HAL
open science

Go's Concurrency Constructs on the SCC

Andreas Prell, Thomas Rauber

To cite this version:

Andreas Prell, Thomas Rauber. Go's Concurrency Constructs on the SCC. The 6th Many-core Applications Research Community (MARC) Symposium, Jul 2012, Toulouse, France. pp.2-6. hal-00718924

HAL Id: hal-00718924

https://hal.science/hal-00718924

Submitted on 18 Jul 2012

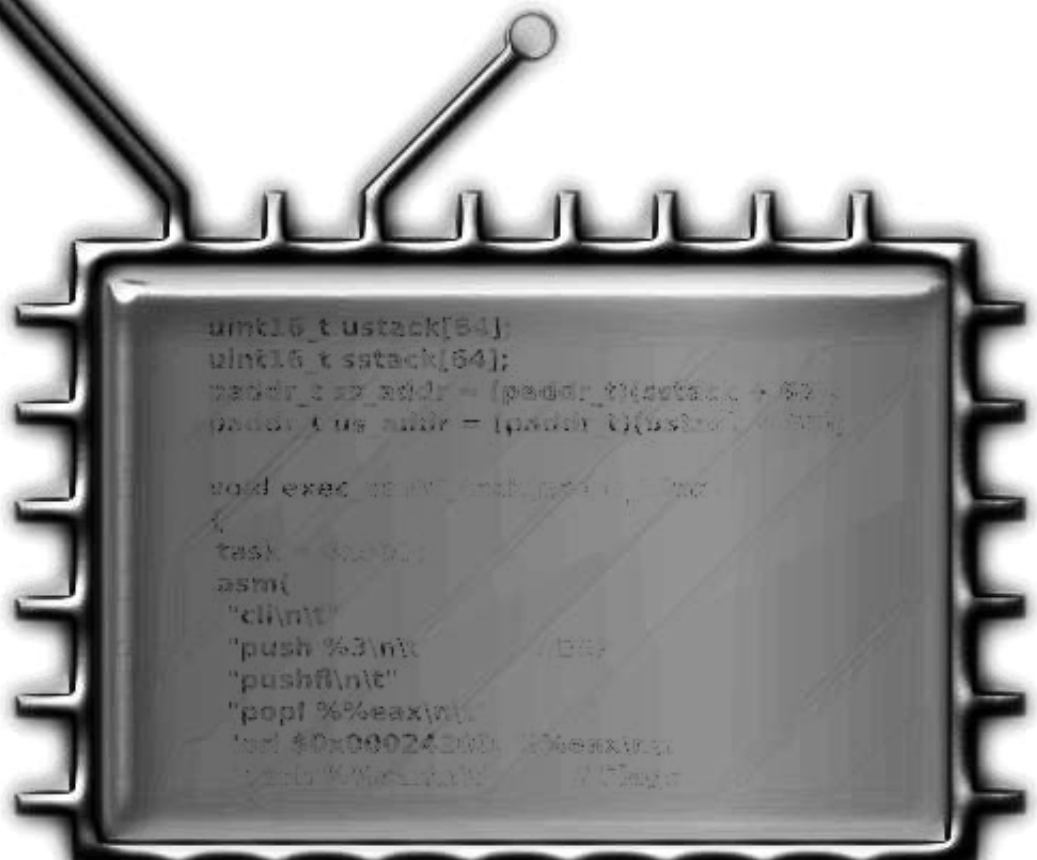
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA

THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

Go's Concurrency Constructs on the SCC

Andreas Prell and Thomas Rauber
Department of Computer Science
University of Bayreuth, Germany
{andreas.prell,thomas.rauber}@uni-bayreuth.de

***Abstract*—We present an implementation of goroutines and channels on the SCC. Goroutines and channels are the building blocks for writing concurrent programs in the Go programming language. Both Go and the SCC share the same basic idea—the use of messages for communication and synchronization. Our implementation of goroutines on top of tasks reuses existing runtime support for scheduling and load balancing. Channels, which permit goroutines to communicate by sending and receiving messages, can be implemented efficiently using the on-die message passing buffers. We demonstrate the use of goroutines and channels with a parallel genetic algorithm that can utilize all cores of the SCC.**

