UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores

Bakker, R.; van Tol, M.W.

Publication date
2012
Document Version
Author accepted manuscript
Published in
Universitaet Potsdam. Hasso-Plattner-Institut fuer Softwaresystemtechnik. Technische Berichte

Link to publication

Citation for published version (APA):
Bakker, R., & van Tol, M. W. (2012). Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores. *Universitaet Potsdam. Hasso-Plattner-Institut fuer Softwaresystemtechnik. Technische Berichte*, *55*, 55-60.

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:09 Mar 2023

Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores

Roy Bakker and Michiel W. van Tol
Informatics Institute, University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—The Single-chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs targeting the many-core research community. It has hardware support for sending short messages between cores, while large messages have to go through off-chip shared memory. In this paper we discuss our implementation of the SVP model of concurrency on this architecture, and how we deal with its distributed memory design and communication bottlenecks. We employ our previously developed *copy core* technique and show which approaches show scalable performance against our original implementation.**

I. Introduction

The Single-chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides an on-chip message passing network, a non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling. In this paper we discuss our implementation of SVP on this platform, a hierarchical concurrent execution model [2]. In future work, we will use this implementation and exploit its dataflow-style execution to provide us with a handle for adaptive power management.

The Self-adaptive Virtual Processor, or SVP, is an abstract concurrent programming and machine model, which evolved from the earlier work on the Microthread CMP architecture [3] which implemented SVP in hardware [4]. The model can be used to express concurrency at many levels of granularity for multi- or manycore systems, and uses shared memory semantics with a weak consistency model. As the SCC has a distributed shared memory architecture without cache coherency, this suits the consistency model of SVP very well. As SVP actions can be trivially translated into messages in a distributed environment, this maps well onto the message passing communication infrastructure of the SCC.

Effectively, the SCC is an on-chip distributed system, and therefore we can already run the available distributed implementation of SVP [5] without any modifications. As this is based on the coarse grained communication primitives of TCP/IP sockets, we experimented with different approaches to more efficiently use the hardware messaging support on the SCC. However, we have already shown in previous work [6] that using the on-chip message passing buffers with RCCE [7] or iRCCE [8] are not sufficient for such an implementation. In this paper we will employ several of the techniques that we investigated in our earlier work to efficiently copy memory on the SCC, for example by using dedicated copy cores.

In this paper we will discuss our experiences with porting the distributed SVP runtime to the SCC. We assume sufficient knowledge of the SCC architecture and its memory system as this is broadly covered by both related work [1], [9] as well as our previous [6] work. First we will discuss the SVP model of concurrency and its consistency model in Section II, and discuss which approaches we considered for the implementation on the SCC in Section III. In Section IV we evaluate these different approaches and we conclude with a discussion and future work in Section V.

II. SVP

SVP is a generic concurrent programming and machine model which has a separation of concerns between the expression and management of concurrency. The SVP model defines a set of actions to express concurrency on groups (*families*) of indexed asynchronous activities (*threads*).

Each thread can execute a *create* action to start a new concurrent child family of threads, making the model hierarchical, and later on use the *sync* action to wait for its termination. The *create* action has a set of parameters to control the number and sequence of created threads, as well as a reference to the thread function that the threads will execute. This thread function can have a set of communication channels defined that are explained later on.

Besides these two basic constructs, there is the *kill* action to asynchronously terminate an execution. Programs for SVP based architectures or run-times are written in a dialect of the C language which has extensions to explicitly support these SVP actions and thread definitions.

A. Resources

While SVP code has no notion of what a resource physically is or how code is scheduled, an abstract resource identifier, a *place*, is provided. On a *create* action a *place* can be specified where the new family should be created, binding the execution onto a certain resource, similar to sending an Active Message [10]. What this *place* physically maps to, is left up to the SVP implementation; for example, on our implementation on the SCC it will be a single core, but on the Microgrid CMP [11], it is a group of cores. On other implementations it could, for example, be a reserved piece of FPGA fabric, an ASIC, or some time-sliced execution slot on a single- or multi- processor system.

As long as the underlying implementation supports it, multiple *places* can be virtualized onto a single physical resource. Mutual exclusion is supported through places; families delegated to an *exclusive place* are guaranteed to be sequentialized so that only one family can be executing on such a place at a time.

