Low-Overhead Barrier Synchronization for OpenMP-like Parallelism on the Single-Chip Cloud Computer

Hayder Al-Khalissi*, Andrea Marongiu† and Mladen Berekovic‡
*,‡ Chair for Chip-Design for Embedded Computing
TU Braunschweig, Germany
Email: {alkhalissi, berekovic}@c3e.cs.tu-bs.de
†DEIS - University of Bologna
40136 Bologna - Italy
Email: a.marongiu@unibo.it

***Abstract*—To simplify program development for the Single-chip Cloud Computer *(SCC)* it is desirable to have high-level, shared memory-based parallel programming abstractions (e.g., OpenMP-like programming model). Central to any similar programming model are barrier synchronization primitives, to coordinate the work of parallel threads. To allow high-level barrier constructs to deliver good performance, we need an efficient implementation of the underlying synchronization algorithm. In this work, we consider some of the most widely used approaches for barrier synchronization on the SCC, which constitutes the basis for implementing OpenMP-like parallelism. In particular, we consider optimizations that leverage SCC-specific hardware support for synchronization, or its explicitly-managed memory buffers. We provide a detailed evaluation of the performance achieved by different approaches.**

***Index Terms*—Barrier synchronization, System-on-Chip, Manycores, OpenMP, Performance Evaluation.**

I. Introduction

It is nowadays clear that the huge number of transistors that can be integrated on a single chip (1 billion today and continuously growing) can no longer be effectively utilized by traditional single-processor designs. Multi-core technology has been succesfully adopted for the past seven years, and is currently in the many-core era, where hundreds of tightly-coupled simple processor cores are integrated in the same on-chip system.

As the complexity of *systems-on-chip* (SoCs) continues to increase, it is no longer possible to ignore the challenges caused by the convergence of software and hardware development [1]. In particular, the challenge for effective programming models at these scales is renewed, as the software stack is nowadays responsible for making effective use of the tremendous peak performance that these systems can deliver, assuming that all the processors can be kept busy most of the time.

Shared memory-based programming models have proven to be very effective at simplifying application development, since they provide the notion of a global address space, to which programmers are accustomed. OpenMP [2] has emerged as a *de-facto* standard for shared memory programming, since it provides very simple means to expose parallelism in a standard C (or C++, or Fortran) application, based on code annotations (compiler directives). This appealing ease of use has recently led to the flourishing of a number of OpenMP implementation for embedded MPSoCs [3] [4] [5] [6] [14].

OpenMP (and most related shared memory-based programming models) relies on a *fork-join* execution model, which leverages a *barrier* construct to synchronize parallel threads. Barriers – implicit or explicit – are central constructs to the OpenMP execution model and to any shared memory parallel program.

With the longer-term goal of supporting full-OpenMP parallelism, in this paper we present a study of several implementations of OpenMP-like barrier algorithms for the *Single-Chip Cloud Computer (SCC)*. The aim of this work is to gain insight into the behavior of different barrier algorithms in the OpenMP context in order to determine which of them is most appropriate for a given scenario. In particular, we consider barrier optimizations that leverage SCC-specific hardware support for synchronization, or its explicitly-managed portion of the memory hierarchy (i.e., message passing buffers). Our experimental results section provides a detailed evaluation of the performance achieved by different approaches.

The rest of the paper is organized as follows. The specification of the target system SCC is presented in Section II. Section III describes the target barrier model within the OpenMP fork-join mechanism. Different barrier algorithms are described in Section IV. Section V discusses the experimental results. Finally, our conclusion and future works are given in Section VI.

II. Target Description

The SCC [8], has been designed to explore the future of many-core computing by Intel. The architecture of the SCC resembles a small cluster or “cloud” of computers. The Intel SCC is composed of 48 independent Pentium cores, each with 16KB data and program caches and 256KB L2 cache. Fig 1

shows the cores connected with a 4x6 2D mesh. The SCC has 24 dual-core tiles connected to a router. Each tile contains two cores, a *Mesh Interface Unit* (MIU) and two *test-and-set* registers. The cores are connected via a mesh network with low latency and high bandwidth (256 gigabytes per second).

