The 48-core SCC processor: the programmer's view

Timothy G. Mattson, Rob F. Van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, Saurabh Dighe

Abstract— **The number of cores integrated onto a single die is expected to climb steadily in the foreseeable future. This move to many-core chips is driven by a need to optimize performance per watt. How best to connect these cores and how to program the resulting many-core processor, however, is an open research question. Designs vary from GPUs to cache-coherent shared memory multiprocessors to pure distributed memory chips. The 48-core SCC processor reported in this paper is an intermediate case, sharing traits of message passing and shared memory architectures. The hardware has been described elsewhere. In this paper, we describe the programmer's view of this chip. In particular we describe RCCE: the native message passing model created for the SCC processor.**

Index Terms—**many-core processors, message passing APIs, non-cache-coherent shared memory.**

I. Introduction

The many-core transition is well underway. To optimize performance per watt, the number of cores per processor will inevitably increase over time. The key questions are how to connect the cores on a die, and how to program them.

An evolutionary approach based on familiar multiprocessor designs is attractive. A significant legacy of multi-threaded software for cache coherent shared address spaces exists, making such designs a natural choice. These designs depend on cache coherence protocols that keep the view of memory coherent across the cores. The protocol overhead per core grows with the number of cores, leading to a "coherency wall" beyond which the overhead exceeds the value of adding cores [1]. The impact of this wall can be delayed using clever caching schemes, but eventually the overhead associated with cache coherence will limit scalability for important workloads.

An alternative approach is to avoid cache coherence between cores altogether, with cores interacting by exchanging messages or through non-cache-coherent shared memory. Exploring these inherently scalable designs is a key aspect of the TeraScale Research program at Intel. The first processor in this program was the 80-core TeraScale processor [2]. The goal for the 80-core processor was to explore tiled architectures, 2D on-die meshes, and other issues pertaining to the circuits used on the chip. The 80 core chip featured a tiny, non-IA instruction set and had no compiler, no external memory, no I/O, and no operating system. Consequently, programming this chip was a difficult process that only a few programmers accomplished [3].

In this paper, we report on the second processor in the TeraScale Research program, the 48-core SCC processor [4]. Once again, this processor explores a scalable many-core architecture that does not use a cache-coherent shared address space. Unlike the 80-core processor, however, SCC uses a mainstream x86 instruction set. The Linux operating system is available, as well as C, C++, and Fortran compilers. An NFS file system can be mounted on the chip, allowing a full range of I/O operations. In other words, while the 80-core chip was a hardware experiment with software added later, the 48-core SCC processor is a true hardware/software co-design that can support a wide variety of software research projects.

We believe that it is important to document research processors during this historic period of transition to many-core computing. The SCC hardware was described elsewhere [4]. In this paper, we focus on the programmer's perspective. We begin with a brief overview of the SCC architecture and the software platform provided with the chip. Then we explore the memory subsystem provided by SCC, describe the message passing API (known as RCCE) designed for the SCC processor, and present benchmark results. We close with a look at the lessons we have learned and future plans for software research with the SCC processor.

II. SCC Hardware

The SCC chip is a many-core CPU consisting of 24 dual-IA-core tiles connected by a 2D-grid on-die network. The physical features of the chip are:

- 45 nm high K CMOS technology, 1.3 billion transistors.
- Tile area 18 mm2, die area 567 mm2.
- Power for the full chip ranges from 25 to 125 watts.
  - 25W at 0.7v, 125MHz core, 250MHz mesh and 50°C
  - 125W at 1.14V, 1GHz core, 2GHz mesh and 50°C
- Power for the on-die network
  - 6 W for a 1.5 Tb/s bisection bandwidth
  - 12 W for a 2 Tb/s bisection bandwidth

The features of the chip essential to programmers are summarized in figure 1. The tiles are organized in a 6 by 4 mesh with each tile containing:

- Two blocks, each with a P54C core, 16 KB instruction and data L1 caches plus a unified 256 KB L2 cache.

Manuscript received April 12, 2010.

Authors work at Intel Corporation. Corresponding author, T. G. Mattson, 2800 Center Drive, DuPont WA, 98327, phone: 253-228-4758, email: timothy.g.mattson@intel.com.

© 2010 IEEE Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

- A Mesh Interface Unit (MIU) with circuitry to allow the mesh and the interface to run at different frequencies.
- A 16 KB Message Passing Buffer.
- Two test-and-set registers.

The SCC processor's P54C cores are second generation Pentium® processors [5]. These are small in-order cores available as RTL that can be fully synthesized for layout onto the chip. In essence, the major research questions pertain to ways to connect large numbers of x86 cores and how this architecture interacts with and enables application software. To answer these questions the absolute performance of the cores is not important so we opted for a well known core that we could place on a tile with minimal effort.

Each tile connects to a router. This router works with the Mesh Interface Unit (MIU) to integrate the tiles into a mesh. The MIU packetizes data onto the mesh and de-packetizes data from the mesh using a round-robin scheme to arbitrate between the two cores on the tile. The MIU catches cache misses and decodes the 32-bit memory addresses from the core into a system address to access up to 64 GB of memory. This is managed through a lookup table (LUT) on each core to define how addresses are mapped onto system addresses. We discuss the LUT and the translation process in the next section.

To move data off chip, the SCC chip includes four DDR3 memory controllers. In addition, a router is connected to an off-package FPGA to translate the mesh protocol into the PCI express protocol, allowing the chip to interact with a PC serving as a management console.

The SCC chip includes instructions that let programmers control voltage and frequency. There are 8 voltage domains on a chip: one for the memory controllers, one for the mesh, and 6 to control voltage for the tiles (at the granularity of 4-tile blocks). Frequency is controllable at the granularity of an individual tile, with a separate setting for the mesh; providing a total of 25 distinct frequency domains. Additional details about SCC voltage and frequency control can be found in [4].

III. SCC Memory Architecture

The SCC processor is more than a distributed memory "cluster on a chip". Its memory architecture is composed of multiple distinct address spaces to support both distributed and shared memory programming models.

The most obvious architectural innovation with respect to memory is the message passing buffer (MPB) included on each tile. It provides a fast, on-die shared SRAM, as opposed to the bulk memory accessed through four DDR3 channels. While the processor does not offer any hardware-managed memory coherence, it features a new memory type to enable efficient communication between the cores. This new memory type is called the Message Passing Buffer Type (MPBT).