B. Communication and Synchronization

Each family has a set of synchronized communication channels that link up the threads and the parent context. There are two types of unidirectional write-once channels; *global* and *shared* of which multiple can be present. These channels have non-blocking writes and blocking reads. A *global* channel allows vertical communication from the parent thread to all threads in the family. A *shared* channel allows horizontal communication, as it daisy-chains through the sequence of threads in the family, connecting the parent to the first thread and the last thread back to the parent. These channels are defined as arguments of a thread function and identify the data dependencies between the threads.

Due to this restricted definition, and under restricted use of exclusive places, we can guarantee that the model is composable and free of communication deadlock [12], and that there is always a well defined sequential schedule if parallel execution is infeasible.

C. Memory Consistency

The model assumes a shared memory with a restricted consistency model. It is seen as asynchronous and therefore it is not suitable for synchronizations, and no explicit memory barriers or atomic operations are provided. The consistency model is described by the following three rules:

- A child family is guaranteed to see the same memory state as the parent thread saw at the point of *create*.
- The parent thread is only guaranteed to see the memory changed by a child after *sync* on the child has completed.
- A family on an *exclusive place* is guaranteed to see the changes to memory by earlier families on that place.

The memory consistency relationship between parent and child threads is similar to the well-known *release consistency* model [13]. The *create* resembles an *acquire*, and *sync* resembles the *release*. We should note that the third rule is a very important property as it can be used to implement communication between two arbitrary threads, but it can also be used to implement a service; state is resident at the *exclusive place* and instances of the functions implementing that service are created on the *place* by its clients.

D. Distributed SVP

Distributed SVP, or DSVP, is an extension to SVP to handle distributed memory [5]. The implementation of DSVP was our starting point for an SVP implementation on the SCC. The DSVP extension introduced the idea of a *data description function* which tells the implementation which parts of memory need to be sent/received when a thread function is started remotely with a create or completes with a sync, similar to how

```
thread fibonacci(shared int p1, shared int p2, int* result)
{
    index i;
    result[i] = p1 + p2;
    p2 = p1;
    p1 = result[i];
}
DISTRIBUTABLE_THREAD(fibonacci)(int p1, int p2, int* result, int N)
{
    INPUT(p1);
    INPUT(p2);
    ARRAY_SIZE(result, N);
    for(int i = 2; i < N; i++)
        OUTPUT(result[i]);
}
main()
{
    family fid;
    int    result[N];
    int a = result[1] = 1;
    int b = result[0] = 0;

    create(fid;;2;N;;) fibonacci(a, b, result);
    sync(fid);
}
```

Figure 1: Fibonacci code example

in- and outputs are annotated in CellSs [14] and Sequoia [15]. This is based on the premise that a thread needs to receive a reference to any data it will access through its communication channels, and therefore this identifies which data needs to be communicated to adhere to the consistency model.

An example code is given in Figure 1, showing a program that stores the Fibonacci sequence up to N into a result array. Threads 2 to N are created for the corresponding iterations and they communicate their dependent values through their shared channels p1 and p2. The data description function takes the two initial values for p1 and p2 as input, and returns the resulting fibonacci array as output. Please note that some parameters for create are omitted, for example one to set the *place* where the computation is executed and others to control more complexing indexing.

III. Implementation

The distributed SVP implementation that uses TCP/IP for communication between places [5] runs on the SCC without any modifications. However, it supports heterogeneous platforms with different data representations, which adds additional overhead. All data that needs to be communicated (indicated by a *data description function*) is serialized to a platform independent representation using XDR [16] before it is sent to the other *place*, where it needs to be deserialized again. On the SCC we can avoid this step, as it is a homogeneous system with the potential to use shared memory.

Our initial optimization was to skip the XDR step and send each data element that is part of the data description function directly through the socket. For scalar values this causes a large communication overhead, since they are now all pushed separately through the channel. For (large) arrays this means a great reduction in the overhead of encoding and copying, especially on the SCC where memory operations are expensive. The data description function was altered to support sending arrays in a single shot. To send arrays, we no longer need to explicitly touch each single element of the array, but

just provide a pointer to the first element, and the number of elements in that array.

A. Using (i)RCCE

Our first approach was to modify the communication layer of the existing implementation to make use of the RCCE and iRCCE libraries. We could easily replace all send and receive calls with the appropriate iRCCE functions. For the connection establishment we used an iRCCE waitlist with a receive request for each possible sending core. However, the (i)RCCE implementation only matches requests on core identifier. As a result, we cannot have multiple outstanding receive requests for a single sending core, and the first message that fits the size will complete. Therefore, we cannot issue another receive request in the connection establishment function while there is already a connection active. The messages in the connection establishment function are rather small, and therefore match any other larger sized message. We see that messages sent through a previously established connection now initiate a new connection, and fail thereafter. iRCCE does not support virtual channels, and therefore was not suitable for our SVP implementation in its current form.

B. Memory Remapping

The SCC allows us to share memory from one core with an other by using the programmable *look-up tables* (LUT), which means that the communication of large data chunks through a channel can be avoided. However, the virtual memory system of Linux makes this difficult as the virtual addresses seen by a process are not the same as the physical addresses. A function in the special SCC Linux memory kernel driver provides a virtual to physical address translation. The sccLinux virtual memory system uses 4KB pages, and chunks of contiguous virtual memory that span more than one page, therefore do not necessarily map to contiguous core-physical memory. However, as sccLinux does not support the use of swap space, the virtual to physical address mapping of a page is stable. Once the core-physical memory address is known, the real-physical address can easily be obtained from the LUT.

To make effective use of the memory remapping approach, we need to manage our own virtual to core-physical memory mappings, avoiding the fragmentation induced by the sccLinux virtual memory system. We use an sccLinux image that has only 320MB of private memory configured, which leaves about the same amount of memory for the application to manage as there is 656MB of private memory reserved for each core. We *mmap()* this region so that we have a contiguous mapping of virtual to core-physical addresses, and within that use our own memory allocator with a simple first-fit algorithm.

When using memory remapping, we can either use cached memory or non-cached memory. Cached memory has the downside that the L2 cache is write back and therefore needs to be flushed, which is an expensive [6] operation, to make sure all data will be in physical memory on the sending side. We can avoid the L2 flush at the receiving side by mapping the memory with the MPBT tag on, so that the L2 cache is bypassed, but multiple accesses within the same cache line will hit in the L1 cache. Then, it is enough to issue the cheap CL1INVMB instruction that invalidates all data in the L1 cache with the MPBT tag. The alternative, the use of non-cached memory, was not considered; it is too expensive as every individual access needs to go to main memory.

In the memory remapping implementation, the sending core will send the core-physical address through the socket to the receiving core. The receiving core checks the LUT of the sending core to obtain the real-physical address. It will dynamically map this address range on free LUT pages and start a memory copy operation to the receive buffer. As the initial socket communication, lut writing procedure and cache flushing will cause overhead, this approach is only efficient when the message is large enough to compensate for this overhead. To allow faster writing, the target area is remapped with the MPBT flag on, which enables the WCB. As these addresses are independent from the L2 cache perspective, and MPBT bypasses the L2 cache, no additional flushes are required in this approach.

The remapping approach is similar to the *Privately Owned Public Shared Memory* (POPSHM) approach proposed by Intel. However, POPSHM requires a copy of the data into the shared memory region on the sending side, and a copy out of the shared memory region on the receiving side. In contrast, we map the memory locations directly on demand at the receiving side, therefore only requiring a single copy operation which uses specialized memory flags for the fastest possible reading and writing.

C. Copy Cores

Instead of performing the memory copy operation at the receiving core, we can also choose to use our earlier proposed *copy cores* [6] to copy memory regions. Copy cores are dedicated cores that run a memory copy service; when data needs to be copied between cores, multiple copy cores can be employed to copy the data, similar to DMA engines, which due to the limited memory throughput of a single core should be able to deliver a better performance. Copy cores use the same approach to copy memory as the remapping implementation, using specialized flags for reading and writing. In the current implementation, all copy tasks are issued round robin to a set of copy cores.

IV. Evaluation

A. Benchmarks

1) Ping Pong: The first benchmark that we use is an SVP based Ping-Pong application which creates a computation on the remote node which terminates immediately, but using the data description function sends chunks of data back and forth with incremental size. We use this benchmark to measure the latency and throughput achieved by our different approaches. The sizes we measured range from 4 bytes up to 16MB, and are transferred between cores 0 and 1.

2) Matrix Multiplication: A benchmark that fits the distributed implementation of SVP with the potential to copy lots of data is matrix multiplication. We implemented a recursive decomposition algorithm that splits a matrix in sub matrices

and performs the calculations on sub matrices only. Figure 2 shows the decomposition algorithm. We can apply the decomposition recursively as long as the square matrix size is still dividable by two. Each step splits the calculation in eight parts that can execute concurrently, followed by four additions that can also execute concurrently. Note that the addition can be performed on the individual sub matrices, or on the combined larger matrices. In this benchmark we perform the addition on the sub matrices, since this exposes more concurrency without changing the representation again. This implementation works on square matrices and operates on double precision floating point values.

$$A \times B \rightarrow \begin{vmatrix} a1 & a2 \\ a3 & a4 \end{vmatrix} \times \begin{vmatrix} b1 & b2 \\ b3 & b4 \end{vmatrix} =$$
$$\begin{vmatrix} a1 \times b1 & a1 \times b2 \\ a3 \times b1 & a3 \times b2 \end{vmatrix} + \begin{vmatrix} a2 \times b3 & a2 \times b4 \\ a4 \times b3 & a4 \times b4 \end{vmatrix} = \begin{vmatrix} c1 & c2 \\ c3 & c4 \end{vmatrix} \rightarrow C$$

Figure 2: Matrix decomposition: Matrices A and B (both $N \times N$ are split into four $\frac{N}{2} \times \frac{N}{2}$ matrices each. Eight matrix multiplications and four matrix additions are performed on the sub matrices.

In order to make the decomposition more time and space efficient, the matrix representation in memory is a column of pointers that all index a row in the matrix. All matrix rows together form one contiguous block of memory, both virtual and physical, guaranteed by our own memory allocator. This representation is visualized in Figure 3. The normal lines indicate a pointer to an element in memory, while the dashed lines refer to the same element in the corresponding matrix. Pa, Pb, Pc and Pd are pointers to arrays with pointers to sub matrix rows. This allows us to do the decomposition by creating a new array of pointers and assign the pointers to elements in the original matrix rows, without the need of copying data.

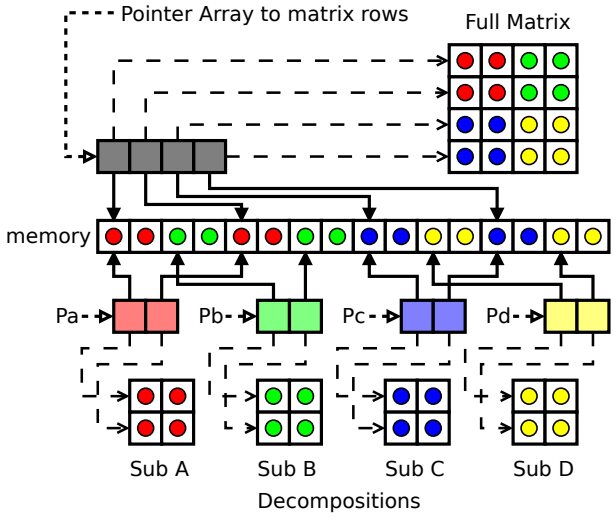


Figure 3: Representation of the original and decomposed matrices in memory

We have run two versions of the matrix multiplication benchmark with different distribution strategies. The first has only one master node that decomposes the matrices and sends the sub matrices to worker nodes. Initial experiments have shown that only a single master node can not keep the other cores busy when we use two decomposition steps creating $8 \times 8 = 64$ concurrent multiplications on only 47 (or 48, when the master is included) nodes. The overhead for the decomposition and communication is too large compared to the computation performed by the worker nodes. In the second version the master node does a single recursion step, and then delegates the work to eight nodes which in turn do the second recursion step to create a total of 64 tasks. Using this method, we divide the communication overhead over multiple nodes, but the total amount of required communication is higher.

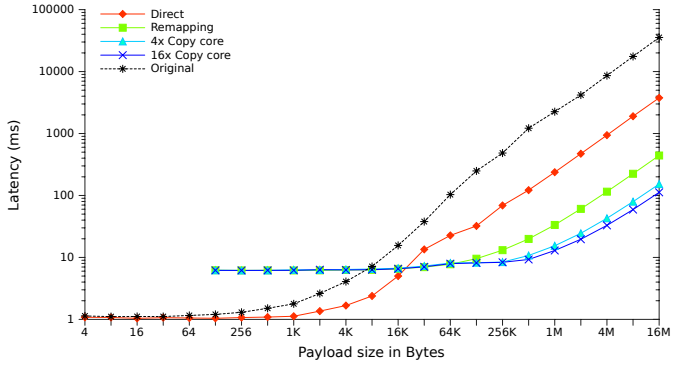
B. Results

1) Ping Pong: In Figure 4, the results of the Ping Pong benchmark are visualized in two graphs. The first graph shows the best achieved latency of creating a remote computation, followed by a synchronization directly thereafter with different data payloads. All results are the minimum over 10 measurements, to compensate for outliers generated by TCP/IP timeouts that would have a large impact on an average. We show the results for the previously discussed implementations; *Direct* is the same implementation as the original but without the XDR encoding and decoding steps, *Remap* is the approach that remaps and copies the memory on the receiving side, and *Copy core* is spreading the copy operation over 4 or 16 dedicated copy cores.

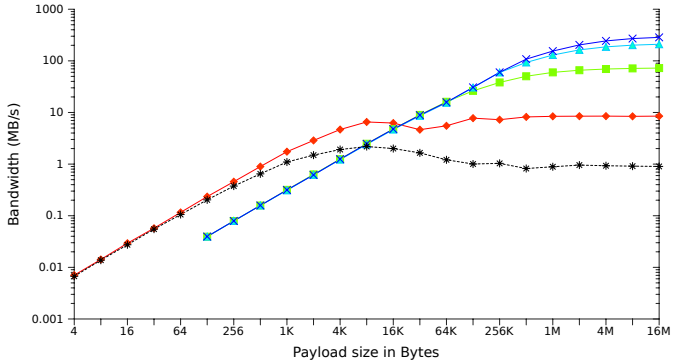
The new approaches have a higher latency, around 6 ms instead of 1 ms, for a small payload, due to the required L2 cache flush, but have a much better latency when communication a lot of data. This is further shown in the second graph, which shows the corresponding throughput, note that this graph is also on a log to log scale. As the initial communication of the addresses still goes through TCP/IP, we set a threshold for using remapping or copy cores to 128 bytes, however they only become faster then the direct approach when transferring more then 16 KB.

The direct communication approach reduces the execution time by almost an order of magnitude compared to the original implementation. The speedup peaks at a factor of 9 for messages larger than 512KB. Remapping memory is about two orders of magnitude faster than the original implementation. The copy core approach clearly improves on the memory remapping approach, as it can aggregate more bandwith by using multiple cores, as described in [6]. It is three times as fast as remapping when using 4 copy cores, and four times as fast with 16 copy cores, where you start to notice the delegation and synchronization overheads to send the Copy cores their work requests.

2) Matrix multiply using one decomposition step: In this benchmark we run our matrix multiplication application using one decomposition step, resulting in eight remote creates. We ran the benchmark for square matrix sizes of 128, 256, 512 and 1024 elements, resulting in sub matrices of half that square size. The amount of communication is order $O(n^2)$ while the computation is in the order $O(n^3)$, which results in better scalability for larger matrices due to a better computation to



(a) Latencies for the different implementations



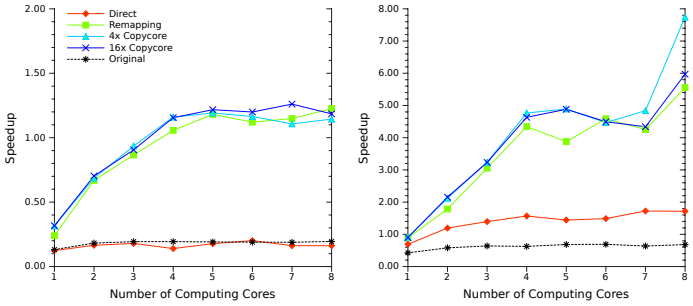
(b) Throughput for the different implementations

Figure 4: Results for the SVP based Ping Pong benchmark

communication ratio. The master node initializes the matrices, performs the decomposition, and distributes the work over 1 to 8 remote places. The results averaged over three runs and are shown in Figure 5. We benchmark again our four implementations; the original, the direct implementation, remapping and copy cores with 4 or 16 copy cores which are placed at the edges of the SCC chip around the memory controllers. All speedups are measured against a baseline of a non-threaded local matrix multiplication using the same computation kernel but without distribution or decomposition.

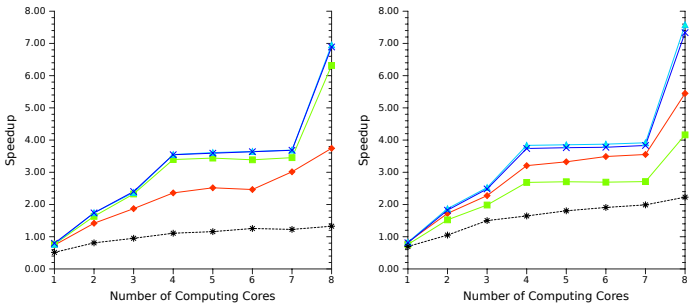
For a small matrix size of 128×128 elements (Figure 5a), we see the impact of the large overhead of communication. The original and direct implementation perform about the same but do not scale at all. In this case, the messages are rather small due to the memory layout of the matrices. Every row of each matrix has to be sent separately, consisting of 64 double precision floating point elements of 8 bytes each, resulting in a message size of 512 bytes. Using the original and direct implementations, each of these messages will be sent separately, resulting in a lot of TCP/IP overhead. The remapping and copy core implementations show some scalability as they only receive a list of addresses that they have to copy their data from. The L2 cache also does not have to be flushed for every message, as it recognizes that all these messages together are part of the same remote computation.

For size 256, (Figure 5b), we still do not see a lot of speedup for the original and direct implementation. The data size per message has increased to 1KB, which is a clear advantage to the remapping and copy core approaches, where the measurement with 4 copy cores manages to nearly gain a perfect speedup of 8. For size 512, (Figure 5c), the direct implementation starts to show some scalability for multiple cores, scaling up to a speedup of 4. The remapping and copy core approaches scale well and perform roughly the same, though the latter shows some superlinear speedup for 4 cores, probably as the data fits well into the L2 cache. The last graph (Figure 5d) shows similar results, except that the remapping implementation is now clearly outperformed by the copy cores as they provide more communication bandwidth. The original implementation again scales poorly, which is caused by the different ratio between communication bandwidth and computational power, compared to a cluster environment.



(a) Matrix size 128×128.

(b) Matrix size 256×256.



(c) Matrix size 512×512.

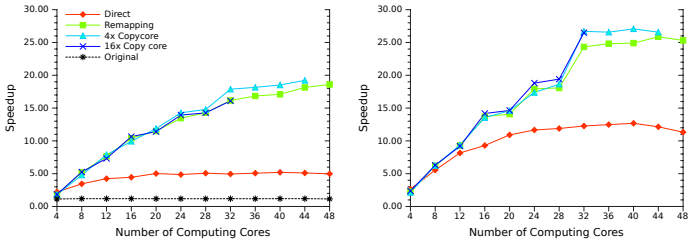
(d) Matrix size 1024×1024.

Figure 5: Matrix multiply with 1 decomposition step

3) Matrix multiply using two decomposition steps: The matrix multiplication benchmark exposes eight times the concurrency for each decomposition step that is performed. However, the additional recursion step leads to more communication overhead due to increased number of messages. We ran this benchmark for sizes 1024×1024, and 2048×2048, but the original implementation could not run on the latter size due to memory constraints.

In Figure 6, we see no speedup for the original implementation when using additional cores. It fails to scale as the master node is fully occupied with the distribution of tasks while most of the workers are idle waiting for work. This also limits the scalability of the direct approach to about a factor of 5 at 20 cores, but this is not the case for remapping or copy cores. As these two approaches fetch the memory, more concurrency in the communication is exposed when the number of workers is increased, resulting in more scalable communication and corresponding speedups, peaking at 27 with the copy core approach. The master node only needs

to send a set of addresses which reduces the communication time for the master so it can distribute tasks faster. The copy core approach performs slightly better then remapping with more clients as the master node still becomes a bottleneck on receiving back the result of the computations. Note that computations using the copy core approaches can not be run on all 48 cores as some cores are reserved for the copy tasks.

