmark with large tasks. There are two reasons for the poor scaling: (1) The deques in MPB memory are fixed-size, and when full, require that new tasks are inlined and executed sequentially. The decision to inline a task is always a potential source of load imbalance, especially if the inlining worker is the only producer of tasks. (2) Unlike work-sharing, work-stealing involves probing deques, often randomly, as in this case, until a deque is found from which it is possible to steal. This probing is unnecessary in the SPC setting and likely adds to the time it takes to find new work.

It is relatively straightforward to improve work-stealing performance on SPC type of workloads. Increasing the deque size, perhaps dynamically at runtime, or trying to insert tasks in remote deques could effectively prevent producers from inlining large tasks. In cases where the single producer doesn't switch its place, directing work-stealing towards the deque of the producer could reduce consumer idle time.

The BPC benchmark shows better scaling than SPC because producer tasks can execute in parallel with other producer tasks. As a result, more parallelism is available for load balancing and there is generally less contention for work compared to SPC.




(a) Work-sharing

(b) Work-stealing

Figure 4: Total MPB usage of work-sharing and work-stealing schedulers running the tree-recursive benchmark with 48 workers. The minimum and the maximum allowed MPB usage as set by the runtime are indicated by the bottom and top lines.

### 5.3 MPB Usage

Our preliminary experiments indicate that work-stealing is much more scalable and thus practical than work-sharing. This performance advantage is not without a cost. We measured the MPB usage of both schedulers running the tree-recursive benchmark with 48 workers. One worker (worker ID 47) was excluded from the computation and used to collect the data from all the other workers. The results should tell us something about the pressure on MPB memory. Recursive *taskwaits* put a much higher pressure on the MPBs than synchronizing with *taskbarrier* because every *taskwait* requires a synchronization variable (`num_children` in Listing 2) to count the number of children. After a *taskwait* completes, the synchronization variable can be reused.

The minimum MPB usage can be obtained from the SPC or BPC benchmark (the lower bound in Figure 4). Work-sharing is more space efficient than work-stealing, primarily because it keeps only one deque compared to one deque per MPB. The tree-recursive benchmark requires that MPB memory is allocated for every pending *taskwait*. Tasks with return values (*futures*) need additional space to copy their results back to their parents. The runtime must record the available MPB memory to avoid overflows. Currently, we limit MPB usage to 90% (the upper bound in Figure 4). New tasks that would exceed this limit are not created but inlined and executed sequentially. Again, this decision could lead to some load imbalance.

## 6 Ongoing and Future Work

We consider two general directions for future work: (1) improving the programming model by adding compiler support for task parallelism and (2) exploring new load balancing strategies for future manycore processors.

## 6.1 Compiler Support for Task Parallelism

A key advantage of having compiler support for task parallelism is that it allows to shield the programmer from the details of interacting with the tasking environment. Without compiler support, setting up the tasks in a program may require quite a bit of effort. A compiler can automate that effort by inserting the necessary code for task creation and synchronization.

Language constructs, such as `async` and `wait`, make it straightforward for programmers to denote potential parallelism and synchronization between tasks. A source-to-source translation step takes care of rewriting the code and inserting appropriate calls to the runtime library. We have started to implement our ideas as a set of language extensions for the xoc compiler [11]. Xoc is a C compiler front end whose focus on extensibility makes it comparatively easy to prototype new language constructs. Listing 4 shows how async function calls are translated into tasks. Depending on the runtime in use, the compiler generates different code to implement the task creation routine.

Language-level support for parallel loops can be built on top of these translations. Because iterations of a parallel loop tend to be fine-grained, an important performance optimization would be to bundle multiple iterations into a single task at runtime.

## 6.2 Load Balancing Strategies

The hybrid nature of the SCC makes it an interesting platform for software research. There are many ways to load-balance a computation on a processor that supports both distributed and shared memory spaces. Interesting questions include: Which runtime strategies are the most efficient in the context of the SCC? Which strategies are general enough that they can be adapted to work in distributed environments, such as clusters of manycore processors? And what must be done to achieve global load balancing?

To answer these questions, it will be important to explore alternative runtime systems that differ from the common work-sharing and work-stealing approaches that were originally designed for shared-memory parallelism. One such runtime system could be based on partitioning the chip and assigning each partition to a manager thread that is responsible for load balancing within its partition and for communicating with the managers of other partitions. Worker threads could notify their managers when they are idle (or better, ahead of time) and wait for tasks to arrive. Such a scheduler could be adapted for distributed systems in which workers have no way to access the task queues of other workers.

Similar message-passing schedulers have been proposed in [23]. Given fast messaging between threads, efficient schedulers can be built using only private task queues.