To understand this new memory type, consider how memory is accessed in the x86 architecture. An x86 processor uses a page table to define memory access semantics. Caching can be enabled or disabled at page granularity. If caching is enabled, the data moves with cache line granularity, i.e. as 32-byte packets, through the network, whereas for uncacheable accesses the original 1-, 2- or 4-byte requests are immediately forwarded to the destination memory. In the absence of a cache coherence protocol, uncacheable memory regions are the natural choice for shared access. However, since the P54C's Front Side Bus (FSB) interface supports only one outstanding request, this would impose significant performance penalties for communication between cores.

To alleviate this problem, the SCC processor provides MPBT memory commonly used with the tile's MPBs. A

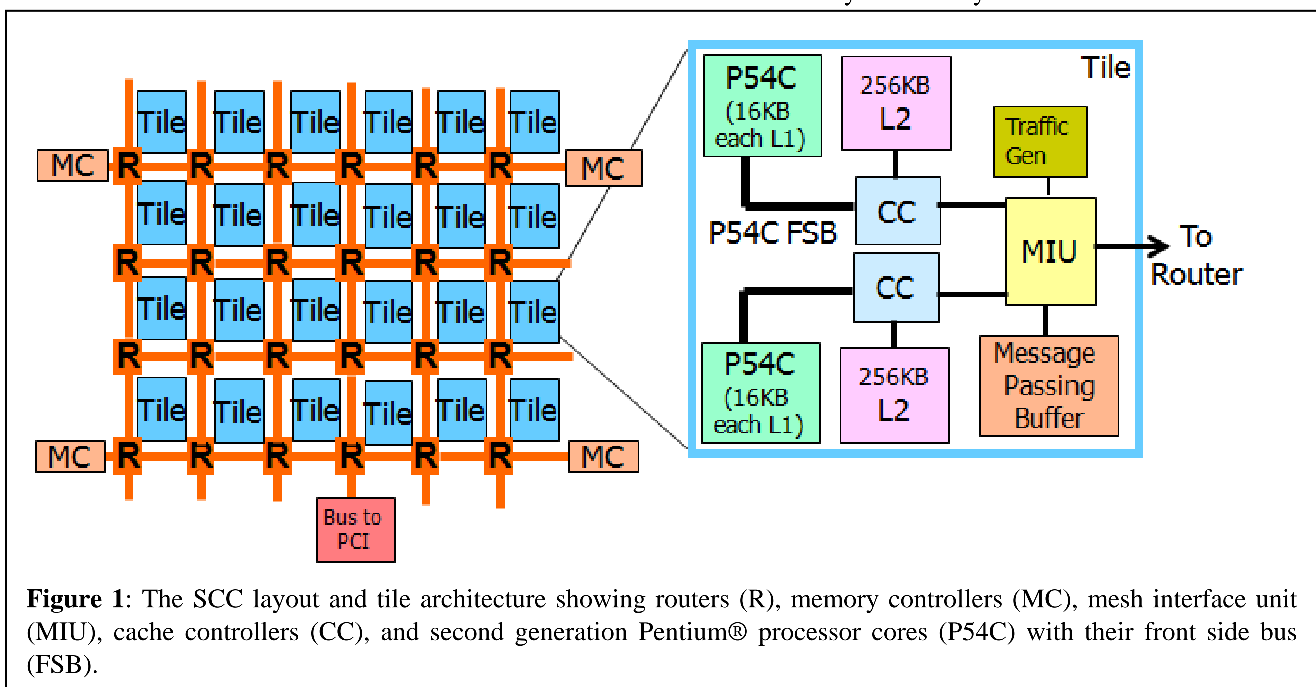

Figure 1: The SCC layout and tile architecture showing routers (R), memory controllers (MC), mesh interface unit (MIU), cache controllers (CC), and second generation Pentium® processor cores (P54C) with their front side bus (FSB).

reserved bit in the P54C's page table is used to mark MPBT data. Legacy software by default does not mark memory as MPBT and runs without modifications. If the bit is set, data is cached in L1 but it bypasses L2. To speed up writes, which do not get cached, a special write-combine buffer is placed downstream of the L1. It aggregates write requests from the core and forwards the data either when a whole cache line is written, or when content on another cache line is written. To support software-managed coherence a new instruction called CL1INVMB was added to the P54C. In one cycle it invalidates (not flush!) all cache lines tagged as MPBT in the L1 so any subsequent access will go to memory.

An important feature of the SCC is the way the system memory is mapped into a core's address space. In the early 90's when the P54C was designed, 32 address bits provided ample scope. Today, however, 4 GB of memory is insufficient for 48 cores. So on the SCC the 32-bit physical address space of the cores is divided into 256 chunks of 16 MB, each of which is mapped through a look-up table (LUT) to a 34-bit system address and the destination coordinate in the mesh (4 MCs @ 34b address → max. 64 GB DDR3 memory). Hence, the LUT configuration determines whether a physical address refers to off-chip DDR3 memory or on-die MPB memory.

A common system configuration, and the one used for the rest of this paper, views the address space in terms of three major regions (see figure 2):

- Private off-chip memory associated with each core: The LUTs are configured such that specific regions of the DDR3 memory are only accessible by a single core. This corresponds to the main memory of a conventional PC, i.e. the standard P54C memory model applies (private DRAM → L2 → L1 → CPU).
- Shared off-chip DRAM: Such shared memory regions are mapped by all LUTs and the system driver configures it as uncacheable regions by default to avoid consistency issues. This provides a direct link to the CPU, or more precisely, to the register file in an individual core.
- Shared on-chip SRAM (MPB): These addresses are mapped by all LUTs and marked at MPBT data in the page tables so the data can be cached in the L1 caches of the cores. .

Since coherence between cores is managed by the programmer, enforcing partial order of operations across cores is an important aspect of programming SCC. In some instances that requires mutually exclusive access to shared memory locations. For that purpose a test-and-set register (test&set) is provided for each core. Together with all other control and configuration registers, they are mapped into the physical address space through appropriate LUT entries. Since the test&set operation is atomic, the register supports lock semantics and can be used to implement atomic updates of more complex data structures.

The SCC processor does not provide a flush operation to help the programmer maintain consistency between views of data in the cache and in shared memory. This practically excludes conventional cacheable memory from data sharing as the L2 content cannot be controlled directly. When loading MPBT data it is necessary to invalidate the corresponding lines in L1 in order to prevent reading stale information. In case of writes it must be ensured that the data reaches its final destination before e.g. releasing a lock. One way to accomplish this is, again, through MPBT invalidation (i.e. the CL1INVMB instruction) before updating MPB content (this guarantees a write miss which will propagate downstream) and by always writing entire cache lines in order to flush the write-combine buffer.

IV. SCC Platform

A major research thrust for the SCC chip is to explore different ideas for how to construct scalable platforms for many core chips. At this time we have developed two platforms based on the SCC chip.

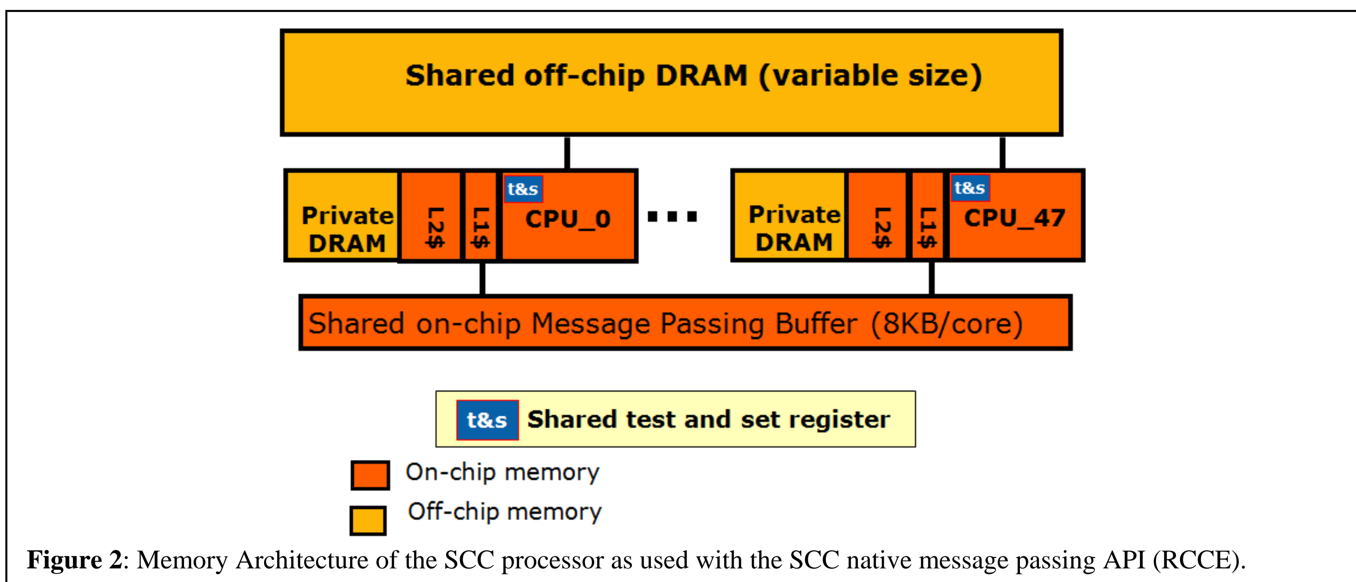

Figure 2: Memory Architecture of the SCC processor as used with the SCC native message passing API (RCCE).

The most commonly used platform is based on running a Linux kernel on each core. We used the Linux Kernel 2.6.16 with Busybox 1.15.1. A TCP/IP driver was implemented to support safe communications between cores on a die (not used by our native message passing layer for efficiency reasons, see section V) and to provide a connection to a management console. This allowed us to export an NFS file system visible among all the cores. In addition, drivers for low-level access to the MPB and other hardware features of the SCC were included. Since each core runs a familiar Linux OS and x86 instruction set, we were able to use the Intel suite of compilers, and Intel's sequential Math Kernel Library (MKL) for mathematical functions.

In addition to the Linux platform, we adapted a C-based programming framework that lets code run directly on the SCC processor without an OS. We called this the "BareMetalC" environment. It provides direct access to all hardware features of the SCC processor and is useful to establish benchmarking baselines. The software feature set available with this environment, however, is limited (for example, only rudimentary memory mapped I/O is supported), making it difficult to use for significant application development.

The BareMetalC and Linux platforms both use a Management Console PC to bring up and control the SCC platform. It is written in C++ and uses the Nokia Qt cross-platform application and UI framework. This console interacts with the SCC processor through the PCIe interface with drivers that provide:

- TCP/IP connection to SCC
- Connection to Management Console PC applications.
- Access to all memory and register locations of SCC.

Functionality is exposed through a C++ programming API (sccApi), through command-line tools (e.g. sccReset to reset the cores) or through a graphical user interface (sccGui). These utilities are provided with an SCC platform and will not be described further in this paper.

V. SCC Communication Environment (RCCE)

The SCC architecture supports a variety of parallel programming models. At its foundation, however, the SCC processor is a message passing chip. The cores may interact through shared memory, but with the total lack of cache coherence between cores, the most natural and efficient programming models for this chip build on the ability to send messages between cores. The message passing library developed for this chip and used to analyze workloads as the chip was designed is called RCCE.

RCCE is based on one-sided "put and get" primitives similar to those in the well known SHMEM library [6]. These primitives move data from the private memory through the L1 cache of the sending core to the message passing buffer (MPB) and then to the L1 cache of the receiving core. The MPB allows L1 cache lines to move between cores without having to use the off-chip memory. We note that RCCE was designed to support Linux as well as BareMetalC, which means it must be able to function without any operating system. This places a number of restrictions on the library. For example, no threads are present in BareMetalC, so asynchronous message passing is not possible.

RCCE uses a static Single Program Multiple Data (SPMD) model familiar to message passing programmers. When an application is launched (using a program we provide called "rccerun"), the user specifies the number of cores to use from a given subset of cores on the chip. Identical executables are launched on all cores, where they are run by a "unit of execution" or UE. The UE is an agent that "owns the program

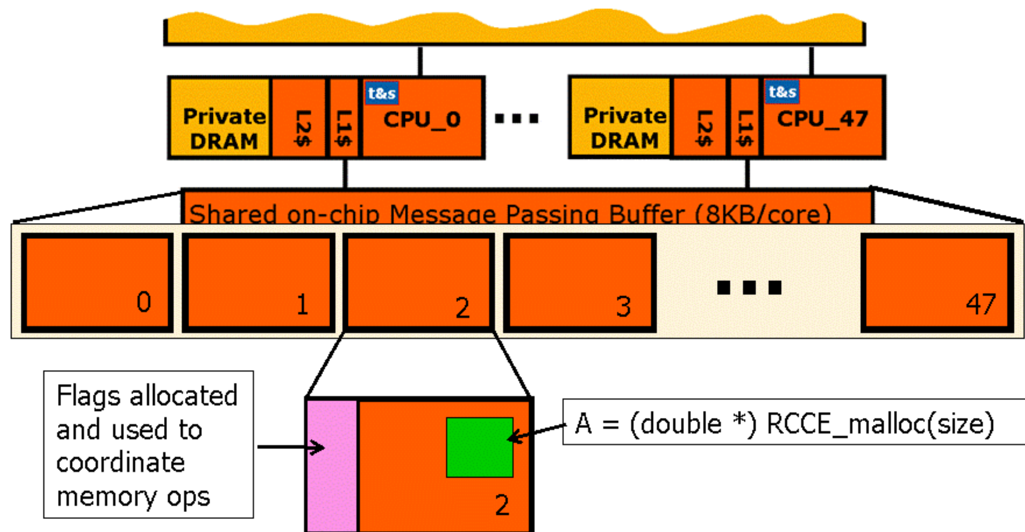

Figure 3: Symmetric name space model for the MPB when used with RCCE, showing the MPB segments for cores 0 to 47, and an expanded view of the MPB segment for core 2. While we show storage for flags in a contiguous block at the beginning of the segment, flags may be spread throughout the segment depending on RCCE's usage mode.

counter" and makes progress in a computation; i.e. it is an abstraction that can be implemented as a thread or process. Once assigned to a core, a UE remains pinned to that core. Each UE is assigned a rank, which is a sequence number ranging from 0 to N-1 (N is the number of participating cores). Since a UE is pinned to a core, the rank uniquely defines a core and a UE. The beginning of most RCCE programs includes calls to the functions:

```
RCCE_init(&argc, &argv);
int num_cores = RCCE_num_ues();
int ID = RCCE_ue();
```

These functions initialize the RCCE library, return the total number of cores, and define a core's ID as its rank.

In the RCCE execution model, programs start on the cores in an unspecified order. Any program that depends on a particular order is a non-conformant RCCE program. As a further simplification, we assume that only one RCCE program is running on the SCC at one time. Finally, RCCE assumes that the contents of the test&set registers and the MPB are "zeroed" when a program launches. Utilities that help a user ensure that the system is in a clean state are provided with the SCC platform.

The key to understanding the implementation of RCCE is the MPB. While the MPB is physically distributed about the tiles of the SCC processor, it is logically a single shared address space; any core can write to any address in the MPB, using a simple memcpy. Managing the consistency of this address space between cores for general data structures would be difficult, so we adapted the MPB to a specialized structure to support message passing.

We call our approach a "symmetric name space" model (see figure 3). In this approach, the MPB is logically divided into 8 KB contiguous blocks, one per core, which we create by dividing the 16 KB MPB on each tile into two equal segments. A portion of each segment is used for flags that coordinate communication between cores. The rest is available to use as buffers for passing messages between cores. We create these buffers through a collective function call to the RCCE MPB memory allocator, for example;

```
char *A = (char *)RCCE_malloc(size);
```

By signifying this as a collective function call, we require that every core participating in the computation call this function and do so in the same order with respect to other RCCE functions. RCCE_malloc() creates a named region "A" on each core defined by its offset from the beginning of each core's segment in the MPB. This allows a core to later "put" or "get" a packet (buffer) in private memory into or from the MPB with reference to the ID of the target core and the name of the region in the calling core's MPB (A):

```
RCCE_put(A, buffer, size, ID);
RCCE_get(buffer, A, size, ID);
```

The symmetric name space allows us to manage the complexity of the MPB address space inside our RCCE functions and saves the programmer from explicitly computing addresses in other cores' segments of the MPB.

RCCE is a low level API that exposes the full capabilities of the SCC processor. Hence, details that are typically hidden in mainstream message passing environments such as MPI are fully exposed in RCCE. For example, the named region "A" must be cache aligned, occupy full L1 cache lines (i.e. multiples of 32 bytes) and fit within the available space in the MPB. Since the SCC processor does not provide cache coherence between cores, the programmer must enforce protocols for safe movement of data. This is done with

```
#include "RCCE.h"
 int RCCE_APP() {

  RCCE_init(&argc, &argv);
  NUES = RCCE_num_ues();
  ID = RCCE_ue();

  ID_right = (ID+1)%NUES;
  ID_left = (ID-1+NUES)%NUES;
  size = BUFSIZE*sizeof(double);
  buffer  = (double *) malloc(size);
  cbuffer = (double *) RCCE_malloc(size);

 /* create and initialize flag variables */
   RCCE_flag_alloc(&flag_sent);
   RCCE_flag_alloc(&flag_ack);
   RCCE_flag_write(&flag_sent,
          RCCE_FLAG_UNSET, ID))
   RCCE_flag_write(&flag_ack,
          RCCE_FLAG_SET, ID_left))

for (int round=0; round<nrounds; round++) {

   RCCE_wait_until(flag_ack, RCCE_FLAG_SET);
   RCCE_flag_write(&flag_ack,
                      RCCE_FLAG_UNSET, ID);
   RCCE_put(cbuffer, buffer, size, ID_right);
   RCCE_flag_write(&flag_sent,
                     RCCE_FLAG_SET, ID_left);

   RCCE_wait_until(flag_sent,
                     RCCE_FLAG_SET);
   RCCE_flag_write(&flag_sent,
                    RCCE_FLAG_UNSET, ID);
   RCCE_get(buffer, cbuffer, size, ID);
   RCCE_flag_write(&flag_ack,
              RCCE_FLAG_SET, ID_left);
}
```

Figure 4: A simple RCCE program to shift messages around a logical ring of UEs (with one UE per core on current experiments with SCC). This version of the program uses the low level one-sided communication layer.

explicit synchronization flags to ensure that bulk reads and writes complete in the right order. To support this functionality, we added functions to allocate, set and wait on Boolean flags in the MPB. First, the programmer needs to define a variable of type RCCE_FLAG (defined in RCCE.h) and allocate space for the flag.

```
RCCE_FLAG fl;
RCCE_flag_alloc(&fl);
```

The flag is set or unset with a RCCE flag_write function. A UE can wait on a value of a flag using a RCCE wait_until function:

```
RCCE_flag_write(&fl, RCCE_FLAG_SET, ID);
RCCE_wait_until(fl, RCCE_FLAG_UNSET);
```

To work with the flags, we have defined values RCCE_FLAG_SET and RCCE_FLAG_UNSET in RCCE.h. To understand the operation of the flags and how they interact with the one sided communication functions, consider the program listing in figure 4. This program treats the collection of UEs as a ring and passes messages clockwise around the ring (each UE gets a packet from the "left" and passes it to the "right" during each round). Flags are allocated and used to enforce a safe order to put values into the MPB and to later pull them from the MPB. Note that this code assumes that the message fits in the MPB, that the buffer in the MPB is cache aligned, and that reads and writes occur in full cache lines (BUFSIZE is divisible by 4).

RCCE allows the programmer to specify whether each flag occupies a whole cache line, or only a single bit within an MPB cache line. The latter results in a storage compression factor of 256 compared to whole-cache-line flags, which means that more space within the MPB can be allocated to communication payload (potentially higher bandwidth). However, it also requires the guarantee of atomic write access to the flags, which on the SCC can only be accomplished by guarding writes with a lock. Conveniently, the test&set register can be used for that purpose, but the locking operation does result in higher latencies.

The one-sided API exposes all the details of managing the movement of cache lines through the SCC processor. Our goal was to create the smallest practical message passing library suitable for the SCC processor. We found, however, that as we ported applications to RCCE, we often needed two-sided synchronous message passing of the type used in the ring shift example. Hence, we added a pair of two-sided communication functions to RCCE that build on the elementary RCCE_put/get and flag manipulation routines (flag and MPB management is done by the library and may be hidden from the programmer), and relax restrictions on the message size:

```
RCCE_send(sendbuffer, size, target_ID);
RCCE_recv(recvbuffer, size source_ID);
```

The behavior of these functions will be familiar to message passing programmers. The functions block until matching calls on both sides complete execution (synchronous communication). The private buffer on the sending side is broken into packets that are passed through the MPB to a private buffer on the receiving side. Note that in MPI, the functions MPI_Send/Recv imply synchronization even for a zero sized message. This is not the case in RCCE, whose messages are headerless. A zero sized buffer is analogous to a no-op and returns immediately without implying any synchronization.

The ring shift example using these two new routines is shown in figure 5. Because all aspects of flag management are now hidden from the programmer, we cannot judiciously seed the flags with values that allow the code to make progress without changing the structure of the loop body. By splitting the body of the loop into two phases and adding a memory copy operation, we can avoid deadlock and guarantee correctness. This has proved a recurring theme when porting codes from MPI to RCCE. In MPI one can use asynchronous communication to avoid deadlock—at the cost of maintaining and traversing message queues. In the strictly synchronous RCCE environment, deadlock-free communication patterns must be chosen upfront, requiring additional programmer effort, but paying off through improved performance,

```
#include <string.h>
#include "RCCE.h"
int RCCE_APP() {

  RCCE_init(&argc, &argv);
  NUES = RCCE_num_ues();

  ID = RCCE_ue();

  ID_right  = (ID+1)%NUES;
  ID_left  = (ID-1+NUES)%NUES;
  int size  = BUFSIZE*sizeof(double);
  buffer   = (double *) malloc (size);
  buffer2 = (double *) malloc (size);
```

```
for (int round=0; round<nrounds; round++) {

 for (int c = 0; c<2; c++) {
    if ((ID+c)%2)
         RCCE_send(buffer, size, ID_right);
    else
         RCCE_recv(buffer2, size, ID_left);
   }
   memcpy(buffer, buffer2, size);
}
```

Figure 5: A simple RCCE program to shift messages around a ring of UEs. Most declarations are omitted to save space. This version of the program uses the basic two-sided communication layer.

particularly through lower latency.

The addition of the two-sided synchronous mode to message passing is typical of our approach. As we find the need for new functionality, we add it to RCCE, but retain access to the lower level functions. The goal is to keep RCCE small, but to include essential functionality to support programmer productivity. In addition to the basic message passing, RCCE currently includes:

- A power management API to modify voltage and frequency within sectors of the SCC (Section VI)
- Communicators to define partitionings of UEs within collective communications (reduction, broadcast, barrier)
- Memory (de-)allocation for the MPB and the off-chip shared DRAM

As the user base grows around RCCE, we anticipate adding a small number of functions to the library. The overall size of the API, however, will be limited. RCCE is small enough so it is easy to port and easy to maintain by a small team. This is an important advantage of RCCE. If more fully functioning message passing is required, a programmer can always use MPI (which has been ported to SCC under Linux through an interface to the TCP/IP stack running on each core). In addition, we have shown it is possible to build abstraction layers on top of RCCE—such as the well-known Actors model [15]—without any modification to the library.

VI. Software Controlled Power Management

Limiting the power consumed by processors is one of the most important tasks facing designers and system builders today. RCCE adopts the philosophy that whenever a workload can use the cores at a reduced power but without affecting performance, it should be possible to throttle energy consumption under control of the library. For example, cores may idle while waiting for work, or they may enter a phase dominated by I/O or memory accesses, in which case the local clock can be slowed without changing total wall clock time of an application.

The SCC processor offers elementary facilities for varying voltage and frequency. Specifically, the voltage within each block of 2x2 tiles (8 cores) can be specified by writing an appropriate control value into a single designated register for the entire chip. Similarly, the frequency within any tile can be specified by writing an appropriate control value into a designated register within that tile. Because voltage and frequency are not independent, care must be exercised that the cores not leave the safe electrical operation zone. For that purpose, RCCE contains a pair of power management functions that initiate and finalize a discrete step (up or down) in power usage within the voltage domain containing the calling UE by changing voltage and frequency (Fdiv specifies a target integral CPU clock frequency divider) in tandem.

```
RCCE_iset_power(Fdiv, &RCCE_request);
RCCE_wait_power(&RCCE_request);
```

The reason for the split in two functions is that SCC voltage changes currently incur a significant latency (on the order of one millisecond, or half a million core cycles, on a quiescent processor with baseline frequency setting). Without the split, the affected cores would need to block during a power transition, thus wasting cycles. However, if RCCE_wait_power is issued sufficiently long after the corresponding RCCE_iset_power, no wait is incurred at all. Often, it is only the application programmer who can determine when these power change instructions can be inserted to advantage.

Use of these power management functions is complicated by the granularity of the voltage control domains. Power changes apply to all 8 cores within a domain, so programmers must ensure that all participating cores in the domain benefit from a power change. This may require careful synchronization among those cores. Moreover, only one voltage change request can be serviced by the processor at any one time, creating the possibility of induced core stalls when multiple requests are issues (nearly) simultaneously.

When the application can not accommodate the long latencies associated with voltage changes, programmers can reduce power by reducing the clock frequency within a whole voltage control domain. This is done with a call to:

```
RCCE_set_frequency_divider(Fdiv);
```

where the integer parameter, `Fdiv`, stipulates an integral divisor to reduce the frequency from the baseline reference clock for the chip. This incurs a latency of only about 20 core cycles, but is less effective than combined frequency/voltage scaling.

VII. Results

With two platforms (BareMetal and Linux) and a myriad of power settings to consider, a full exploration of SCC performance would go well beyond the scope of this paper. Instead, we provide a preliminary snapshot of SCC performance numbers. In particular, we report network latency and bandwidth numbers with RCCE, results from our port of two NAS parallel benchmarks (BT and LU) [7], and some preliminary results from our power management experiments. For all measurements the cores were running at 533 MHz and the mesh at 800 MHz.

To evaluate network latency we used “ping-pong,” a program that uses RCCE_send/recv to bounce messages between pairs of cores. We fixed one core at the corner of the chip and then changed the second core in the pair to measure the round-trip latency of a 32 byte message as a function of the number of hops across the network. The observed data was an excellent fit to a straight line with slope of 30 nanoseconds and a y-intercept of 5 microseconds. The y-intercept of 5 microseconds represents the cost incurred within the cores at either side of the communication and compares favorably with the measured round trip latency between the cores on a single tile (5.2 microseconds). The slope corresponds to the incremental cost of moving messages across the network. Since a 32 byte message fits within a single cache line, the message and the two flags used inside RCCE_send/recv to manage the messages result in three

network transactions per network hop. As shown in [4], the cost of a hop across a router is 4 cycles. Hence, for a network running at 800 Mhz, we expect a roundtrip latency of 2*3*4/0.8 GHz or 30 nanosecond per hop; which is precisely the value we observed.

We also used the ping-pong program to measure bandwidth between a pair of cores with a network distance of 8 hops (see figure 6). We tested the following configurations:

A. MPB of nominal size (8KB/core) and 32-byte flags.
B. MPB of nominal size and 1-bit flags.
C. MPB of 50% of nominal size and 32-byte flags.
D. MPB of 25% of nominal size and 32-byte flags.

As expected, the cost is largely due to operations executed by the cores, not to time spent on the network. We see four performance domains for case A in figure 6, with some variations for the other cases. The bandwidth climbs steadily up to approximately half the size of the L1 cache (L1C). The 16 KB L1C saturates around this point, since on the receiving core, the L1C is consumed by both the message read directly from the MPB and by the destination buffer in the core's private memory. At messages of 8 KB, and beyond the L1C experiences increasingly frequent evictions, due to conflicts between MPB reads and message writes, as well as between the writes themselves. This results in a steep drop in bandwidth. until at 24 KB no useful private memory message data remains in L1 at the end of the message transfer, and bandwidth levels off to that sustainable by the L2 cache. As messages continue to grow, they exceed a size that can be accommodated by the L2 cache and performance drops again, to a level determined by bandwidth to the off-chip DRAM.

Comparing cases A and B, we observe that 1-bit flags are always slower than 32-byte flags. The 1-bit flags incur an overhead because updates to these flags must be protected with locks (implemented with test&set registers). The 32-byte flags, however, consume more space in the MPB, leaving less space for payload, and thus requiring more synchronizations. For example, if all 48 cores participate in the computation, the 32-byte flags consume about 30% of the MPB, as opposed to less than 1% for the 1-bit flags. Our observations show that the benefits of more space in the MPB for messages are clearly overshadowed by the extra overhead of manipulating locks..

To further explore effects due to the size of the MPB, we reduced the effective size of the MPB by half (case C) and by 75% (case D). As seen in figure 6, case C has virtually the same performance as case A, except at a message value just larger than 50% of the MPB. Reducing the effective MPB even further (case D) has an unexpected effect. While some bandwidth degradation can be observed for message sizes greater than 25% of the MPB, we actually see bandwidth improvement for messages greater than the MPB. The smaller MPB for case D reduces the number of cache lines needed to hold a block of data read from the MPB; thereby leaving more space in the receiving core's L1 cache to hold the destination buffer for the message. The dips in the bandwidth curves for cases C and D at messages of size 4096 and 2048,

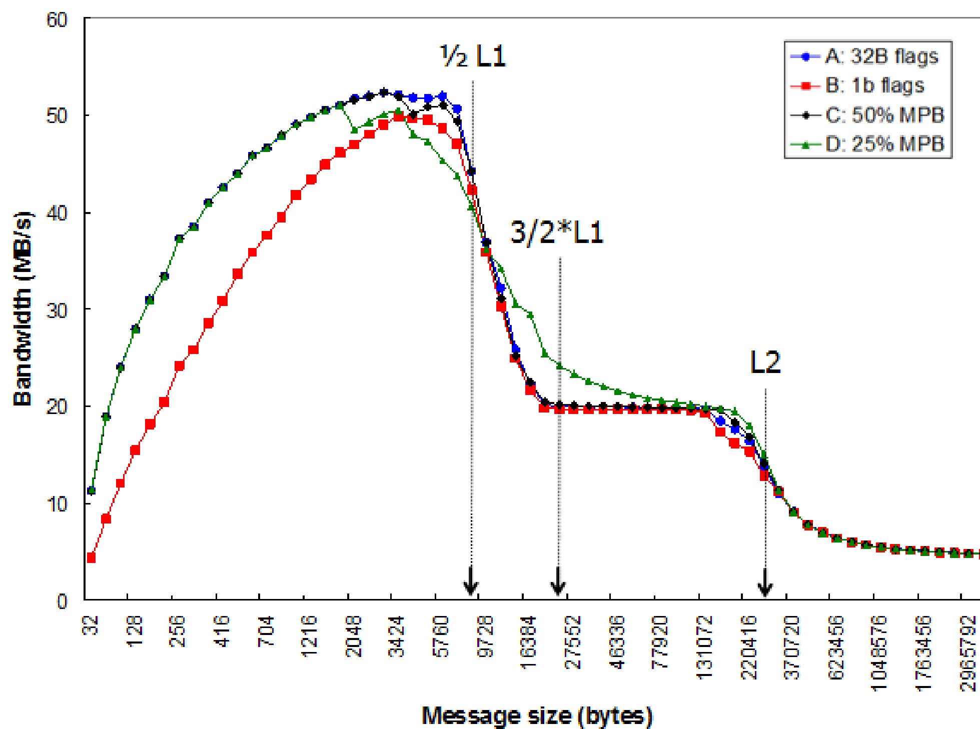

Figure 6: Bandwidth between core 0 and 46. Cases A, C and D use 32-byte flags. Cases A and B use the full size MPB. We provide three reference points (1/2 L1 cache (8 KB), 3/2 L1 cache (24 KB), and L2 cache (256 KB)) to define four performance regions for different message sizes (MSG): (i) MSG < 8 KB, no L1 conflicts, (ii) 8KB<MSG<24Kb, increasing L1 conflicts, (iii) 24KB<MSG<256 KB, saturated L1, performance dictated by L2 bandwidth, and (iv) 256 KB<MSG, saturated L2, performance leveling off to speed of DRAM.

respectively, are due to the fact that these incur a second synchronization as the message just spills out of the MPB payload area.

For a more challenging problem, we ported the LU and BT benchmarks from the well-known NAS parallel benchmarks [7] to RCCE. These have distinctly different communication patterns. LU uses a “pencil decomposition” to assign a column block of a 3D discretization grid to each core. A 2D pipeline algorithm is used to propagate a wavefront communication pattern across the cores. BT decomposes the problem into larger numbers of blocks that are assigned to the cores using a cyclic distribution. The communication patterns are regular and employ geometrically nearest neighbor exchanges as the algorithm sweeps over successive planes of blocks. We ran the Class A, B and W benchmarks. We found the best performance for the Class B benchmarks on a 102x102x102 discretization grid; results of which are shown in figure 7. The speedup is good across the range of problems studied. This is not surprising. The LU and BT benchmarks are highly scalable. Since the SCC network is much faster than the cores, network overhead is insignificant relative to the performance of the cores and we expect such scalable benchmarks to scale well on SCC.

To test RCCE’s power management API we implemented a simple synthetic application based on a task queue. Each task was executed by a team of cores comprising an entire power domain. The tasks were further decomposed into subtasks, which were internally implicitly synchronized through fine-grain communications. This produced the desired effect of keeping all cores in a power domain in the same computational intensity state during the life of each task. No synchronization was implied between the top-level tasks. Each task featured a memory intensive initialization phase, providing a good opportunity for power reduction, as well as a CPU intensive phase requiring maximum voltage and frequency. By matching the voltage and frequency to the needs of the phases, the resulting application experienced instantaneous power reductions of up to 74%. We did not determine aggregate power savings which will depend strongly on the ratio of the time spent in computationally (and power) intensive phases over other phases in an application. Moreover, for time sensitive applications the programmer should take care to not increase the execution time by inadvertently causing slowdown of computationally intensive phases or by introducing stalls due to inappropriately scheduled power transitions. Power management on SCC is still an active area of research.

VIII. Related work

Message passing in the form of MPI [8] is the legacy API of parallel computing. MPI is a full-featured API that has proven effective for a remarkably diverse range of parallel workloads on both shared memory and distributed memory platforms. Indeed, MPICH [9] has been ported to the SCC processor and ran without modification on top of the TCP/IP drivers on the Linux SCC platform.

MPI, however, is large and mapping it directly onto the low-level features of a platform (as opposed to using a portable networking protocol such as TCP/IP) can be prohibitively difficult. Furthermore, MPI was designed for general applications programming for large distributed multi-computers such as clusters or MPP systems. Our interest with RCCE is to consider a message passing API designed specifically to the needs of a many-core processor. RCCE started small and only grows new functionality as the need arises. It will be interesting to see how much of MPI’s functionality needs to be added to RCCE as our user base grows.

The Multicore Association developed MCAPI [10], a message passing API designed specifically for many-core processors. MCAPI has two-sided message passing similar to that found in MPI, but it also includes channel based

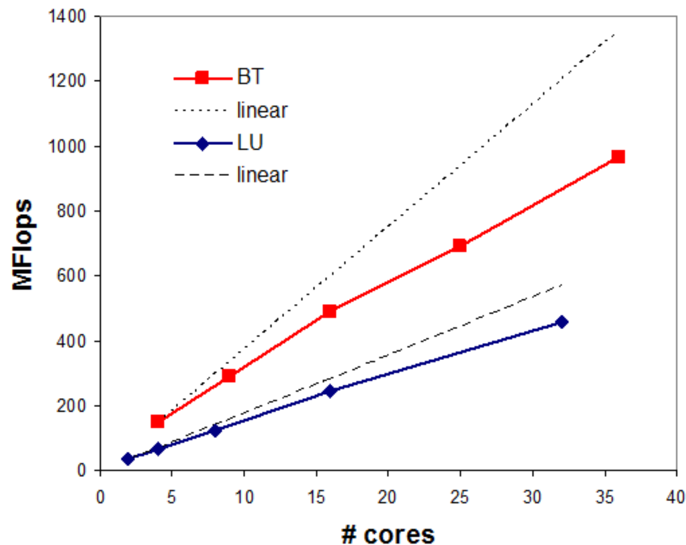

Figure 7: MFLOPS vs. number of cores (UEs) for the NAS Parallel Benchmarks LU and BT [7] with the class B problems on a 102 x 102 x 102 grid. The benchmarks used RCCE and ran on the SCC processor with 533 MHz cores, 800 MHz routers and 800 MHz DDR3 memory controllers.

protocols. These let a programmer set up a communication pattern in terms of fixed channels, which can then be used repeatedly for low latency message passing. MCAPI is around one third the size of MPI, but compared to RCCE it is still a large API. More significantly, MCAPI is a portable API that should be able to map onto almost any many-core processor. This contrasts with RCCE, which was created as part of a hardware/software co-design process for the SCC processor.

We believe a streaming protocol built on top of RCCE's put/get API would let us support MCAPI. For applications with regular communication patterns, MCAPI's channel based approach could reduce latencies. We plan to experiment with this approach and determine if the latency savings are significant on the SCC processor.

An important project closely related to RCCE is SHMEM [6]. We have already described how the one sided API used in RCCE was modeled closely after SHMEM. A more significant similarity between the two projects, however, is that both SHMEM and RCCE were designed with optimization to a specific hardware platform in mind; the SCC processor in the case of RCCE and the Cray MPP systems (e.g. T3D and T3E) in the case of SHMEM. This results in lower latencies, but we note that for both SHMEM and RCCE, the low-latency one-sided communications require complicated synchronization mechanisms that can be challenging to manage in application programs. Consequently, we added a two-sided communication protocol to RCCE.

Other projects whose one-sided communications resemble those in RCCE are ARMCI [13] and GASNet [14]. Both of these communication libraries, as well as SHMEM, rely on the capability to move data between different address spaces via a native network-specific Direct Memory Access (DMA) mechanism, or on the availability of globally shared memory. Neither of these facilities exists on SCC. The processor does not provide the functionality of proprietary Network Interface Cards required for DMA, and the on-chip shared memory that RCCE uses to speed up data transfer is so small that it can only be used as a staging buffer, not as the final destination for data. Moreover, the absence of cache coherence of the MPB makes it unsuitable for direct access by the application programmer. Thus, while the syntax of RCCE's one-sided communications is very similar to that of SHMEM, ARMCI, and GASNet, there is an important semantic difference in that a single one-sided RCCE copy operation cannot complete a general communication between cores.

IX. Future Work and Conclusions

Our software work with the SCC processor has just begun. Unlike the 80-core chip, which was accessible to only a handful of researchers inside Intel Corporation, the SCC processor is being made available to research collaborators in both industry and academia [11]. Consequently, the scope of software research for the SCC processor will be extensive. Projects either planned or already underway include:

- Porting numerical linear algebra software from the FLAME group at UT Austin [12].
- Improving bandwidth of large message transfers by pipelining.
- Constructing deadlock-free efficient collective communications using only blocking calls for important communication patterns (all-to-all, permutation, etc.)
- Exploring different ways to utilize the power management features of SCC using system software utilities or explicitly inside application software.
- Detailed benchmarking studies to establish a performance baseline and evaluate overheads from different system software configurations and synchronization flag implementation strategies.
- Evaluating the effectiveness of different many-core operating systems on the SCC processor

Our preliminary work has demonstrated that the SCC processor and its native message passing API provide an effective software development platform. While RCCE is small, it has been sufficient to program both the benchmarks reported here and a range of different applications not discussed in this short paper. The expected difficulties due to the lack of asynchronous message passing have so far not materialized. We have been able to construct ad hoc synchronous communication patterns required by RCCE for all ported applications, and have started to develop theory on how to make our approach more general.

We validated that the on-chip shared memory buffer—added to the design by request of the SCC software team—though small, is large enough to support efficient inter-core communications. The symmetric memory model RCCE employs for the message passing buffer, which at first glance may appear rather restrictive, proved to be not a hindrance at all. And because it enables on-chip shared memory access without coordination with other cores, it is very efficient, in addition to being easy to program.

Using SCC power management capabilities in their current form for application-steered power optimization is cumbersome at best. We expect great improvements in utility and programmability if the following expected trends materialize:

- Shrinking the size of voltage and frequency domains to a single core.
- Reducing the latency of voltage change commands to a few (less than 100) cycles.

Finally, we found that although it would have been desirable to allow atomic operations on data in the message passing buffer, the availability of a single test&set register per core provided us with sufficient capability to implement locks needed for mutual exclusion. In RCCE these are currently used for a single purpose, namely the write protection of packed synchronization flags, which all fit within a single cache line. Hence, there is no need for more such registers. It remains to be determined whether more registers would improve performance as more programming models are ported directly to the SCC or layered on top of RCCE.

REFERENCES

[1] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob van der Wijngaart, "The case for Message Passing on Many-core Chips", University of Illinois Champaign-Urbana Technical Report, UILU-ENG-10-2203 (CRHC 10-01), 2010.

[2] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Member, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Member, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar, "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," IEEE Journal of Solid-State Circuits, Vol. 43, No. 1, Jan 2008.

[3] Tim Mattson, Rob van der Wijngaart, Michael Frumkin, "Programming Intel's 80 core terascale processor," Proceedings of the 2008 ACM/IEEE conference on Supercomputing , SC08, Austin Texas, Nov. 2008

[4] 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. De1, R. Van Der Wijngaart, T. Mattson, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS", Proceedings of the International Solid-State Circuits Conference, Feb 2010

[5] D. Anderson, T. Shanley, Pentium *Processor system architecture*, Addison Wesley, 1995.

[6] R. Bariuso, Allan Knies, "SHMEM'S User's Guide," Cray Research, Inc., SN-25 16, rev. 2.2, 1994.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S., Schreiber, H. D. Simon, V. Venkatakrishnan and S. K. Weeratunga, "THE NAS PARALLEL BENCHMARKS," Intl. Journal of Supercomputer Applications, vol. 5, no. 3 (Fall 1991), pg. 66-73

[8] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, "MPI: The Complete Reference," MIT Press, 1996

[9] http://www.mcs.anl.gov/research/projects/mpich2/

[10] http://www.multicore-association.org

[11] http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[12] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. "Introducing: The libflame Library for Dense Matrix Computations." IEEE Computing in Science & Engineering.11 (6):56--62, 2009.

[13] J. Nieplocha, V. Tipparaju, M. Krishnan, D. Panda. High Performance Remote Memory Access Comunications: "The ARMCI Approach. International Journal of High Performance Computing and Applications, Vol 20(2), 233-253p, 2006"

[14] D. Bonachea GASNet Specification, v1.1, U.C. Berkeley Tech Report (UCB/CSD-02-1207), 2002.

[15] Gul A. Agha[a, *] and Wooyoung Kim, "Actors: A Unifying Model for Parallel and Distributed Computing," J. Systems Architecture, Vol. 45, no. 15, September 1999, pp. 1263-1277