I. INTRODUCTION

Go is a general-purpose programming language intended for systems programming [1]. We leave out a general description of Go, and rather focus on its support for concurrent programming, which is not the usual "threads and locks", even if threads and locks are still used under the covers. Programmers are encouraged to "share memory by communicating", instead of to "communicate by sharing memory". This style of programming is reminiscent of message passing, where messages are used to exchange data between and coordinate execution of concurrently executing processes. Instead of using locks to guard access to shared data, programmers are encouraged to pass around references and thereby transfer ownership so that only one thread is allowed to access the data at any one time.

Go's way of thinking is also useful when programming Intel's Single-Chip Cloud Computer (SCC) research processor. The SCC is intended to foster manycore software research, using a platform that's more like a "cluster-on-a-chip" than a traditional shared-memory multiprocessor. As such, the SCC is tuned for message passing rather than for "threads and locks". Or as Intel Fellow Jim Held commented on the lack of atomic operations: "In SCC we imagined messaging instead of shared memory or shared memory access coordinated by messages. [...] Use a message to synchronize, not a memory location." [2], [3] So, we think it's reasonable to ask, "Isn't Go's concurrency model a perfect fit for a processor like the SCC?" To find out, we start by implementing the necessary runtime support on the SCC.

II. CONCURRENCY IN THE GO PROGRAMMING LANGUAGE

Go's approach to concurrency was inspired by previous languages that came before it, namely Newsqueak, Alef, and Limbo. All these languages have in common that they built on Hoare's Communicating Sequential Processes (CSP), a formal language for writing concurrent programs [4]. CSP introduced the concept of channels for interprocess communication (not in the original paper but in a later book on CSP, also by Hoare [5]). Channels in CSP are synchronous, meaning that sender and receiver synchronize at the point of message exchange. Channels thus serve the dual purpose of communication *and* synchronization. Synchronous or unbuffered channels are still the default in Go (when no buffer size is specified), although the implementation has evolved quite a bit from the original formulation and also allows asynchronous (non-synchronizing) operations on channels.

Go's support for concurrent programming is based on two fundamental constructs, goroutines and channels, which we describe in the following sections.

A. Goroutines

Think of goroutines as lightweight threads that run concurrently with other goroutines as well as the calling code. Whether goroutines run in separate OS threads or whether they are multiplexed onto OS threads is an implementation detail and something the user should not have to worry about. A goroutine is started by prefixing a function call or an anonymous function call with the keyword **go**. The language specification says: "A **go** statement starts the execution of a function or method call as an independent concurrent thread of control, or goroutine, within the same address space." [6] In other words, a **go** statement marks an asynchronous function call that creates a goroutine and returns without waiting for the goroutine to complete. So, from the point of view of the programmer, goroutines are a way to specify concurrently executing activities; whether they are allocated to run in parallel is determined by the system.

B. Channels

In a broader sense, channels are used for interprocess communication. Processes can send or receive messages over channels or synchronize execution using blocking operations. In Go, "a channel provides a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type." [6] Go provides both unbuffered and buffered channels. Channels are first-class objects (a distinguishing feature of the Go branch of languages, starting with Newsqueak): they can be stored in variables, passed as arguments to functions, returned from functions, and sent themselves over channels. Channels are also typed, allowing the type system to catch programming errors, like trying to send a pointer over a channel for integers.