# 7 Related Work

The importance of task parallel programming becomes evident in the wealth of work going on to improve the status quo of mainstream parallel programming. Cilk has explored many runtime techniques that are of importance today, first and foremost the scheduling of tasks by work-stealing [7]. The most prominent and widely used

```
int do_sth(int a, int b);

z = async do_sth(x, y);
...
wait z;

// is translated into
// a struct for the argument values
struct do_sth_task_data {
  int a, b;
};

// a wrapper function to execute the task
void do_sth_task_func(struct do_sth_task_data *data)
{
  int tmp0 = data->a;
  int tmp1 = data->b;
  int tmp2 = do_sth(tmp0, tmp1);

  // Update future with tmp2
  // ...
}

// a wrapper function to create the task
// or inline it if that is required
void do_sth_async(int x, int y, MPB_int_with_flag *f)
{
  Task task;
  struct do_sth_task_data *data;

  // Fill in argument values
  data = (struct do_sth_task_data *)task.data;
  data->a = a;
  data->b = b;

  // Setup task metadata
  // ...

  // Try to enqueue task
  // ...
}

// and the rewritten code at the call site
MPB_int_with_flag *__z = MPB_malloc(sizeof(*__z));
do_sth_async(x, y, __z);
...
z = *(int *)RT_force_future(__z);
MPB_free(__z);
```

Listing 4: Compiler support for async function calls.

libraries for task parallel programming include Java Concurrency Utilities (JUC) [21], Intel's Threading Building Blocks (TBB) [22], Microsoft's Task Parallel Library (TPL) for .NET [18], and Apple's Grand Central Dispatch (GCD) [3].

OpenMP 3.0 has started to shift its focus from threads to tasks to better handle unstructured parallelism [6]. The task model we describe in this paper is similar to that of OpenMP[4]. In fact, our implementation could serve as a starting point for porting OpenMP to the SCC.

The High Productivity Computing Systems (HPCS) languages Chapel [12], Fortress [4], and X10 [24] 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 `forall` loops, are implemented on top of tasks.

Other languages introduce task-like concepts to express concurrency, such as the goroutines in the Go Programming Language [1]. A goroutine is essentially a task; it defines a function that may execute in parallel with the code that called the goroutine. Whether a thread is spawned for each goroutine or whether goroutines are mapped to long-lived threads is an implementation detail and not something the user needs to be concerned with.

X10 and Habanero-C, a research language derived from an earlier version of X10, are in the process of being ported to the SCC [8, 19]. Preliminary results are in line with our findings that work-stealing works well on the SCC.

# 8 Conclusions

In this paper, we have described a tasking environment that enables task parallel programming on the SCC experimental processor. We have shown that efficient schedulers can be built by taking advantage of the processor's shared on-chip memory. One problem that remains is the rather limited size of this memory. Even if we limit the number of tasks in advance, the runtime system may take up a significant portion of the available memory, leaving little space for other uses. We continue to search for ways to reduce this memory pressure, without sacrificing too much on the performance side.

Tasks are becoming an important abstraction for parallel programming. But the future of parallelism is far from clear. Task abstractions and their implementations will have to continue to evolve for some time to come.

# References


[1] The Go Programming Language. `http://golang.org`.

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


[4] More precisely, our task model is similar to that of *tied* tasks in OpenMP. We don't support *untied* tasks. An untied task can be suspended and resume execution on a different worker than the one that started the task.

[3] Grand Central Dispatch. Technology Brief, June 2009.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification, Version 1.0. `http://projectfortress.sun.com`, March 2008.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[6] Eduard Ayguadé et al. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46:720–748, September 1999.

[8] Keith Chapman, Ahmed Hussein, and Antony L. Hosking. X10 on the Single-Chip Cloud Computer. In *Proceedings of the ACM SIGPLAN 2011 X10 Workshop*, X10 '11, 2011.

[9] Philippe Charles et al. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[10] David Chase and Yossi Lev. Dynamic Circular Work-Stealing Deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[11] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an Extension-Oriented Compiler for Systems Programming. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 244–254, New York, NY, USA, 2008. ACM.

[12] Cray Inc. Chapel Language Specification, Version 0.8. `http://chapel.cray.com`, April 2011.

[13] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[14] K.-F. Faxen. Efficient Work Stealing for Fine Grained Parallelism. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 313 –322, September 2010.

[15] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[16] Jason Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*, pages 108–109, 2010.

[17] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob van der Wijngaart. The Case for Message Passing on Many-Core Chips. Technical report, University of Illinois Urbana-Champaign, 2010.

[18] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[19] Deepak Majeti. Lightweight Dynamic Task Creation and Scheduling on the Intel Single Chip Cloud (SCC) Processor. In *Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, PLACES '11, pages 35–42, 2011.

[20] Timothy G. Mattson et al. The 48-core SCC Processor: the Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[21] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

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

[23] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 311–322, New York, NY, USA, 2010. ACM.

[24] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 Language Specification, Version 2.2. `http://x10.codehaus.org`, May 2011.

[25] Michael B. Taylor et al. Tiled Multicore Processors. In Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, editors, *Multicore Processors and Systems*, pages 1–33. Springer Science+Business Media, LLC, 2009.

[26] David B. Wagner and Bradley G. Calder. Leapfrogging: A Portable Technique for Implementing Efficient Futures. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM.