X10 on the Single-Chip Cloud Computer

Porting and Preliminary Performance

Keith Chapman Ahmed Hussein Antony L. Hosking
Purdue University, West Lafayette, Indiana
{keith,hussein,hosking}@cs.purdue.edu

Abstract

The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. SCC is essentially a 'cluster-on-a-chip', so X10 with its support for places and remote asynchronous invocations is a natural fit for programming this platform. We report here on our experience porting X10 to the SCC, and show performance and scaling results for representative X10 benchmark applications. We compare results for our extensions to the SCC native messaging primitives in support of the X10 run-time, versus X10 on top of a prototype MPI API for SCC. The native SCC run-time exhibits better performance and scaling than the MPI binding. Scaling depends on the relative cost of computation versus communication in the workload used, since SCC is relatively underpowered for computation but has hardware support for message passing.

1. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research [MARC; Mattson et al. 2010]. This processor explores a scalable many-core architecture that dispenses with hardware support for cache-coherent shared memory. SCC uses a mainstream x86 instruction set, runs Linux, and has compilers supporting C, C++, and Fortran. It also supports a simple message-passing API called RCCE, allowing point-to-point synchronous communication over an on-die mesh connecting the cores. Each core boots its operating system independently of the other cores so that, in concert with the mesh network, the processor can be programmed as a 'cluster on a chip'. Messages over the network coordinate processes running on the cores, and communicate data among those processes.

SCC is a natural platform for the X10 language [Charles et al. 2005]. X10's places are a direct abstraction for SCC's distributed cores and for communication among them. X10's data distribution constructs naturally support partitioning of work among SCC cores. Supporting X10 on SCC can be achieved in a number of ways. The X10 run-time can be deployed as either native code (using the C++ backend) or as Java bytecode (using the Java backend). The C++ backend is currently more mature (though the Java backend is rapidly gaining), which suits our porting effort because of SCC's established support for C++.[1]

For transport, the X10 run-time already supports various options, including sockets and MPI. Unfortunately, X10's sockets transport is built above SLURM (Simple Linux Utility for Resource Management), which is not currently available on SCC so we cannot use that. Intel provided us with a prototype of an MPI API for SCC, which proved capable of running some X10 programs (but not all reliably). However, we were also interested in exploring direct use of SCC's native message passing API known as RCCE (pronounced "rocky"). Indeed, we have had good success with RCCE, though we have made significant changes and extensions to it to match the needs of the X10 run-time which we call RCCE-X10. We describe our experiences with both the MPI on SCC prototype and with RCCE-X10 and compare their performance running several X10 benchmark applications.

The rest of this paper is organized as follows. We review the features of SCC in Section 2, then describe our port of X10 to SCC in Section 3 including the details of our RCCE-X10 extensions to the RCCE native message passing API for SCC. Section 4 presents the results of running several X10 benchmark applications on SCC, comparing both the MPI and RCCE-X10 transport layer bindings, and replicating the results on a stock Linux cluster as a sanity check. These applications demonstrate good scalability for some input workloads, though we note that other inputs do not yield such good results. Section 5 offers conclusions and directions for further work.

2. SCC

The SCC chip is a many-core CPU comprising 24 dual-core x86 tiles connected via a 2D-grid on-die network. The tiles are arranged in a 6 by 4 mesh with each tile containing two blocks. Each block implements a Pentium P54C processor core, 16-Kbyte L1 caches for instructions and data, and a unified 256-Kbyte L2 cache. Each tile has a mesh interface unit (MIU) which allows the mesh and the interface to run at different frequencies. Each tile also has a 16-Kbyte message passing buffer (MPB) for fast communication between tiles, and two test-and-set registers (one per core) for synchronization among cores.

The cores are second-generation P54C Pentium processors, which execute in-order. The P54C does not support the SSE instructions available on later Pentium processors. Each tile connects to a router via the MIU, which packetizes data to and from the mesh. The MIU handles data cache misses by translating the 32-bit memory address of the core to a wider system address that allows access to up to 64 Gbytes of (non-coherent) off-chip memory, accessible via four DDR3 memory controllers. A router is also con-

[1] SCC does run stock Java virtual machines, so in future we plan to focus on the Java backend.

X10'11, June 04 2011, San Jose, CA, USA
Copyright © 2011 ACM 978-1-4503-0770-3/11/06. . . $10.00.

nected to an off-package FPGA which translates the mesh protocol into the PCI Express protocol so that the chip can interact with a PC management console. Voltage and frequency can be controlled per-tile and on the mesh, for a total of 25 frequency domains.

The MPB is in fast on-die SRAM, as opposed to the system memory accessed through the four DDR3 controllers. Pages in the P54C's page table have a special reserved bit to mark MPB data, which is cached in L1 but bypasses L2. A new instruction allows invalidation of all MPB lines in the L1 to force subsequent accesses to memory.

A common system configuration (and the one we use) views the address space of a core in three regions:

1. Private off-chip DRAM regions associated one per core. The system is configured so that each of these regions are accessible only by one core. This corresponds to the main memory in a conventional system. On the SCC machine we have access to these private memories are configured to be around 322 Mbytes per core.
2. Shared off-chip DRAM configured as uncacheable to avoid consistency issues, used for direct access to each core's register file by SCC's monitoring and configuration tools.
3. Shared on-chip SRAM mapped by all cores as MPB (so any core can see another's MPB) and marked as MPB data so it can be cached in L1. On our SCC machine we have 8 Kbytes of MPB per core.

