Towards Converting POSIX Threads Programs for Intel SCC

Patrick Cichowski, Gabriele Iannetti, and Joerg Keller

***Abstract*—The Intel SCC is a multiprocessor-on-chip with 48 cores interconnected by an on-chip communication network, and an off-chip shared memory. As each core is running its own linux operating system instance, executing a POSIX threads program using multiple SCC cores is not possible directly. Thus, in spite of SCC's shared memory, communication between cores normally happens in message passing style via the RCCE library. We investigate how to transform the source code of a POSIX threads program, so that, with the help of a small runtime library, it can be compiled and executed on multiple cores of the Intel SCC. We validate a proof-of-concept implementation with the help of a synthetic benchmark program, and report on the speedups obtained.**

***Index Terms*—Shared memory programming, POSIX threads, Intel SCC, multiprocessor-on-chip**

I. Introduction

PARALLEL programming can roughly be classified in shared memory programming and message passing programming, where the programming styles typically correspond to the structure of the underlying parallel machine. A prominent variant of shared memory programming is the use of POSIX threads [1] (or pthreads for short) in C programs, with lots of application codes available. The Intel SCC [2] is an experimental 48-core multiprocessor-on-chip created by Intel. It comprises 48 cores (IA32 architecture) forming 24 tiles, each containing two cores with separate caches and message passing buffers, interconnected by a 6×4-mesh network-on-chip. The network also connects to four on-chip memory controllers that provide access to off-chip main memory. Each core normally runs its own instance of a linux operating system. Thus, although a shared memory physically is provided and can be accessed via the RCCE communication library [3], the direct use of Pthreads is restricted to one core at a time. Communication normally happens in message-passing style via the RCCE or the iRCCE libraries [4]. We refrain from a more detailed description and refer to the literature on SCC hardware and programming [2], [3].

Our goal is to provide a possibility to compile and run C programs with Pthreads on multiple cores of the Intel SCC. We assume that the source code of the program is available, and that we do not modify the operating systems. Thus, we target a user-space solution. Our approach is to provide a small runtime library for functionality not present so far, and a source-to-source transformation. The transformed program then is compiled and linked with the runtime library, and executed on the Intel SCC. While the transformation so far is done manually (thus only the transformation rules are available), we provide a prototype implementation of the runtime system for basic functionality such as thread creation. Thus, this is a work in progress. We evaluate the prototype with the help of a synthetic benchmark program. The experiments indicate that speedups are achieved over running the pthread program on one core.

The remainder of this article is structured as follows. In Section II, we analyze the problem how to bring pthreads to Intel SCC and derive possible solutions. In Section III, we sketch the prototype implementation of a runtime library to run transformed pthread programs on Intel SCC. In Section IV, we present our preliminary experimental evaluation of the prototype. In Section V, we conclude and mention future work.

All authors are with the Faculty of Mathematics and Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany. e-mail: (firstname.lastname@fernuni-hagen.de).

Manuscript received Oct 14, 2012; revised XX XX, 2012.

II. Pthreads and Intel SCC

In a program written in the language C that uses the pthread API, initially one thread exists, that executes the main-function (including function calls). At any point in the program, another thread can be spawned by calling `pthread_create`. This call specifies the function that the new thread is to execute. Upon completion of that function, the thread terminates. The threads run either asynchronously (detached), or the spawning thread eventually calls `pthread_join` and blocks until the called thread terminates. All global variables are shared among the threads, i.e. only allocated once in the shared memory and visible to all threads. The same holds for dynamically allocated memory, if a pointer to it is held in a global variable. Local variables, or in general the function stack, are allocated separately for each thread, and are visible only to the thread that created them. Access to shared memory by several threads must be coordinated explicitly, for example by a mutex variable. For further details, we refer to a textbook on pthread programming, e.g. [5].

In this section, we will mainly deal with thread creation and shared memory access in the SCC, and only sketch solutions for coordination functions like mutexes.

A. Thread creation

As a first basic decision, we decided to use the existing pthread API available on each core, to ease implementation. Because any thread can spawn another thread, the question arises whether the thread should be created on the core where it was spawned, and later migrated to another core for the purpose of load balancing, or whether the thread should be created where it should run later, thus avoiding thread migration. As we did not want to interfere with the operating system, we decided for the latter possibility. This necessitates that we need a thread description (which function is to be executed with which argument) that can be sent to the target core, and be used to create the thread there. A similar thing happens upon termination of a thread. The return value must be sent back to the thread that spawned it, to be used in a join-function.