The SCC does not use any cache coherency between the cores, but rather offers a special hardware in terms of *Message Passing Buffer* (MPB) or *Local Memory Buffer* (LMB) for explicit message-passing between cores. The MPB (16KB) is small but fast memory buffer, shared by all the cores to enable fast message passing between cores. Each core has a 8KB partition of MPB space. The SCC architecture provides a new instruction called *CL1INVMB* and a new memory type called *MPBT*, to provide the coherence guarantee between caches and MPBs. The MPBT data is not cached in the L2 cache, but only in the L1 cache. Of course, when reading the MPBs, a core needs to clear the L1 cache. As the SCC cores only support a single outstanding write request, a *Write Combine Buffer (WCB)* has been added to combine adjacent writes up to a whole cache line which can then be written to the memory at once. It is used only for writes to memory typed MPBT.

When a core wants to update a data item in the MPB, it can invalidate the cached copy using the CL1INVMB instruction [9]. Since explicit management of MPBs for message passing is quite burdensome, Intel provides an MPI-like message passing interface, called *RCCE* [10]. It is a small library for message passing tuned to the needs of many-core chips such as SCC. The communication between cores occurs by transferring data from the private memory through the L1 cache of the sending core to the MPB and then to the L1 cache of the receiving core. The MPB allows L1 cache lines to move between cores without having to use the off-chip memory.

There are also four DDR3 *memory controllers* on the chip, which are connected to the 2D-mesh as well. Each controller supports up to 16GB DDR3 memory, resulting in a total system cacpacity of 64GB. Each core is able to access only 4GB of memory because it is based on the IA-32 architecture. Therefore there must be a way to tell which parts of memory at the controllers belongs to which core(s). To solve this problem, Intel has come up with *Lookup Tables* (LUT). Each core has a LUT, which is a set of configuration registers that map the cores' 32-bit physical addresses to the 64GB system memory.

In addition, an external programmable off-chip component *(FPGA)* is provided to add new hardware features to the prototype. The off-chip FPGA in SCC has additional registers which could used by cores to notify each other; *Atomic Increment Counters (AIC)* and *Global Interrupt Registers (GIR)* [11]. The SCC's cores are able to send an interrupt to another core by writing a special value to the configuration registers of that core by using *GIR*.

III. The OpenMP Fork-Join Model and the Target Barrier

The cores of our system SCC can execute only one process at a time, therefore in the rest of the paper, we consider core and thread as equivalent. OpenMP adopts the *fork-join* execution model. Here, the program executes serially within a single thread, referred to as the *Master thread*. The Master thread executes sequentially until it encounters a *#pragma omp parallel* directive. Here, a parallel region is created, and a number of threads (*slaves*) is involved, which execute the code inside the region (parallel construct). At the end of the parallel construct the slave threads synchronize on a barrier, then only the Master thread resumes execution.

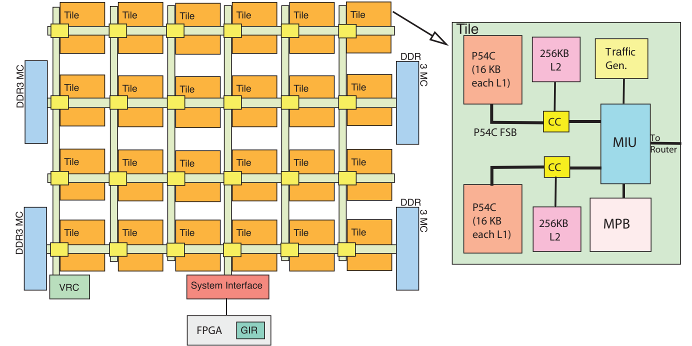

Fig. 1: Layout and tile architecture for the SCC

One common way to implement parallel regions is to create new threads on the fly relying on standard threading libraries such as *Pthreads*. However, *Pthreads* on SCC would require dedicated abstraction layers to allow threads on different cores to communicate. Therefore, our approach relies on a custom micro-kernel code [12] [14] executed by every core at startup. To minimize the cost associated to dynamic thread creation we assume a fixed allocation of the Master and slave threads to the processors. Master and slave threads execute different code based on their core ids. After system initialization, the Master core jumps to the execution of the parallel program, while the slaves wait on the barrier. When the Master encounters a parallel region, it invokes the runtime system, points the slaves to the parallel function, then releases them from the barrier. At the end of the parallel region, a global barrier synchronization step is performed. The Master continues executing sequential parts of the application, while the slaves come back on the barrier, thus implementing the join mechanism. Our implementation is based on the work from Marongiu et al. [7].

IV. Barrier Algorithms

There are several implementations of OpenMP for MPSoCs that adopt a centralized shared barrier [3] [4] [5]. This kind of barrier relies on shared entry and exit counters, which are atomically updated through lock-protected write operations. In a centralized barrier algorithm, each processor updates a counter to indicate that it has arrived at the barrier and then repeatedly polls a flag that is set when all threads have reached the barrier. Once all threads have arrived, each of them is allowed to continue past the barrier. A serious bottleneck arises with this algorithm because busy waiting to test the value of the flag occurs on a shared location [12]. Moreover, in non-cache coherent systems such as SCC, the updates to barrier

structures (e.g., control flags, counters) in shared memory must be explicitly kept consistent.

In the RCCE native programming model [10] there is a simple barrier algorithm based on a *local-put, remote-get* approach for message passing. Namely, a flag based synchronization only touches the MPB at that core, which has initiated an update. Consequently, the release cycle requires remote polling of the Master core, repeatedly for all following cores. At least, this approach avoids a centralized structure.

We exploit a *Master-Slave barrier* scheme and implement several algorithms on SCC as described below, as part of an effort to investigate ways in which OpenMP and its implementations may scale to large thread counts.

1) **Shared algorithm (SB)** This is the baseline implementation of the barrier algorithm, which allocates the flags in local shared memory (Master's MPB) and every core is responsible to initialize its own flag therein. We have used the MPB to allocate flags and the approach of RCCE programming model as a baseline, since this memory region is fast and accessible by all cores without any coherency issue and mainly used for message-passing between the SCC cores in an explicit way.
2) **Master-Slave algorithm (MSB)** The *Master-Slave* form of the barrier algorithm [12] [14]. In this approach the Master core is responsible for locking and releasing the slave processors. This is accomplished in two steps. The Master core is responsible for gathering slaves at the barrier point. This operation is executed without resource contention, since every slave signals its status on a separate flag. After this notification step slaves enter a waiting state, where they poll on a private location. In the release phase of the barrier (that has been implemented as a separate function to allow doing independent work before releasing the slaves) the Master broadcasts a release signal on each slave polling flag. The *Master-Slave* barrier algorithm removes the contention for shared counters. However, the traffic generated by polling activity is still injected through the interconnect towards shared memory locations, potentially leading to congestion. This situation may easily arise when a single processor performs useful work while the others wait on a barrier. Marongiu [14] exploited a distributed implementation of the barrier algorithm to address this issue by allocating each of the slave poll flags onto their local memory and using a message passing-like approach for signaling.
3) **Shared-Master-Slave algorithm (S-MSB)** The same as MSB, but instead of allocating each of the slave's poll flag onto their local memory this scheme uses *local-get, remote-put* approach. During the gather phase the Master core polls on memory locations through which slaves indicate their arrival from their own MPB.
4) **Shared-Master-Slave-Interrupt algorithm (S-MSBI)** The SCC has an FPGA as shown in Fig 1, which is directly connected to the on-die mesh interconnect and allows for adding new hardware features. Reble [15] used exponential backoff and AIC for Lubachevsky barrier implementation. This implementation significantly reduces the contention and leads to promising results. Also, he implemented another approach based on AIC and MPB, only by using a single AIC to indicate incoming threads and MPB located flags to release waiting threads. An S-MSBI uses a GIR to release the slaves, and the same approach of gather phase in S-MSB. Petrovic [16] presented the broadcast algorithm based on GIR to address the problem of the delay using MPB polling for notification. We exploited this technique to reduce the time consumption in the release phase of MSB by using the user-space library for interrupt handling. The slave core waits until it receives the interrupt signal from the Master.
5) **MSB-Interrupt algorithm (MSBI)** It is a same approach of MSB, but using GIR to terminate the slaves.

Listing 1 and Listing 2 show the software implementation of barrier algorithms based on the *Master-Slave* approach (similar implementations have been done for the other barriers).

Listing 1: The barrier code executed by the Master core to gother and release Slaves.

```
void Wait()              // Wait()
{
    int counter;
    while (counter != (nCores - 1))
    {
      do{
            CL1INVMB();
        }while(*MASTER_FLAG(counter) != 0);

     CL1INVMB();
     *MASTER_FLAG(counter) = 1;
     FLUSH_MPB();

     counter++;
}

void Release()        // Release()
{
    unsigned char old_val;
    int i;

    CL1INVMB();
    old_val = *SLAVE_FLAG(myCoreID);

    CL1INVMB();
    for ( i = 1; i < num_threads; i++)
         *SLAVE_FLAG(i) = old_val + 1;

    *SLAVE_FLAG(myCoreID) = old_val + 1;
    FLUSH_MPB();
}
```

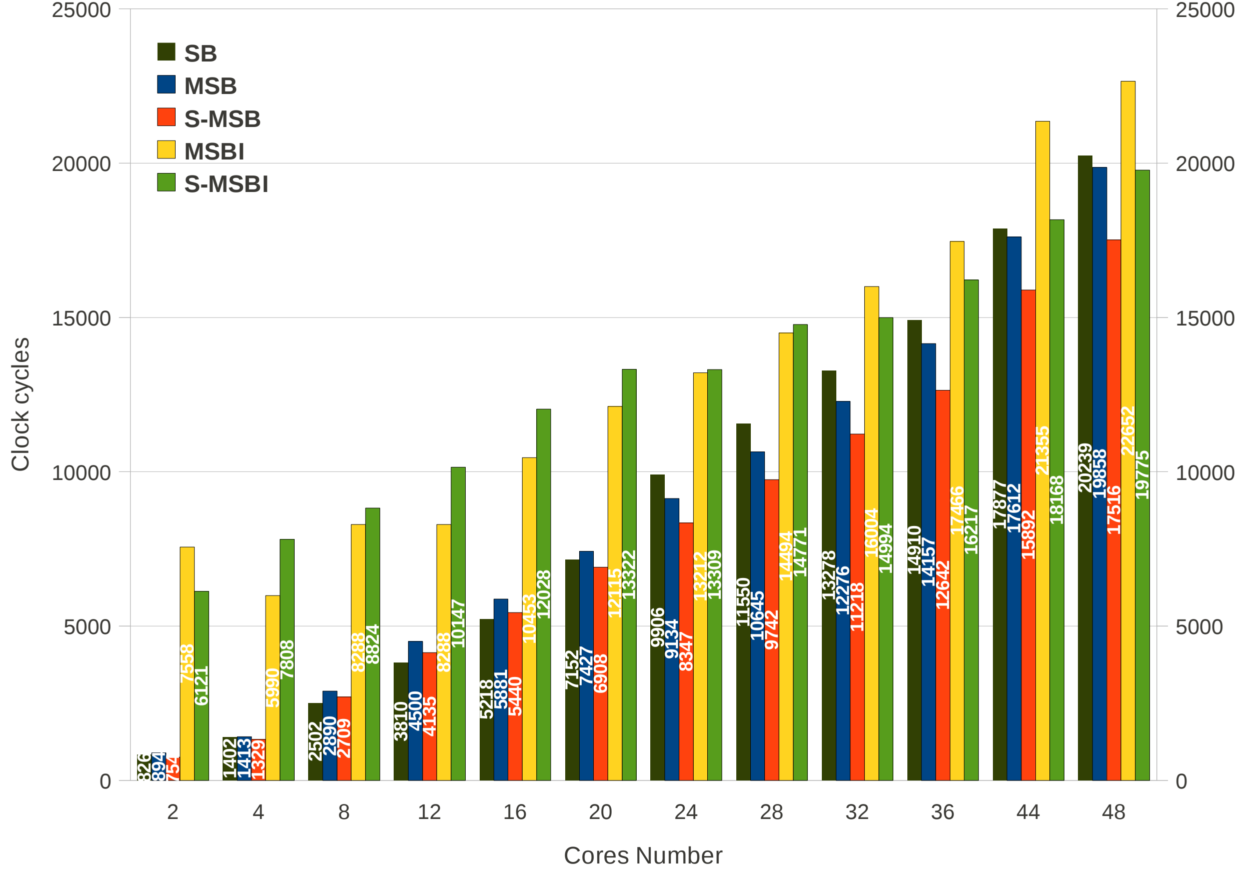

Fig. 2: Cost of barrier algorithms with increasing number of cores

Listing 2: The barrier code executed by the Slave Core to notify the Master core

```
void Slave_Enter()        //Slave_Enter()
{
  CL1INVMB();
   volatile unsigned char old_val;
   old_val = *SLAVE_FLAG(myCoreID);

     /* Upadted Master_flag pointer
          and fool WCB to write to MPB. */
  CL1INVMB();
  *MASTER_FLAG(myCoreID) = 0;
  FLUSH_MPB();

  do{
         CL1INVMB();
  }while(*SLAVE_FLAG(myCoreID) == old_val);
}
```

V. Experimental Results

In this section we present the experimental setup and the results achieved. All the experiments have been conducted under the default SCC settings: 533 MHz tile frequency, 800 MHz mesh and DRAM frequency and standard LUT entries. We use sccKit 1.4.2.2, running a custom version of sccLinux, based on Linux 2.6.32.24-generic. In order to perform timing analysis, Intel *RDTSC* (Read Time Stamp Counter) instructions [17] are inserted before and after the barrier algorithm for 100,000 iterations and then the difference between their values is computed. The time measurement is only performed and printed on the Master core.