As previously noted, each core also has a test-and-set register to support inter-core locking for synchronization.

A TCP/IP driver also permits communication among the cores over the mesh via the usual TCP/IP stack and to provide access to a console for each core. This also allows the cores to access a shared memory-mapped NFS file system.

2.1 RCCE native message passing API

The RCCE native message passing API is based on one-sided put and get primitives which move data from private memory through the L1 cache of the sending core via the MPB to the L1 cache of the receiving core, without having to go off-chip with the data. While the MPB is physically distributed among the tiles, it is logically a single shared address space so any core can access any address in the MPB. The MPB is logically divided into 8 Kbyte contiguous blocks, one per core, created by equally dividing the 16 Kbyte MPB on each tile. A portion of each 8 Kbyte segment can be devoted to flags for coordinating communication between cores, while the rest is available for sending and receiving messages. Because there is no coherence, the flags must be used for explicit synchronization.

RCCE also provides a pair of two-sided synchronous send and receive functions that support two-sided communication. These functions block until matching calls on both sides complete execution. A private memory buffer on the sending side is packetized through the MPB to a private memory buffer on the receiving side. Whereas in MPI one can use asynchronous communication to avoid deadlock, RCCE's synchronous communication requires that deadlock be avoided by design.

3. X10 on SCC

Running X10 on SCC is relatively straightforward. We use the X10 C++ backend and compile using the SCC C++ cross-compiler. Because the SCC does not support SSE we disable SSE instructions in the X10 build. Because X10 already has an MPI transport implementation we are able to take advantage of a prototype MPI API provided by Intel, and run some X10 applications 'out-of-the-box'. Unfortunately, X10 seems to tickle some residual bugs in the MPI API, which we are trying to resolve with help from Intel. As a result, our experiments using this transport are not complete.

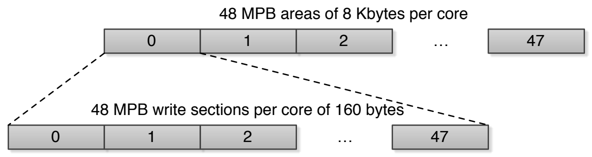

48 MPB areas of 8 Kbytes per core
0
1
2
...
47
48 MPB write sections per core of 160 bytes
0
1
2
...
47

Figure 1: MPB allocation for SCC MPI over MPB

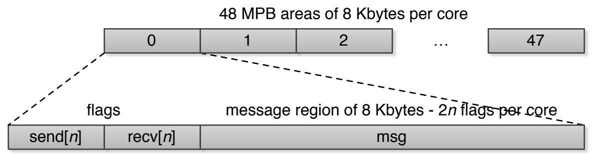

48 MPB areas of 8 Kbytes per core
0
1
2
...
47
flags
message region of 8 Kbytes - 2n flags per core
send[n]
recv[n]
msg

Figure 2: MPB allocation for RCCE-X10

In addition, the X10 C++ backend uses the Boehm garbage collector, which we have yet to enable successfully on SCC (because of C/C++ library incompatibilities). As a result we must run without garbage collection, which means the range of applications we can run is limited by the available memory on the SCC. Our SCC machine allocates 322 Mbytes of private DRAM per core. Of course, limited memory also means that we can often not run workloads that provide sufficient work to keep all cores occupied. This means that we must strike a fine balance between work and memory footprint to demonstrate useable scaling.

Our primary interest in X10 on SCC was to explore how best to make use of the underlying hardware support for messaging between cores using the on-die SRAM MPB, the integrated per-tile MIUs, and the mesh. While the MPI API provided by Intel does use this underlying hardware support, we wanted to tailor the hardware-supported messaging to X10's run-time conventions as best we could. Because the SCC hardware does not map to the MPI communication mechanisms directly, it makes sense to implement a native SCC-based transport that better fits the X10 run-time. We first describe the SCC MPI library implementation, and then our own `RCCE-X10` native implementation.

3.1 SCC MPI library

The SCC MPI API is an effort to provide lightweight MPI in support of MPI applications running out-of-the-box on SCC. It has been implemented to support message passing over multiple transports:

mpi-mpb: direct use of the MPB and MIU to communicate between cores;

mpi-sock: using sockets through the TCP/IP stack (which communicates via the TCP/IP driver for the mesh);

mpi-shm: using shared memory to pass messages.

We are unable to use the mpi-shm transport because it depends on a kernel patch that is not yet available to us. Thus, our experiments include only mpi-mpb and mpi-sock.