In order to free the cores as far as possible from these administrative tasks, such as finding an appropriate core to run a new thread on, we decide for a central instance, to which all descriptions of newly spawned threads and return values of terminated threads are sent, and which distributes these descriptions to target cores where the respective threads shall be created and run, and which forwards the return values to the spawning thread. Thus, the proposed solution fits the master/worker pattern, see e.g. [6].

In the pthread program, every call to `pthread_create` and `pthread_join` is replaced by a call to the respective library functions. The `main`-function is renamed, and a new main-function is added that starts the cores as master or worker, and lets the master start the first thread, that executes the original main-function.

B. Shared memory access

On the SCC, shared memory must be allocated explicitly before it can be accessed. The allocation is done with a shmalloc-call by all participating cores. We use the shared memory without caching because the SCC hardware does not support any cache coherence protocol. Implementing cache coherency in software is a problem independent of pthread programming, and thus not in the focus of our research. It has been addressed in a number of approaches since Li and Hudak's seminal paper [7], and has been investigated also for the Intel SCC [8].

All global variables must be placed in this shared memory. This can be realized by accessing each variable via its offset in the shared memory. Additionally, dynamic allocations of shared memory, i.e. a call to `malloc` where the returned pointer is at some time assigned to a shared variable, must be realized with the above allocated shared memory. As we already have proposed a centralized solution for thread creation, and as malloc-calls typically are as infrequent as thread creation, we use the master also for managing the dynamically allocated memory. Thus, a thread executing a malloc sends a message with the required size to the master, which performs the allocation and returns in another message the offset of the allocated memory within the shared memory.

In the pthread program, the transformations for variable access can be extensive if the address operator is used. Otherwise, each occurence of a shared variable is replaced by an expression of the form pointer to shared memory plus offset of variable. The analysis to find the malloc's of shared memory can also be difficult, as the returned pointer might first be assigned to a private thread-variable, and only assigned to a shared variable after several intermediate steps. If the analysis cannot be performed satisfactorily, a work around is to declare all malloc-calls as malloc of shared memory.

C. Coordination functions

For coordination functions like a mutex variable, which are typically called much more frequently than thread creation or memory allocation, a centralized solution does not seem preferable. Therefore, we envision hybrid solutions, where first the threads in each core coordinate, e.g. try to get a local mutex, and then the local "winners" coordinate themselves with the help of a distributed algorithm for leader election (see e.g. [9]), and create a global winner. An example of an efficient implementation of a global minimum computation with the help of the message-passing buffers can be found in [10].

The SCC also provides hardware support for synchronization primitives such as special registers that can be test and set atomically [8], so that such an approach seems feasible.

We are aware that some more complex coordination functions might need more elaborate approaches.

III. Prototype Implementation

The implementation of the runtime system uses one core to run the master and a configurable number of cores (up to 47) to run the workers, where one of the worker cores is reserved for the main function of the original program, i.e. the first thread. This is done to ensure enough computational power for the main thread, as the original pthread program often exhibits a master-worker pattern. Initially, all cores execute the shmalloc function of the RCCE library, to allocate a part of SCC's shared memory. All shared variables, i.e. the variables that are globally defined, are placed there, and this shared memory also serves calls to malloc where the address of the allocated memory at some point will be stored in a shared variable, to form some dynamically allocated shared memory. Those malloc-calls are transformed into messages to the master, see below.

The master receives information about each created thread in a message. The information consists of an identifier for the function that the new thread shall execute, and the argument pointer to that function. To achieve this, the master regularly polls for messages that have newly arrived, with the help of a function from the iRCCE library.

The master chooses the worker core that currently runs the smallest number of threads and sends this information in a message to that worker. Such a strategy does not incur additional overhead as the master is informed about all thread creations and terminations.

While we avoid dynamic load-balancing with thread migration, we still balance the number of threads per core. As the runtime of threads is rather long, i.e. creation and termination are infrequent events, this also balances the core workload as long as the threads are not too heterogeneous in their performance characteristics.