A. Barrier synchronization

In this section we discuss the cost for different approaches to perform barrier synchronization. The experiments have been carried out by executing only barrier code on the platform (see Listing 1 and 2). No other form of communication between cores takes place, thus allowing to estimate how the algorithm scales with increasing traffic for synchronization only. The scalability of the different barriers described in Section IV is analyzed by varying the NoC topology size. We refer to the topology size only by the number of cores, however the topology takes also into account the external off-chip device (off-chip memory and FPGA register) .

The direct comparison of these barriers is shown in 2. The SB algorithm provides the worst results as compared to MSB and S-MSB, as expected. The cost to perform synchronization

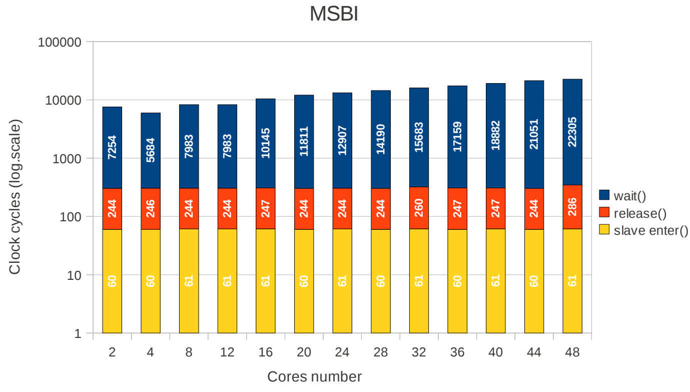

Fig. 3: MSBI algorithm

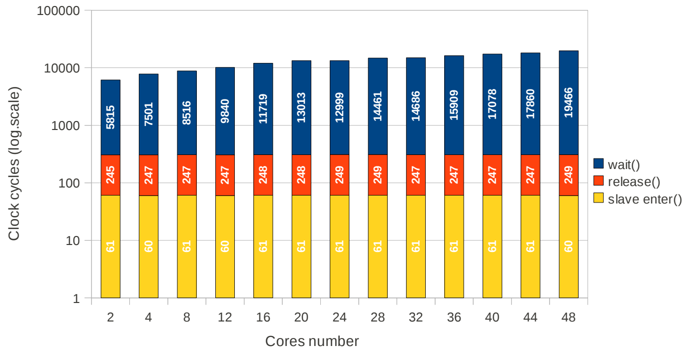

Fig. 4: S-MSBI algorithm

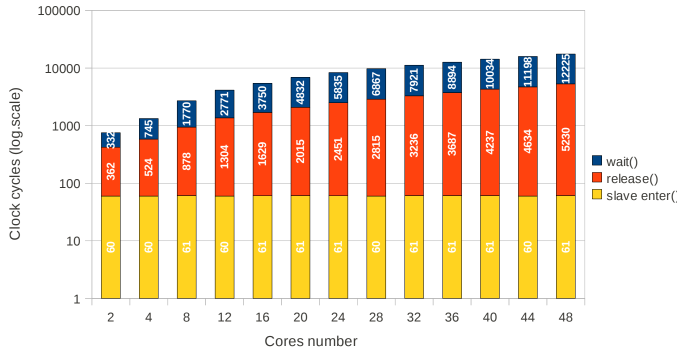

Fig. 5: S-MSB Algorithm

across 48 cores with this algorithm is 20,239 cycles. The MSB scheme, slightly mitigates the effects of the bottleneck due to contended resources, allowing to synchronize 48 cores in 19,800 cycles. Globally, S-MSB results always the fastest barrier, making it the ideal candidate to perform barrier synchronization regardless of system size and NoC topology when no hardware support is provided. This algorithm reduces the cost for synchronizing 48 cores to 17,500 cycles.