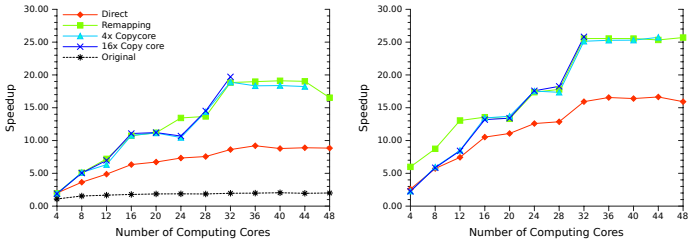


(a) Matrix size 1024×1024. (b) Matrix size 2048×2048.

Figure 6: Matrix multiply with 2 decomposition steps

4) Recursive decomposition over multiple nodes: A solution that decreases the load of a single master node, is to split the recursion steps over multiple nodes. The master node performs one decomposition step and delegates the next decomposition to other nodes. These nodes then perform the second step and distribute the work over even more nodes, using a round robin algorithm that guarantees an as much even distribution as possible. This approach introduces a lot of additional communication, but not much computational overhead as the decomposition on a single node can be done without additional copy operations due to the way we structure our matrices in memory.

The benchmark results are shown in Figure 7, where again the original implementation was unable to run the 2048 configuration. The original and direct approaches clearly benefit from the different communication pattern, while the remapping and copy core approaches perform about the same as with the other communication pattern, peaking at a factor 25 speedup.



(a) Matrix size 1024×1024. (b) Matrix size 2048×2048.

Figure 7: Matrix multiply with distributed decomposition.

V. Conclusion

We have shown our initial results of porting our implementation of the SVP model of concurrency to the Intel SCC. One of the biggest problems was the efficient communication of data; it is difficult to keep all the cores busy and to find a good communication to computation ratio.

We have discussed several approaches on how we improved our communication bottleneck; removing XDR encoding, remapping and copying data directly at the receiving core, and employing our copy core techniques. The latter showed a two orders of magnitude improvement in throughput, and has the potential to scale up by employing multiple copy cores. However, the matrix multiply benchmarks that we used were not able to effectively use the large bandwidth provided by the copy core techniques compared to the remapping approach.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108–109, February 2010.

[2] C. R. Jesshope, "A model for the design and programming of multi-cores," *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[3] K. Bousias, N. Hasasneh, and C. Jesshope, "Instruction level parallelism through microthreading—a scalable approach to chip multiprocessors," *Comput. J.*, vol. 49, pp. 211–233, March 2006.

[4] J. Sykora, L. Kafka, M. Danek, and L. Kohout, "Analysis of execution efficiency in the microthreaded processor UTLEON3," in *Proceedings of the 2011 Conference on Architecture of Computing Systems (ARCS 2011)*, vol. 6566 of *Lecture Notes in Computer Science*, pp. 110–121, Springer, 2011.

[5] M. W. van Tol and J. Koivisto, "Extending and implementing the self-adaptive virtual processor for distributed memory architectures," *CoRR*, vol. abs/1104.3876, April 2011.

[6] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *3rd Many-core Applications Research Community (MARC) Symposium*, KIT Scientific Publishing, September 2011.

[7] 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.

[8] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, (Istanbul, Turkey), July 2011.

[9] Intel Labs, *SCC External Architecture Specification*, revision 1.1 ed., November 2010.

[10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ISCA '92: Proc. of the 19th annual Int. Symp. on Computer architecture*, (New York, NY), pp. 256–266, ACM, 1992.

[11] C. R. Jesshope, M. Lankamp, and L. Zhang, "Evaluating CMPs and their memory architecture," in *Proc. Architecture of Computing Systems* (M. Berekovic, C. Muller-Schoer, C. Hochberger, and S. Wong, eds.), pp. 246–257, 2009.

[12] T. D. Vu and C. R. Jesshope, "Formalizing sane virtual processor in thread algebra," in *ICFEM*, pp. 345–365, 2007.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, (New York, NY, USA), pp. 15–26, ACM, 1990.

[14] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "Cellss: a programming model for the cell be architecture," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 86, ACM, 2006.

[15] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 83, ACM, 2006.

[16] M. Eisler, "XDR: External Data Representation Standard." RFC 4506 (Standard), May 2006.