Each worker runs a coordination thread that waits for messages from the master and manages the internal state of this worker process. If the coordination thread receives a message to start a thread, it spawns a thread locally as detached if necessary (see below). Each started thread receives a pointer to a structure that contains the function to be executed, the argument pointer to that function and a place to store the return value of that function. The thread calls the function with the argument pointer, and upon return from that function, stores the return value in the structure, and sends a completion message to the master, after it has finished the execution of a function. For the purpose of reuse and a reduced overhead which is associated with each thread creation and destruction, each worker holds at least one thread to execute functions from incoming create-messages. If a so called basic worker thread is busy, while executing a function already, the coordinating worker thread spawns a new thread to create a function. If the base worker thread returns from the execution it blocks until it is assigned a new function by the coordinating thread of the worker.

The existence of a coordination thread on each core ensures that only one thread from each core retrieves messages from the master. As the RCCE and iRCCE libraries are not thread-safe, sending of messages is coordinated by a mutex variable.

If a thread of the transformed program reaches a `pthread_join`, then it sends a message to the master and waits for an answer. The master, when it has received such a message, will send back the terminated thread's return value as soon as it has received it, i.e. as soon as the thread is terminated. This message is received by the worker core's coordination thread, that forwards the return value to the waiting thread with the help of a local mutex variable.

If a thread of the transformed program reaches a `malloc`, it sends a message with the required size to the master, that allocates the memory and returns the offset in another message. The forwarding of the offset within the worker core occurs similarly to the return value in case of a join.

There have not been implementations for coordination functions so far.

IV. Experiments

To validate the prototype implementation of the runtime library, we use a synthetic benchmark program that starts with a main thread. There is only one function: If the maximum recursion depth r is not yet reached, a thread executing this function performs the following two steps k times: spawn another thread and perform a complex summation loop afterwards. Then the thread terminates and returns the result of the summation. If the maximum recursion depth is reached, the thread terminates immediately without any calculation. The number of threads thus is regulated by the two parameters r and k, and the total number of threads is

$$\sum_{i=0}^{r} k^i = \frac{k^{r+1}-1}{k-1} \; .$$

The rationale behind the benchmark's structure is to create a larger number of threads, where the number is growing over time and where shared memory access does not play a prominent role, so that the influence of the slow, uncached shared memory on performance is low. For $k = 3$ and $r = 5$, a total of 364 threads (including the main thread) are created.

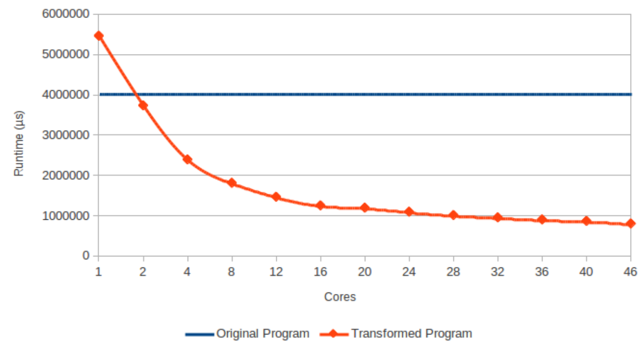

Fig. 1. Benchmark runtime depending on number of workers

We execute the manually transformed benchmark program on the SCC using different numbers of workers. The maximum number of useable worker cores is 46, because one worker exclusively runs the main thread, and one core is reserved for the master process. We repeat each experiment multiple times and compute the average runtime. For comparison, we also execute the original pthread-program on one core of the SCC and measure its runtime. For all experiments, we use a core frequency of 533 MHz and a network frequency of 800 MHz.

The runtime results are depicted in Fig. 1. Note that the worker core that exclusively runs the main thread of the transformed program is omitted. We see that already for 2 workers (plus 1 master and the core for the main thread, i.e. 4 in total), the transformed program achieves a speedup over the runtime of the original pthread program on a single core. We also note, that beyond 16 workers, i.e. 18 participating cores, the runtime improvements become very small.