The cost for each of the phases (gather + release) of the algorithms MSBI and S-MSBI are plotted in Figures 3 and 4 respectively. Comparing Figure 5 with Figures 3 and 4 we can see that, although the S-MSB release phase is the slowest implementation, the corresponding MSBI and S-MSBI implementations are among the fastest. The results indicate that latencies observed by different cores in the release phase are practically indistinguishable (about 250 cycles). This implies that the cost for notification using parallel interrupt is practically constant with respect to the number of cores notified. The reason for bad scaling of the interrupt mechanism is contention, as confirmed by Petrovic [16]. There is a number of steps that a core should perform when receiving an interrupt. This includes reading from the status register, determining the sender and resetting the interrupt by writing to the reset register. Since all the registers related to interrupt handling are on the FPGA, access to them is handled sequentially.

Therefore, when an interrupt is sent to many cores at once, they all try to access their interrupt status register at the same time, but their requests conflict and are handled one after another, which explains the observed performance loss. Consequently, this problem increases the overhead of the barrier.

B. Barrier Algorithm Optimization

The SCC hardware recognizes two types of accesses to its *message passing buffer* (MPB): MPBT or non-MPBT. There is also an (*un-cached (UC)*) mode; the data read from such memory is not cached and write operations are directly issued to the network. Concurrent writes to the same memory line do not conflict. We could also mix two modes to access the same physical address as illustrated in [9]. The barrier algorithms are implemented by using shared bytes and the type of data used is MPBT. A common implementation for a read-updates-write operation on these bytes as below:

```
1.CL1INVMB();
2.<read byte(s)>
3.CL1INVMB();
4.<write modified byte(s)>
5.FLUSH_MPB();
```

The barrier algorithms exploit UC mode, and we could avoid extra overhead for invalidating MPBT lines before read and write operations, as well as the cycles required to flush the *write-combine buffer* (WCB). Figure 6 shows that UC mode allows a 41.5% overhead reduction for the S-MSB algorithm for 48 threads. It is clear here that the choice of a good barrier implementation can be memory access mode dependent.

VI. Conclusion and Future Work

In this paper we discussed several barrier algorithms to support the OpenMP fork-join execution model on the *Single-Chip Cloud Computer*, considering standard implementations and optimizations specific to SCC (i.e., use of HW support for synchronization, or explicit allocation of barrier structures in the MPB for reduced contention). Our experimental results highlight that we can obtain a significant reduction in overhead for standard barrier algorithms when using Shared-distributed busy-wait approaches in UC mode. Regardless of the NoC topology of SCC and system size, the S-MSB-UC (the implementation based on UC mode access) is the best barrier