 ISBN: 978-2-7257-0016-8

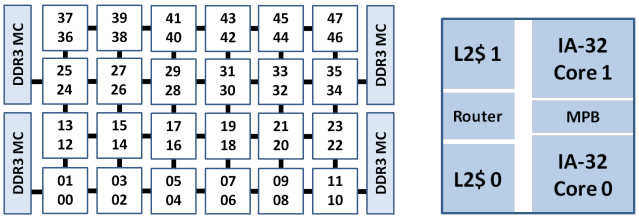


Fig. 1. The 48-core SCC processor: 6x4 tile array (left), 2-core tile (right)

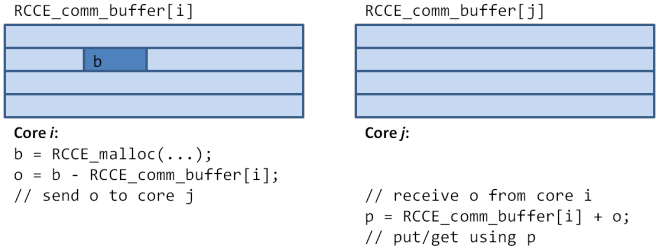


Fig. 2. Accessing buffers in remote MPBs without requiring collective allocations. Because buffer b is not replicated in other cores' MPBs, offset o can't be determined implicitly (as in RCCE_put/RCCE_get), but must be passed between cores.

III. A GLIMPSE OF THE FUTURE? THE SCC PROCESSOR

The Single-Chip Cloud Computer (SCC) is the second processor developed as part of Intel's Tera-scale Computing Research Program, which seeks to explore scalable manycore architectures along with effective programming techniques. At a high level, the SCC is a 48-core processor with a noticeable lack of cache coherence between cores. It does support shared memory, both on-chip and off-chip, but it's entirely the (low-level) programmer's responsibility to avoid working on stale data from the caches—if caching is enabled at all. In the default configuration, most system memory is mapped as private, turning the SCC into a "cluster-on-a-chip", programmed in much the same way as an ordinary cluster.

Message passing between cores is enabled by the inclusion of 16 KB shared SRAM per tile, called Message Passing Buffer (MPB). Programmers can either use MPI or RCCE, a lightweight message passing library tuned to the features of the SCC [7], [8]. RCCE has two layers: a high-level interface, which provides send and receive routines without exposing the underlying communication, and a low-level interface, which allows complete control over the MPBs in the form of one-sided put and get operations—the basic primitives to move data around the chip. RCCE also includes an API to vary voltage and frequency within domains of the SCC, but we won't go into power management issues here.

IV. GO'S CONCURRENCY CONSTRUCTS ON THE SCC

RCCE's low-level interface allows us to manage MPB memory, but with an important restriction. RCCE uses what it calls a "symmetric name space" model of allocation, which was adopted to facilitate message passing. MPB memory is managed through collective calls, meaning that every worker[1] must perform the same allocations/deallocations and in the same order with respect to other allocations/deallocations. Thus, the same buffers exist in every MPB, hence symmetric name space. If we want to make efficient use of channels, we must break with the symmetric name space model to allow every worker to allocate/deallocate MPB memory at any time.

Suppose worker i has allocated a block b from its MPB and wants other workers to access it (see also Figure 2). How can we do this? RCCE tells us the starting address of each worker's portion of MPB memory via the global variable RCCE_comm_buffer. Thus, worker j can access any location in i's MPB by reading from or writing to addresses RCCE_comm_buffer[i] through RCCE_comm_buffer[i] + 8191. Note that in the default usage model, the 16 KB shared SRAM on a tile is equally divided between the two cores. Worker j then needs to know the offset of b within i's MPB. This offset is easily determined on i's side, and after communicating it to worker j, j can get a pointer to b and use this pointer to access whatever is stored at this address. To summarize, any buffer can be described by a pair of integers (ID, offset), which allows us to abandon collective allocations and to use the MPBs more like local stores.

A. Goroutines as Tasks

We have previously implemented a tasking environment to support dynamic task parallelism on the SCC [9]. Specifically, we have implemented runtime systems based on work-sharing and work-stealing to schedule tasks across the cores of the chip. If we map a goroutine to a task, we can leave the scheduling to the runtime system, load balancing included. Scheduling details aside, what **go** func(a,b,c); then does is create a task to run function func using arguments a, b, and c, and enqueue the task for asynchronous execution. Tasks are picked up and executed by worker threads. Every worker thread runs a scheduling loop where it searches for tasks (the details depend on which runtime is used). One thread, which we call the master thread, say, thread 0, is designated to run the main program between the initializing and finalizing calls to the tasking environment. This thread can call goroutines, but it cannot itself schedule goroutines for execution. In addition to the master thread, we need one or more worker threads to be able to run goroutines. This is currently a restriction of our implementation.

Figure 3 shows a pictorial representation of workers running goroutines. Assume we start a program on three cores—say, core 0, core 1, and core 2—there will be a master thread and two worker threads, each running in a separate process. Worker threads run goroutines as coroutines to be able to yield control from one goroutine to another [10]. While a goroutine shares the address space with other goroutines on the same worker, goroutines on different workers also run in different address spaces (sharing memory is possible). This is a deviation from the Go language specification, which states that goroutines run concurrently with other goroutines, *within the same address space*. What we need is a mechanism to allow goroutines

[1]Worker means process or thread in this context. RCCE uses yet another term—unit of execution (UE). On the SCC, we assume one worker per core.

 ISBN: 978-2-7257-0016-8

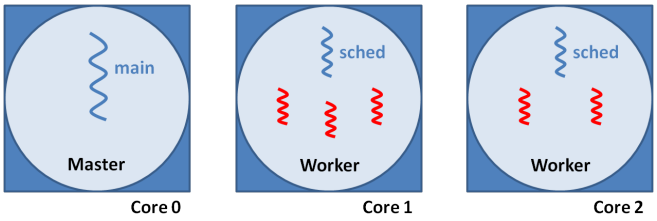


Fig. 3. An example execution of a program with goroutines on three cores of the SCC processor. In this example, worker 1 is running three goroutines, while worker 2 is running two goroutines. The master thread can call goroutines but not run them. Worker threads switch goroutines as needed, whenever a goroutine blocks on a channel.

to communicate, regardless on which core they are running. Channels in shared memory provide such a mechanism.

B. Channels

Our channel implementation takes advantage of the SCC's on-chip memory for inter-core communication. A channel is basically a blocking FIFO queue. Data items are stored in a circular array, which acts as the channel's internal buffer. Channel access must be lock-protected because the SCC lacks atomic operations and only provides one test-and-set register per core for the purpose of mutual exclusion.

A buffered channel of size n has an internal buffer to store n data items (the internal buffer has actually $n+1$ slots to make it easier to distinguish between an empty and a full buffer). If the buffer is full and another item is sent to the channel, the sender blocks until an item has been received from the channel. An unbuffered channel, which is the default in Go when no size is given, is implemented as a buffered channel with an internal buffer to store exactly one data item. Unlike a send to a buffered channel, however, a send to an unbuffered channel blocks the sender until a receive has happened.

A channel (unbuffered) takes up at least 160 bytes—four cache lines to hold the data structure, plus a cache line of internal channel buffer. When we try to write a concurrent program such as the prime sieve presented in the Go language specification [6] and in the Go tutorial [11], we must be aware of channel related memory leaks that can quickly exhaust the MPBs. Go on the other hand, is backed by a garbage collector, which reclaims memory behind the scenes, and which, according to Rob Pike, is in fact "essential to easy concurrent programming" [12]. What the current implementation doesn't include are functions to close a channel and to peek at a set of channels simultaneously (Go's **select** statement, which is like a **switch** statement for channel operations).

C. Channel API

The basic channel API consists of four functions:

Channel *channel_alloc(**size_t** sz, **size_t** n);
Allocates a channel for elements of sz bytes in MPB memory. If the number of elements n is greater than zero, the channel is buffered. Otherwise, if n is zero, the channel is unbuffered. Note that, unlike in Go, channels are untyped. It would be perfectly okay to pass values of different types over a single channel, as long as they fit into sz bytes. Also note that the current implementation does not allow all combinations of sz and n. This is because the underlying allocator (part of RCCE) works with cache line granularity, so we have to make sure that channel buffers occupy multiples of 32 bytes ((n+1)*sz must be a multiple of 32).

void channel_free(Channel *ch);
Frees the MPB memory associated with channel ch.

bool channel_send(Channel *ch, **void** *data, **size_t** sz);
Sends an element of sz bytes at address data to channel ch. The call blocks until the element has been stored in the channel buffer (buffered channel) or until the element has been received from the channel (unbuffered channel).

bool channel_receive(Channel *ch, **void** *data, **size_t** sz);
Receives an element of sz bytes from channel ch. The element is stored at address data. The call blocks if the channel is empty.

Additionally, we find the following functions useful:

int channel_owner(Channel *ch);
Returns the ID of the worker that allocated and thus "owns" channel ch.

bool channel_buffered(Channel *ch);
Returns true if ch points to a buffered channel, otherwise returns false.

bool channel_unbuffered(Channel *ch);
Returns true if ch points to an unbuffered channel, otherwise returns false.

unsigned int channel_peek(Channel *ch);
Returns the number of buffered items in channel ch. When called with an unbuffered channel, a return value of 1 indicates that a sender is blocked on the channel waiting for a receiver.

unsigned int channel_capacity(Channel *ch);
Returns the capacity (buffer size) of channel ch (0 for unbuffered channels).

D. Go Statements

Go statements must be translated into standard C code that interfaces with our runtime library. Listing 1 gives an idea of the translation process. Suppose we start a goroutine running function f(in, out), which operates on two channels in and out. For every function that is started as a goroutine, we generate two wrapper functions, one for creating and enqueuing the goroutine (task), the other for running it after scheduling. Listing 1 shows the corresponding code in slightly abbreviated form. Function go_f creates a task saving all the goroutine's arguments and enqueues the task for asynchronous

 ISBN: 978-2-7257-0016-8

```
void f(Channel *in, Channel *out);

// The go statement
go f(in, out);

// is rewritten into
go_f(in, out);

// with the following definition
// (some details are left out for brevity)
void go_f(Channel *in, Channel *out)
{
  Task task;
  f_task_data *td;

  task.fn = (void (*)(void *)f_task_func;
  td = (f_task_data *)task.data;
  td->in_owner = channel_owner(in);
  td->in_offset = MPB_data_offset(td->in_owner, in);
  td->out_owner = channel_owner(out);
  td->out_offset = MPB_data_offset(td->out_owner, out);

  // Enqueue task
}

// The data structure to hold the goroutine's arguments
typedef struct f_task_data {
  int in_owner, in_offset;
  int out_owner, out_offset;
} f_task_data;

// is passed to the task function that wraps the call to f
void f_task_func(f_task_data *args)
{
  Channel *in, *out;

  in = MPB_data_ptr(args->in_owner, args->in_offset);
  out = MPB_data_ptr(args->out_owner, args->out_offset);
  f(in, out);
}
```

Listing 1: A **go** statement and the translation into tasking code.

execution. Channel references are constructed from channel owner and MPB offset pairs (required to break with the collective allocations model of RCCE, described above), so each channel is internally represented by two integers. The helper functions MPB_data_offset and MPB_data_ptr calculate offsets and pointers based on the MPB starting addresses in RCCE_comm_buffer. The task function f_task_func is called by the runtime when the task is scheduled for execution, after which the goroutine is up and running.

V. EXAMPLE: PARALLEL GENETIC ALGORITHM

To demonstrate the use of goroutines and channels, we have written a parallel genetic algorithm (PGA) that can utilize all the cores of the SCC. We follow the island model and evolve a number of populations in parallel, with occasional migration of individuals between neighboring islands [13].

We choose a simple toy problem: evolving a string from random garbage in the ASCII character range between 32 and 126. More precisely, we want to match the following string that represents a simple "Hello World!" program:

> "#include <stdio.h> int main(void) { printf("Hello SCC!\n"); return 0; }"

The fitness of a string is calculated based on a character by character comparison with the target string, according to $f = \sum_{i=0}^{n}(\text{target}[i] - \text{indiv}[i])^2$, where n is the length of both strings. Thus, higher fitness values correspond to less optimal

```
void evolve(Channel *chan, Channel *prev_in,
  Channel *prev_out, int n, int num_islands)
{
  GA_pop *island;
  Channel *in, *out;

  if (n < num_islands - 1) {
    in = channel_alloc(sizeof(GA_indv), 0);
    out = channel_alloc(sizeof(GA_indv), 0);
    go evolve(chan, in, out, n + 1, num_islands);
  }

  island = GA_create(island_size, target);
  while (GA_evolve(island, migration_rate))
    migrate(island, n, prev_in, prev_out, in, out);
  channel_send(chan, &island->indvs[0], sizeof(GA_indv));
  GA_destroy(island);
}
```

Listing 2: Populations evolve concurrently in goroutines.

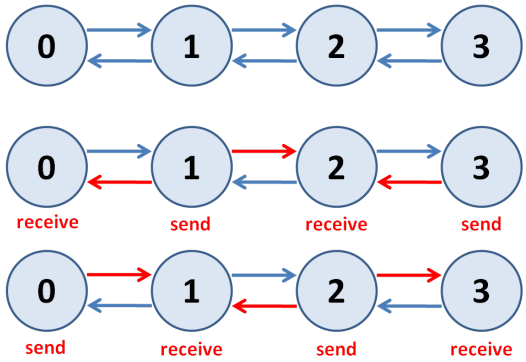


Fig. 4. An example of migration between four islands of a parallel genetic algorithm. Individuals are exchanged over channels in two steps: (1) odd numbered islands send to even numbered neighbors, and (2) even numbered islands send to odd numbered neighbors.

solutions, and our goal is to find the string with the fitness value 0.

Listing 2 shows the code to evolve an island. The GA_* procedures to create, evolve, and destroy a population are not specific to the SCC but part of our generic GA implementation.[2] Islands are created one after another, each in a new goroutine. The main thread of the program starts the evolution by allocating a channel chan, on which the solution will be delivered, and creating the first goroutine with

```
go evolve(chan, NULL, NULL, 0, num_islands);
```

Because the main thread cannot run goroutines, it will block until the evolution has finished when it attempts to receive from chan.

After every migration_rate generations, we migrate two individuals that we pick at random from the current population to neighboring islands. To exchange individuals between two islands a and b, we need two channels: one for sending individuals from a to b, the other vice versa for sending individuals from b to a. Every island other than the first and last has two neighbors and, thus, four channel references to communicate with its neighbors.

[2]The core of the GA uses tournament selection, one-point crossover of selected individuals, and random one-point mutation of offspring individuals.

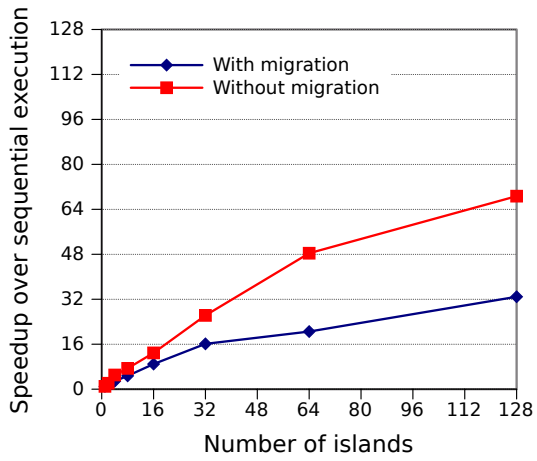


Fig. 5. Speedup with multiple islands over the sequential execution with one island. The total population across all islands has a size of 1280 (island size = 1280 / number of islands). The migration rate is two individuals every ten generations.