3.1.1 SCC MPI over MPB

Each core on the SCC has 8 Kbytes of MPB. In the MPI over MPB (mpi-mpb) implementation each core divides its 8 Kbytes of MPB into 48 sections of 160 bytes for direct use by other cores to write incoming messages as shown in Figure 1. Thus, a core can simultaneously communicate with multiple cores. When sending a

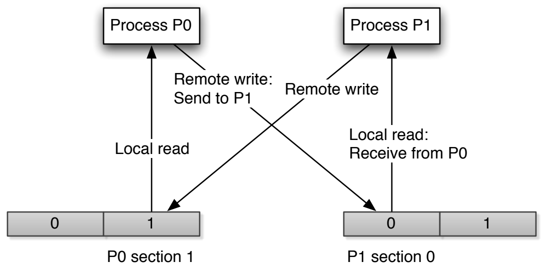


Figure 3: Sending and receiving a message with SCC MPI over MPB

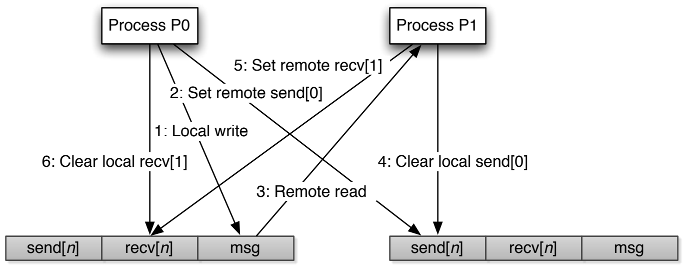


Figure 4: Sending and receiving a message with RCCE-X10

message a core writes to the exclusive write section reserved for it on the receiving core (see Figure 3). The receiver then reads the message locally. This permits both sends by writing to the remote target and receives by reading locally. The per-core test-and-set registers support synchronization of actions on the MPB sections.

3.1.2 Limitations of SCC MPI

Of course, each of the MPI implementations constrains clients to use the MPI API, which may not be a good match for application needs. Directly accessing the hardware via the RCCE APIs exposes the limitations (and efficiencies) of the hardware messaging support that should enable better performance so long as applications can use the RCCE APIs effectively.

3.2 RCCE-X10: A native X10 run-time for SCC

RCCE-X10 is our extension to the RCCE API optimized for the X10 run-time. It introduces new functionality not present in RCCE and tailors existing functionality to suit X10 better.

Unlike SCC MPI, RCCE-X10 does not define exclusive write sections. Instead, the sender deposits a pending message into its local MPB and the receiver reads it remotely. We use instances of `RCCE_FLAG` for explicit synchronization between cores. Each core has n send and n receive flags in its MPB, where n is the number of cores participating in the computation. The remainder of the MPB is used for each outgoing message. The send flags notify a receiver that a message is pending from the corresponding sender, which it can read remotely from the sender's MPB. The receive flags let the receiver acknowledge receipt of the message to the sender. In this way, while a sender is waiting for its message to be delivered it can still probe for any incoming messages. This organization is illustrated in Figure 2.

RCCE allows the programmer to specify whether each flag occupies a whole MPB 32-byte cache line, or only a single bit within a line to save space for increased bandwidth. However, using a single bit requires implementing atomicity of flag access using the test-and-set register of the core, resulting in higher latency. Thus, in our experiments we use a whole cache line per flag to optimize for latency. Thus, the space allocated in the MPB for each core's outgoing message is 8 Kbytes minus $32 \times 2n$ bytes for n cores. Messages larger than this must be broken down and sent in chunks.

3.2.1 Limitations of RCCE

From the perspective of the `X10RT` layer the default RCCE API has some limitations. RCCE-X10 was designed to mitigate these limitations, as follows:

- The `RCCE_send` call blocks until a matching receive has been posted. `X10RT` requires sends not to block receipt of incoming messages or else deadlocks can occur, so we implemented a send call that probes for incoming messages while waiting for the send to finish.
- The `RCCE_recv` function requires specifying the node from which to receive the message. In contrast the `X10RT` layer is not aware of which place it wants to receive a message from and simply probes for an incoming message from any place.
- The `X10RT` layer sends and receives several kinds of messages. When a message arrives the kind of message dictates which callback will be used to handle the message. Also, `X10RT` uses put/get operations to transmit data from/to a predetermined memory location. This requires probing for the details of the message payload. To support both of these cases we add a message header to distinguish the kind and size of the message.
- The `X10RT` layer uses `x10rt_probe` to probe for a message and receive it without blocking. We implement a new `RCCE_X10_iprobe` operation to do this. When probing detects a message we use our own receive function to retrieve the message. We cannot use the existing RCCE operation `RCCE_recv_test` because it is similar to `RCCE_recv` in needing to know the sending node from which to test for an incoming message.

3.2.2 Sending and receiving

Figure 4 shows the sequence of steps required to deliver a message in RCCE-X10. A sender notifies a receiver that a message is pending by setting its corresponding remote send flag in the receiver. A receiver acknowledges receipt of a message by setting its corresponding remote receive flag in the sender. Senders wait for acknowledgment by busy-waiting on the corresponding local receive flag. Receivers poll the corresponding local send flag for incoming messages.

Sending. Sending involves the following steps:

1. Write the message into the local MPB.
2. Set the send flag corresponding to this sender in the remote MPB of the receiver. This indicates to the receiver which process has a message for it.
3. Busy-wait on the receive flag corresponding to the receiver in the local MPB, until it has acknowledged receipt of the message by remotely setting the flag. At each iteration the sender checks its local send flags to see if it has any messages pending from other processes. If there is an incoming message then read it into a local buffer but defer passing it to the corresponding X10 callback. Only release the message to the callback when the X10 run-time executes an `x10rt_probe` call.
4. Clear the receive flag corresponding to the receiver in the local MPB.