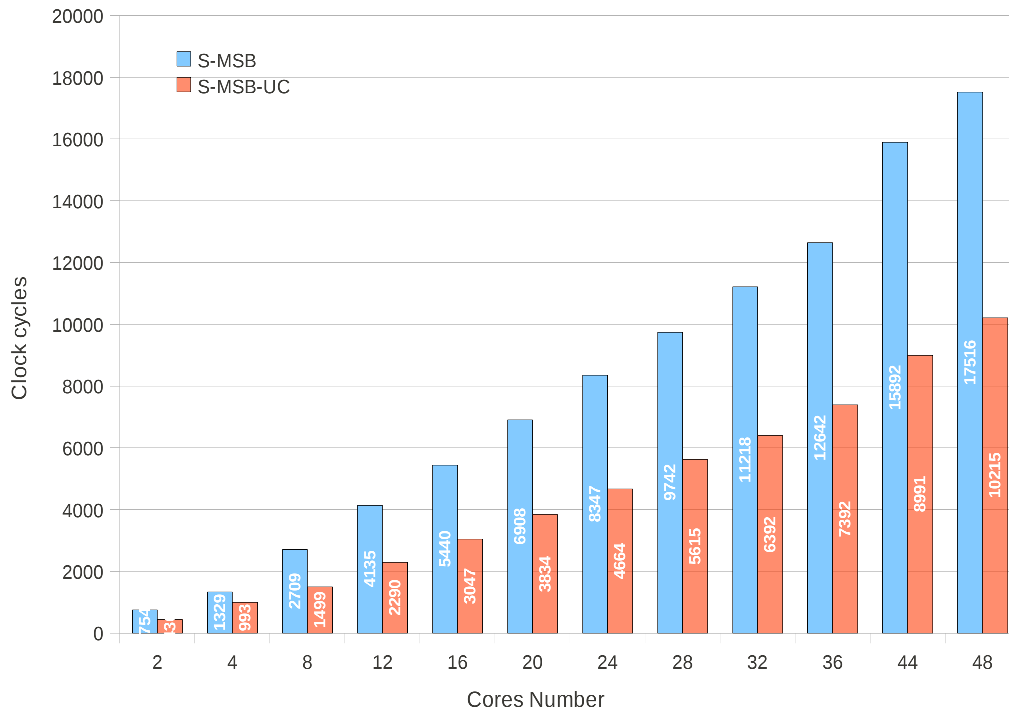

Fig. 6: Performance of S-MSB barrier algorithm with increasing number of cores

approach, which allows 41.5% faster synchronization than S-MSB and 48.5% faster synchronization than SB algorithm. The findings of this paper will constitute the basis for our future work, namely the implementation of a fully compliant OpenMP programming model for the SCC. We are currently dealing with the toughest challenge to support OpenMP data sharing on SCC: making shared data from main memory visible to all threads in presence of several OS instances, each with its virtual memory space. Besides these functionality issues, we are also dealing with the necessity of ensuring a consistent view of shared memory in absence of dedicated hardware cache coherence support.

References

[1] G. Blake, R. Dreslink, and T. Mudge. *A survey of multicore processors*. Signal Processing Magazine, IEEE, 26(6), November 2009.

[2] www.OpenMP.org. *OpenMP application program interface v.3.0*.

[3] W.-C. Jeun, and S. Ha. *Effective OpenMP implementation and translation for multiprocessor system-on-chip without using OS*. Design Automation Conference, 2007. ASP-DAC 07. Asia and South Pacific, pages 44–49, 23-26 Jan. 2007.

[4] F. Liu, and V. Chaudhary. *Extending openmp for heterogeneous chip multiprocessors*. Parallel Processing, 2003. Proceedings. 2003 International Conference on, pages 161–168, 6-9 Oct. 2003.

[5] F. Liu, and V. Chaudhary. *A practical openmp compiler for system on chips*. International Workshop on OpenMP Applications and Tools, WOMPAT 2003, pages 54–68, June 2003.

[6] K. O'Brien, Z. Sura, T. Chen, and T. Zhang. *Supporting OpenMP on cell*. In IWOMP 07, pages 65–76. Springer-Verlag, 2008.

[7] A. Marongiu, P. Burgio and L. Benini. *Fast and lightweight support for nested parallelism on cluster-based embedded many-cores*. Design, Automation & Test in Europe Conference & Exhibition, 2012. DATE '12.

[8] Intel Corporation. *SCC External Architecture Specification (EAS)*. July 2010. Revision 0.99.

[9] R. Rotta. *On Efficient Message Passing on the Intel SCC*. In proceedings of the 3rd Many-core Applications Research Commmunity (MARC) Symposium. KIT Scientific Reports, vol. 7598. KIT Scientific Publishing, 2011

[10] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. *Light- weight communications on intel's single-chip cloud computer processor*. In Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC10), New Orleans, LA, USA, November 2010.

[11] Intel Corporation. *The SccKit 1.4.0 User's Guide*. October 6, 2011. Revision 1.0.

[12] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. *Implementing OpenMP on a high performance embedded multicore MPSoC*. In IPDPS 09, pages 1–8. IEEE Computer Society, 2009.

[13] IV. Gramoli, R. Guerraoui, and V. Trigonakis. *TM2C: a Software Transactional Memory for Many-Cores*. In Proceedings of the 7th ACM european conference on Computer Systems, Pages 351-364, 2012.

[14] A, Marongiu, P, Burgio, and L, Benini. *Supporting OpenMP on a multi-cluster embedded MPSoC*. in Microprocessors and Microsystems - Embedded Hardware Design, 2011, pp.668-682.

[15] P. Reble, S. Lankes, F. Zeitz, T. Bemmerl. *Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor*. In Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12), Berkeley, CA, USA, June 2012.

[16] D. Petrovic , O. Shahmirzadi, T. Ropars, and A. Schiper *Asynchronous Broadcast on the Intel SCC using Interrupts*. In proceedings of 5th Many-core Applications Research Community (MARC) Symposium, Toulouse, France, July 19-20, 2012.

[17] Intel Corporation. *Intel Application Notes - Using the RDTSC Instruction for Performance Monitoring*. Technical report, Intel, 1997.