To analyze the runtime data, we model our benchmark experiment in the following way. We consider the computation to proceed in discrete steps, where each step takes as long as the complex summation (i.e. several hundred thousands of instructions). Each thread on a recursion level of less than r lives for k steps, threads on the last recursion level live for one step. At the beginning of each step, each active thread on a recursion level $r_c < r$ also creates another thread at recursion level $r_c + 1$. A created thread gets active after a certain delay d. This delay is the only overhead for thread management that we consider. Active threads are assumed to be load-balanced. Figure 2 depicts the number s_i of active threads in each step i for two rather extreme values of d. On the one hand, $d = 0$ means that threads can start without delay, i.e. models an ideal case. On the other hand $d = 1$ means that the delay from thread creation to thread start equals the duration of one step, which is rather long.

We see that in both cases, even with immediate thread start, only a fraction of the steps has a thread count that is higher than the number of cores. The maximum possible speedup for n steps is

$$S = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \min(s_i, 46)$$

as a maximum of 46 workers limits the speedup per step. For $d = 0$, $S \approx 31$, and for $d = 1$, $S \approx 23$. For a really

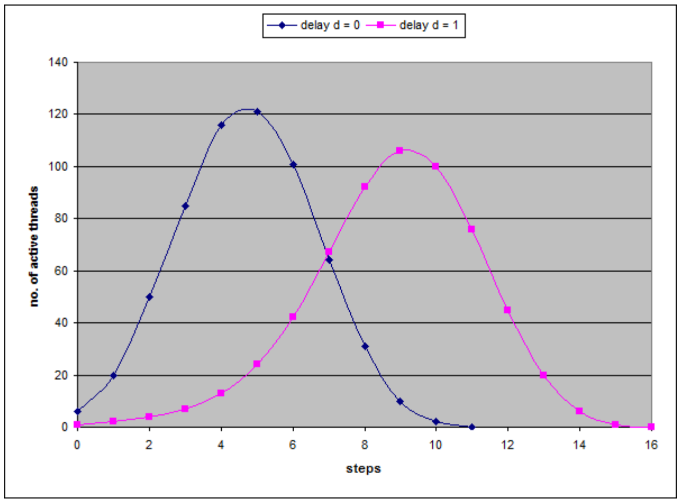

Fig. 2. Number of active threads at different timesteps

huge delay $d = 2$, the maximum possible speedup would go down to 22. We see that the structure of the benchmark, which reflects Pthreads' limitation of being able to create only one thread at a time, leads to a rather restricted speedup. The speedup in the real experiment is lower as it reflects additional overhead in thread management not present in the model. The restricted parallelism available also explains why almost no further speedup can be gained in the experiment when more then 16 workers are used.

V. Conclusions

We have presented the concept and a prototype implementation of a runtime library to transform and execute C programs with POSIX threads on the Intel SCC. Our preliminary experimental validation indicates that speedups over a single core execution can be achieved. Future work will comprise the implementation of more functions of the POSIX threads API, e.g. mutex locks, the implementation of an automated source-to-source transformation, and to improve the efficiency of the basic mechanisms of our implementations. For example, a better compromise between simple thread distribution and load balancing could be found by placing the workers' task queues in the shared memory and employing a work-stealing algorithm.

Acknowledgment

The authors would like to thank Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer".

References

[1] Institute of Electrical and Electronics Engineers, "POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)," 1995.

[2] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart, "A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling," *IEEE J. of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[3] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, P. Haas, W. andKennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC processor: The programmers view," in *Proc. 2010 ACM/IEEE Conf. on Supercomputing (SC10)*, 2010.

[4] C. Clauss, S. Lankes, J. Galowicz, S. Pickartz, and T. Bemmerl, "iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – User Manual," Chair for Operating Systems, RWTH Aachen University, Tech. Rep., November 2011.

[5] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997.

[6] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Pattern for Parallel Programming*. Addison-Wesley, 2010.

[7] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Systems*, vol. 7, no. 4, pp. 321–359, 1989.

[8] P. Reble, S. Lankes, F. Zeitz, and 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. [Online]. Available: https://www.usenix.org/system/files/conference/hotpar12/hotpar12-final9.pdf

[9] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, 2000.

[10] C. Clauss, S. Lankes, and T. Bemmerl, "Mapping the PRAM model onto the Intel SCC many-core processor," in *Proc. High Performance Computing and Simulations Conf.*, 2012, pp. 395–402.