Large messages require repeating these steps as many times as necessary to send each chunk of the message.

Receiving. When `x10rt_probe` is called we must check all the send flags in the local MPB. If any of them are set then the corresponding senders have messages that should be retrieved. Having detected an incoming message, the receiver executes the following steps:

1. Read the message from the sender's remote MPB.
2. Clear the send flag corresponding to the sender in the local MPB.
3. Set the receive flag corresponding to this receiver in the remote MPB of the sender.
4. If all the chunks of the message have been received then return. Otherwise busy-wait on the send flag corresponding to the sending process until it deposits the next chunk into its MPB, then repeat the steps.

3.2.3 Limitations of RCCE-X10

There are a number of limitations of this implementation of RCCE-X10. First, sending a message takes full control of the local MPB to hold the message, and this cannot be relinquished until the receiver has read the message. We use a lock around the send function, which prevents multiple threads at the same process from sending messages concurrently.

Second, the X10 RT specification recommends that the `x10rt_probe` function should not block waiting for network traffic to arrive, or else performance can be affected. We do not buffer messages in RCCE-X10, so once a node starts receiving a message it must wait until the whole message has been received. If a message is larger than the available MPB space then `x10rt_probe` will block until all the chunks have been received and processed. Thus, bulk transfers of large amounts of data using put/get operations can degrade the latency of other messages. In future we plan to use shared memory to transmit large messages to avoid this bottleneck.

4. Experiments

Our experiments explore the performance of several X10 benchmarks running on the SCC using our native `RCCE-X10` message passing run-time, the default X10 sockets run-time, as well as using the prototype SCC MPI implementation. The current choice of benchmarks is constrained by the current X10 port. We are unable to run with garbage collection enabled (the garbage collector used by the X10 C++ backend currently crashes on SCC because of incompatibilities with SCC's rather old C library). Thus, the workload must operate within the constrained 322 Mbytes per core of memory available on the SCC. In tension with this constraint, to achieve scalability we must also use workloads that perform sufficient work on each core to justify the overheads of communication. But larger workloads tend to require more memory. Thus, we currently have only two benchmarks running on the SCC, but we plan soon to have results for others, especially as we are able to get garbage collection to work.

4.1 Platforms

On the SCC we disable SSE instructions in the X10 build, and run without garbage collection. For all our experiments the SCC cores were running at 533 MHz and the mesh at 800 MHz with 800 MHz DDR3. Each core has 322 Mbytes or private memory. We use SCC Linux version 2.6.16-Rev13_unchecked, `libc` version 2.3.6, cross-compile X10 using gcc version 3.4.5 configured as i386-unknown-linux-gnu, use the GNU Scientific library `libgsl` version 1.14, and base our implementation on RCCE trunk revision 69. We use X10 revision 19448 post-2.1.1 trunk, dated January 13, 2011.

In addition to the SCC platform, we also run the benchmarks on our AMD Opteron cluster as a means of validating our experiments and setup. The AMD cluster comprises three 16-way multiprocessors connected via private Gigabit Ethernet, each with eight dual-core AMD Opteron 865 processors running at 1.8 GHz and 32 Gbytes of RAM. On the AMD cluster we compile the X10 benchmarks and run-time using gcc version 4.4.0. We use library `libgsl` version 1.13-3 and are able to run with the garbage collector library `libgc` version 7.1. We use X10 revision 20089 post-2.1.1 trunk, dated February 13, 2011. It is configured to use the MPI transport layer on the cluster.

On both the AMD cluster and SCC we set `X10_NTHREADS=1` so that every core has only one worker thread.

4.2 Benchmarks

We were able to run two representative X10 benchmarks on SCC. One is a molecular chemistry application, and the other is a social networks application.

4.2.1 ANUChem HF

Milthorpe et al. [2011] have implemented the Hartree-Fock (HF) method in X10 as part of a suite of X10 benchmarks [Australian National University]. Hartree-Fock is used in quantum chemistry to solve the pseudo-eigenvalue problem $FC = \varepsilon SC$, where F, C and S are the Fock, molecular orbital Coefficient and Overlap matrices, respectively [Szabo and Ostlund 1989]. The dimensions of these matrices depend on the number of basis functions used to represent the molecular system, and so indirectly on the number of atoms in the system. HF solves for C, with a known constant matrix S. Because F is dependent on C, HF uses an iterative Self Consistent Field (SCF) method. The bulk of the work in HF is setting up the Fock matrix F:

$$F_{ij} = H_{ij}^{\text{core}} + \sum_{k,l}^{N} P_{kl}[\langle ij|kl\rangle - \frac{1}{2}\langle ik|jl\rangle]$$

where i, j, k, l denote atom centered basis function indices, $\langle ij|kl\rangle$ is a four centered two-electron integral and H^{core} is a matrix containing one-electron integrals. P is the density matrix and is obtained from C.

In the X10 implementation, the density matrix is replicated at all places by copying the array as part of the "active message" that initiates computation of the partial contribution to the F matrix. Load balancing of the two-electron integral evaluations is the main concern in HF. Milthorpe et al. [2011] consider several alternatives to achieving this. One uses static load balancing, with a coarse granularity of work mapped using `async` and `at` over the available places in an outer loop, and results gathered and reduced at the end using `finish`. Another approach uses dynamic load balancing with a shared counter, as suggested by Shet et al. [2008], implemented as a global atomic integer manipulated using futures. Each of the places iterates over all work tasks. When the local iteration counter matches the global counter, that unit of work is performed at the current place, otherwise it is skipped. This method involves frequent communication to the first place where the global counter is maintained, which may degrade scaling.

Milthorpe et al. [2011] report results for HF running on a Blue Gene/P with 850 MHz PowerPC 450 processors. They report only results for the static load balancing technique, stating that this is the only scheme that runs to completion and shows positive scaling. In contrast, we are able to run with the dynamic load balancing scheme and observe scaling, reporting our results below.

Our experiments use revision 939 of the 2.1.2 branch of ANUChem dated January 25, 2011. We report results on SCC for the Benzene STO-3G workload. We are unable to run the Benzene 3-21G workload used by Milthorpe et al. [2011] on the SCC, but we do report results for Benzene 3-21G on our AMD Opteron Linux cluster.

4.2.2 BC

Betweenness Centrality (BC) [Anthonisse 1971; Freeman 1977] is used in the analysis of social networks to rank actors according to their prominence within the network. High centrality scores indicate that a vertex in a graph can reach others on relatively short paths, or that a vertex lies on considerable fractions of shortest paths connecting others. Brandes [2001] shows that betweenness can be computed exactly and efficiently, even for reasonably large networks.

Brandes computes betweenness centrality by counting shortest paths using Dijkstra's algorithm for weighted graphs and breadth-first search for unweighted graphs. Given a graph of vertices V and edges E, a *path* from vertex s to vertex t is an alternating sequence of vertices and edges beginning with s and ending with t, such that each edge connects its preceding vertex with its succeeding vertex. Let n and m represent the number of vertices and edges, respectively. Let w be a weight function on the edges, such that $w(e) > 0$ for every edge e. In an unweighted graph $w(e) = 1$. The *length* of a path is the sum of the weights of all the edges in that path. Let σ_{st} denote the *distance* between vertices s and t (by convention $\sigma_{ss} = 1$). Finally, $\sigma_{st}(v)$ denotes the number of shortest paths from s to t that some vertex v lies on. The centrality formula used is:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Brandes describes an algorithm that requires $O(m+n)$ space, and runs in $O(nm)$ and $O(nm + n^2 \log n)$ time on unweighted and weighted graphs, respectively.

The X10 implementation of this algorithm is available in the X10 benchmarks repository. It generates directed, weighted, and bipartite graphs as described by Chakrabarti et al. [2004]. In our experiments we use the version of BC at revision 19602 of the X10 benchmarks trunk, dated January 18, 2011.

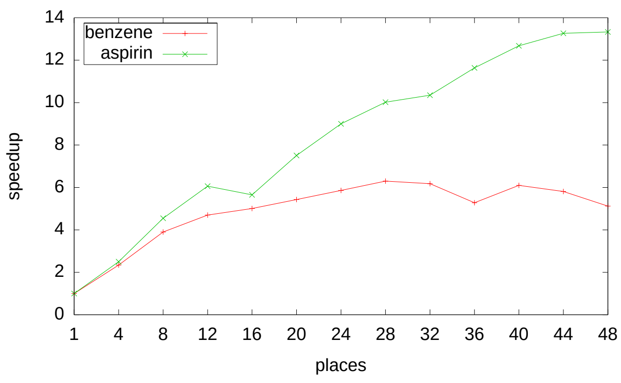
4.3 Results

On SCC we compare the performance of the prototype SCC MPI implementation, using both MPB (mpi-mpb) and sockets transports (mpi-sock), against our RCCE-X10 extended native transport and the default X10 sockets runtime (recall that even sockets uses the native TCP/IP driver for the mesh). Residual bugs in mpi-mpb mean that we cannot run for a single place with BC. For HF we cannot run less than eight places because of memory constraints (we must distribute the work across eight or more places just to fit). Moreover, because of the bugs in mpi-mpb we are unable to run for the same numbers of places as for RCCE-X10 and mpi-sock, so we fill in the results at places where we were able to run successfully. In all experiments we report the mean of three separate benchmark runs.

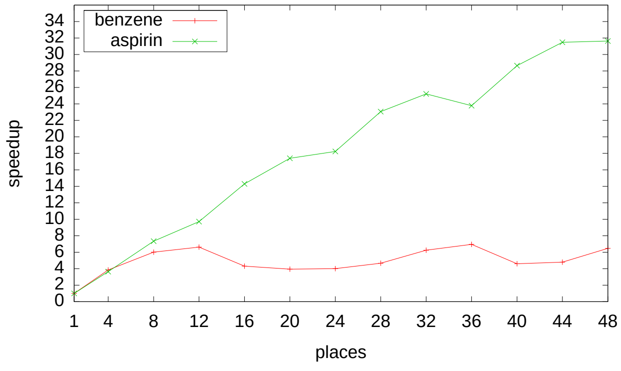
4.3.1 HF results

We use both the static and dynamic load balancing alternatives for HF described by Milthorpe et al. [2011]. These are controlled by parameter `gmatype`: dynamic is `gmatype` 4, and static is `gmatype` 5.

Figure 5 shows speedup results when running HF for the workloads Benzene 3-21G (benzene molecule using the 3-21G basis set) and Aspirin 3-21G (aspirin compound using the 3-21G basis set) on the AMD Opteron cluster with dynamic and static load balancing. Aspirin 3-21G shows better scaling compared to Benzene 3-21G. Frequent communication to the first place and the comparably smaller workload in Benzene 3-21G act to impede scaling with



(a) Dynamic load balancing



(b) Static load balancing

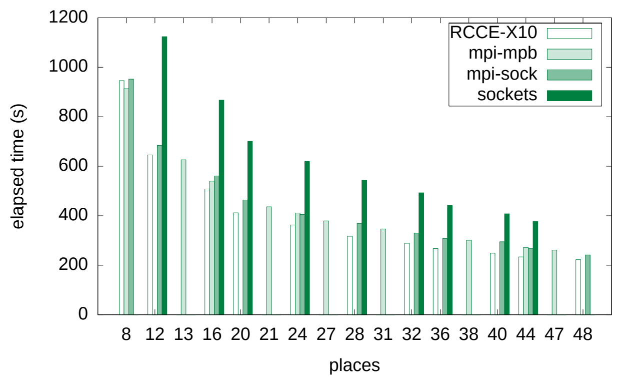
Figure 5: HF speedup on AMD cluster with MPI (Benzene 3-21G and Aspirin 3-21G)

both dynamic and static load balancing, whereas the heavier workload of Aspirin 3-21G gets speedup scaling to 13× on 48 nodes with dynamic load balancing. For static load balancing there is an even more pronounced gap between the speedup of Benzene 3-21G and Aspirin 3-21G; the peak for Benzene 3-21G is 6× at 12 places, while Aspirin 3-21G reaches 31× at 44 places.

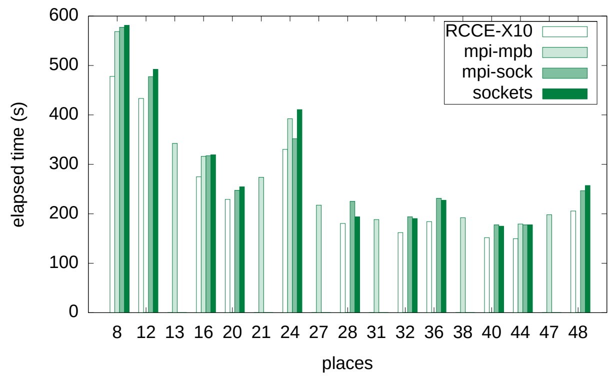
Because X10 does not run with garbage collection on the SCC, and because of its smaller per-core memory, the workloads we can run there are restricted. We had to strike the balance between having sufficient work and fitting within the available memory. Benzene STO-3G is the workload that both runs and shows scaling. Unfortunately, it does not run for fewer than eight places due to memory constraints. For this workload 576 work units are processed when static load balancing is used.

Figure 6 shows elapsed times for running HF Benzene STO-3G with both dynamic and static load balancing. Dynamic load balancing is costly for few places, but at 48 places it gives similar performance to static load balancing. RCCE-X10 has superior performance across the full range of places. It is worth pointing out that our elapsed times for Benzene STO-3G running on SCC are an order of magnitude slower than on the AMD Opteron cluster. Of course, the unit cost of our cluster is approximately two orders of magnitude greater than the cost of an SCC installation.

Scaling is better for RCCE-X10 with both dynamic and static load balancing as shown in Figures 7a and 7b. Static load balancing is less smooth in distributing work, but it scales similarly to dynamic load balancing (though static performs better generally), as shown in Figure 7c. This raises an interesting point regard-



(a) Dynamic load balancing



(b) Static load balancing

Figure 6: HF Benzene STO-3G elapsed time on SCC

Table 1: Parameters for BC benchmark workloads

(a) AMD Opteron cluster with MPI

	n	w	a	b	c	d	verts.	edges
low	14	1	0.55	0.1	0.1	0.25	16384	123258
high	15	1	0.55	0.1	0.1	0.25	32768	249842

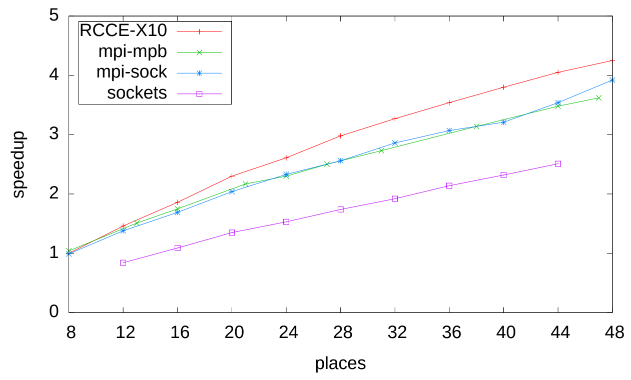
(b) SCC

	n	w	a	b	c	d	verts.	edges
small	11	1	0.2	0.15	0.25	0.4	2048	16250
medium	14	1	0.65	0.1	0.1	0.15	16384	105353
large	14	1	0.55	0.1	0.1	0.25	16384	123258

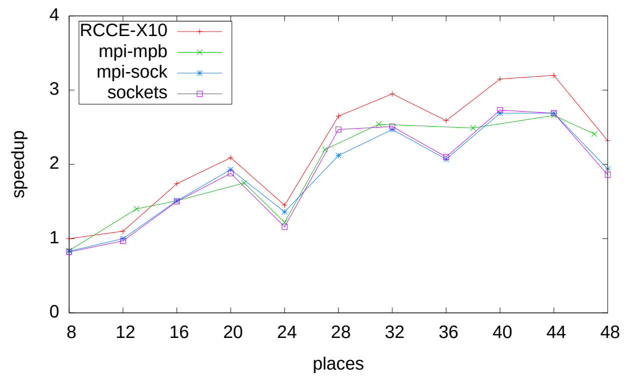
ing the lack of built-in support for work-stealing across places in X10, making load-balancing a difficult problem that the programmer must deal with explicitly. Note that the default X10 sockets transport does not perform as well as the other transports for this workload.

4.3.2 BC results

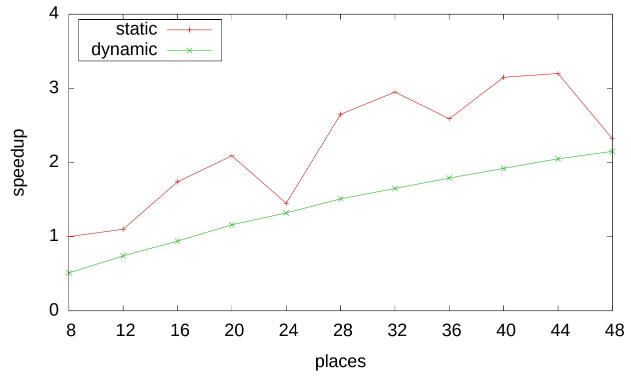
Table 1 shows the parameters for the BC workloads on the AMD Opteron cluster and on the SCC. The parameter n controls the number of vertices in the graph, 2^n. The parameter $w = 1$ indicates that the graph is *weighted*. The parameters a, b, c, d control the generation of the graph using the model of Chakrabarti et al. [2004]. The important graph attributes for our purposes are the size in number



(a) Dynamic load balancing (normalized to RCCE-X10)



(b) Static load balancing (normalized to RCCE-X10)



(c) Dynamic versus static on RCCE-X10 (normalized to static)

Figure 7: HF Benzene STO-3G speedup on SCC

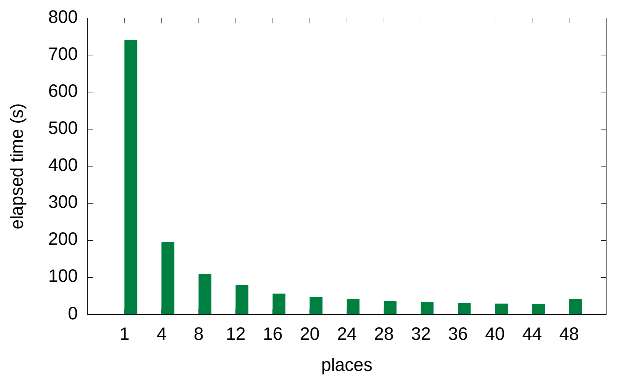
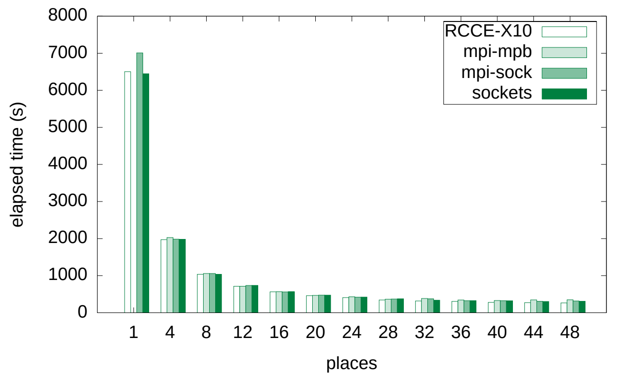
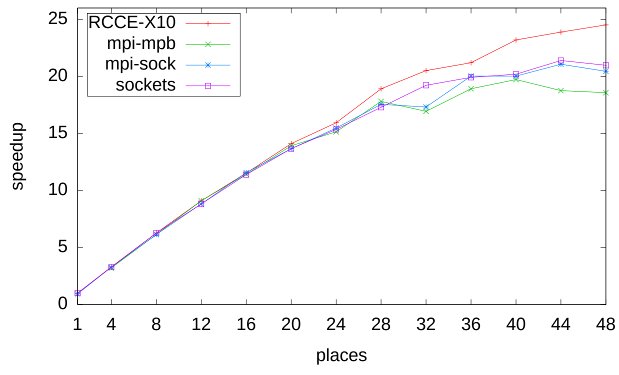


Figure 8: Elapsed time for BC on AMD Opteron cluster with MPI for the low(cluster)=large(SCC) workload



(a) Elapsed time

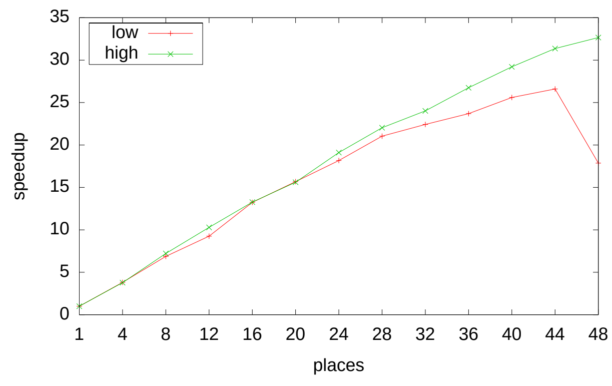


(b) Speedup (normalized to RCCE-X10)