Figure 4 illustrates what happens during migration. The example shows four islands numbered from 0 to 3 and the channels between them. Migration is a two-step process. We use unbuffered, synchronizing channels, which require a rendezvous between sender and receiver. First, odd numbered islands send their individuals, while even numbered islands receive in the matching order. Then the process of sending and receiving is reversed.

Figure 5 shows the speedups we have measured for a fixed population size of 1280.[3] Sequential execution refers to the case where we evolve only one island, so everything runs inside a single goroutine. When the number of goroutines exceeds the number of available worker threads, goroutines are multiplexed onto worker threads. Up to the total number of cores (48), we make sure that every new goroutine runs on a separate core, so that goroutines actually run in parallel (though the runtime doesn't allow us to control on which core a goroutine starts execution). Creating more goroutines than we have cores is no problem; the runtime scheduler switches between goroutines whenever active goroutines block on channels during migration.

The toy problem is simple enough that we don't actually need a sophisticated GA that migrates individuals between islands in order to maintain genetic diversity. The same algorithm leaving out migration and instead just switching between goroutines after every migration_rate generations achieves even higher speedups.

VI. CONCLUSION

We have presented an implementation of goroutines and channels, the building blocks for concurrent programs in the Go programming language. Both Go and the SCC share the basic idea of communicating and synchronizing over messages rather than shared memory. The Go slogan "Do not communicate by sharing memory; instead, share memory by communicating" is a good one to keep in mind when programming the SCC. Communication over channels is akin to message passing, but channels are much more flexible in the way they serve to synchronize concurrently executing activities.

Channels can be implemented efficiently using the available hardware support for low-latency messaging. However, problems are likely the small size of the on-chip memory and the small number of test-and-set registers. The size of the MPB (basically 8 KB per core) limits the number of channels that can be used simultaneously, as well as the size and number of data items that can be buffered on-chip. With only 48 test-and-set registers at disposal, allocating many channels can increase false contention because the same test-and-set locks are used for several unrelated channels. As a result, communication latency can suffer. We could support a much larger number of channels in shared off-chip memory, trading off communication latency, but frequent access to shared off-chip DRAM could turn into a bottleneck by itself.

[3]SCC in default configuration: cores running at 533 MHz, mesh and DDR memory at 800 MHz (Tile533_Mesh800_DDR800).

ACKNOWLEDGMENT

We thank Intel for granting us access to the SCC as part of the MARC program. Our work is supported by the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] "The Go Programming Language," http://golang.org. [Online]. Available: http://golang.org

[2] "Many-core Applications Research Community," http://communities.intel.com/message/113676#113676. [Online]. Available: http://communities.intel.com/message/113676#113676

[3] "Many-core Applications Research Community," http://communities.intel.com/message/115657#115657. [Online]. Available: http://communities.intel.com/message/115657#115657

[4] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[5] ——, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[6] "The Go Programming Language Specification," http://golang.org/doc/go_spec.html. [Online]. Available: http://golang.org/doc/go_spec.html

[7] T. G. Mattson, R. F. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[8] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight Communications on Intel's Single-Chip Cloud Computer Processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[9] A. Prell and T. Rauber, "Task Parallelism on the SCC," in *Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium*, ser. MARC 3. KIT Scientific Publishing, 2011, pp. 65–67.

[10] R. S. Engelschall, "Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267724.1267744

[11] "A Tutorial for the Go Programming Language," http://golang.org/doc/go_tutorial.html. [Online]. Available: http://golang.org/doc/go_tutorial.html

[12] "Go Emerging Languages Conference Talk," http://www.oscon.com/oscon2010/public/schedule/detail/15299, July 2010. [Online]. Available: http://www.oscon.com/oscon2010/public/schedule/detail/15299

[13] E. Cantú-Paz, "A Survey of Parallel Genetic Algorithms," *Calculateurs Paralleles, Reseaux et Systems Repartis*, vol. 10, 1998.