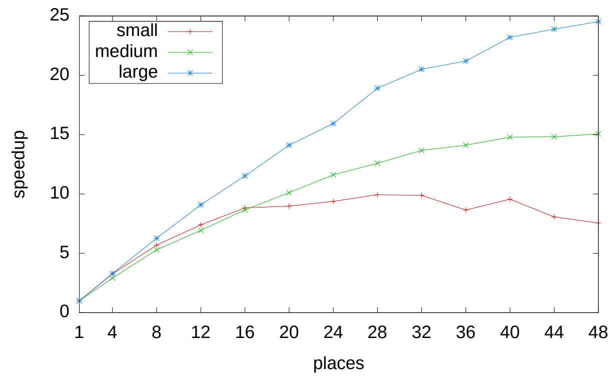
Figure 9: BC on SCC with the large workload

of vertices and number of edges, since these reflect the memory footprint when running on SCC. Note that the low workload for the AMD Opteron cluster is the same as the large workload for SCC. We ran three different workloads for BC on the SCC. We started with a small workload and tried pushing it until it started failing with memory errors.

Figure 9 shows execution of the large workload for the different transports. For comparison, observe the elapsed times for execution of BC on the AMD Opteron cluster, shown in Figure 8. In Figure 9a we see the elapsed times for each transport, revealing that RCCE-X10 has a slight advantage across the full range of places. In Figure 10 we see that all the transports scale similarly, with RCCE-X10 having smoother speedup and scaling further than the other two. Both mpi-mpb and mpi-sock start to tail off at around 36 places, whereas RCCE-X10 continues scaling to 25× at 48 places. Unlike HF, the default X10 sockets transport and the mpi-sock transport seem to track each other rather closely for BC.



(a) AMD Opteron cluster with MPI



(b) SCC with RCCE-X10

Figure 10: BC speedup for varying workloads

Figure 10 shows how speedup varies with workload. As expected the higher the load the better the speedup. Notice that the large SCC workload, which is the same as the low AMD cluster workload, scales similarly on both SCC and the AMD Opteron cluster, except at 48 places where the cluster drops off significantly.

5. Conclusions

We have implemented a native run-time for X10 running on the SCC experimental 'cluster-on-a-chip' processor. Our native RCCE-X10 run-time has performance and scalability that is superior to both a proprietary MPI API for SCC and default X10 sockets as the transport layer for X10. Scalability for representative benchmarks is good so long as sufficient work is available to partition among the cores. One difficulty with SCC is coping with the memory constraints, because this can weigh against generating sufficient work. Nevertheless, X10 and SCC are a good match. X10 eases the task of the application programmer in developing cluster applications that scale, while SCC provides a cost-effective platform for running such applications. X10 is a good fit for programming SCC (and similar processors) as illustrated by the good scaling we see for the benchmarks we used.

Acknowledgments

We thank Vijay Saraswat of IBM, and Josh Milthorpe and Daniel Frampton of the Australian National University, for assistance with X10 benchmarks and benchmarking. We also thank Intel for making the SCC prototype available to us, and especially Ted Kubaska for providing us with the SCC MPI implementation, and for answering our many questions. This work is supported by the National Science Foundation under grants Nos. CCF-0702240, and CCF-0811691, and by Microsoft, Intel, and IBM. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

J. M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, Amsterdam, Oct. 1971. URL `http://oai.cwi.nl/oai/asset/9791/9791A.pdf`.

Australian National University. ANUChem benchmarks. URL `http://cs.anu.edu.au/~Josh.Milthorpe/anuchem.html`.

U. Brandes. A faster algorithm for betweenness centrality. *Mathematical Sociology*, 25(2):163–177, June 2001. doi: `10.1080/0022250X.2001.9990249`.

D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicor, editors, *Proceedings of the Fourth SIAM International Conference on Data Mining*, Lake Buena Vista, Florida, Apr. 2004. URL `http://www.siam.org/proceedings/datamining/2004/dm04_043chakrabartid.pdf`.

P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, San Diego, California, Oct. 2005. doi: `10.1145/1094811.1094852`.

L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, Mar. 1977. URL `http://www.jstor.org/stable/3033543`.

MARC. Many-core applications research community. URL `http://communities.intel.com/community/marc`.

T. G. Mattson, 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 on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, Louisiana, Nov. 2010. doi: `10.1109/SC.2010.53`.

J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a parallel language for scientific computation: practice and experience. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, May 2011.

RCCE. A communication environment for the SCC processor. URL `http://marcbug.scc-dc.com/svn/repository/trunk/rcce`.

A. G. Shet, W. R. Elwasif, R. J. Harrison, and D. E. Bernholdt. Programmability of the HPCS languages: A case study with a quantum chemistry kernel. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, Miami, Florida, Apr. 2008. doi: `10.1109/IPDPS.2008.4536191`.

A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. McGraw-Hill, New York, 1989.