Programming Many-Core Architectures - A Case Study: Dense Matrix Computations on the Intel SCC Processor

FLAME Working Note #55

Bryan Marker[*]
Ernie Chan[†]
Jack Poulson[‡]
Robert van de Geijn[§]
The University of Texas at Austin

Rob F. Van der Wijngaart[¶]
Timothy G. Mattson[‖]
Theodore E. Kubaska[**]
Intel Corp.

January 24, 2011

Abstract

A message passing, distributed-memory parallel computer on a chip is one possible design for future, many-core architectures. We discuss initial experiences with the Intel Single-chip Cloud Computer research processor, which is a prototype architecture that incorporates 48 cores on a single die that can communicate via a small, shared, on-die buffer. The experiment is to port a state-of-the-art, distributed-memory, dense matrix library, Elemental, to this architecture and gain insight from the experience. We show that programmability addressed by this library, especially the proper abstraction for collective communication, greatly aids the porting effort. This enables us to support a wide range of functionality with limited changes to the library code.

1 Introduction

The computer industry is at a crossroads. The number of transistors on a chip continues to climb with successive generations of process technology (Moore's law) while the power available to a socket is decreasing. This has led to a "power wall" and has shifted the focus of computer architecture from raw performance to performance per watt.

A well-known response to the power wall problem is to replace complex cores running at high frequencies with multiple simple but low power cores within a chip [7]. The major microprocessor vendors currently offer CPUs with modest numbers of cores (two to eight) organized around a cache-coherent shared address space. These multicore processors in many ways appear to the programmer as a familiar multiprocessor, multi-socket system integrated onto a single chip. Cache-coherent shared memory is convenient for the programmer since the hardware creates the illusion

[*] Department of Computer Science, The University of Texas at Austin, Austin, Texas 78712, bamarker@cs.utexas.edu.

[†] Department of Computer Science, The University of Texas at Austin, Austin, Texas 78712, echan@cs.utexas.edu.

[‡] Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, Texas 78712, poulson@ices.utexas.edu.

[§] Department of Computer Science, The University of Texas at Austin, Austin, Texas 78712, rvdg@cs.utexas.edu.

[¶] Intel Corporation, Santa Clara, California 95054, rob.f.van.der.wijngaart@intel.com.

[‖] Intel Corporation, DuPont, Washington 98327, timothy.g.mattson@intel.com.

[**] Intel Corporation, Hillsboro, Oregon 97124, theodore.e.kubaska@intel.com.

of a single, coherent address space that spans multiple cores and maintains consistency on behalf of the programmer. But this abstraction adds overhead that grows with the number of cores and hence may not be scalable to support large numbers of cores. An alternative approach is to model chips with multiple cores after clusters, which are parallel architectures with scalable disjoint memories that lack cache coherence. An instance of this approach is the Intel Single-chip Cloud Computer (SCC).

The SCC processor [17, 20] is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research. The chip presents to the programmer a collection of cores with private memories, connected by an on-die network. These can be programmed as a "cluster on a chip" with messages moving around the network to coordinate execution of processes running on the cores and communicate data between those processes. In addition to this logically distributed memory, the SCC processor has two shared address spaces: one on-die and one off-chip. Neither of these address spaces provides any level of cache coherence between cores, which makes the chip highly scalable but leaves the burden of maintaining a consistent view of these address spaces to the programmer.

In this paper, we describe the results of an effort to port a major software library, the Elemental library [21] for dense matrix computations on distributed-memory computer architectures, to this platform. To do so, we start with a minimal programming environment, RCCE [20, 28], that consists of synchronous point-to-point communication primitives. This communication layer allows all issues related to coherency to be hidden in the passing of messages, at the expense of placing the entire burden of parallelization on the library programmer. We show that by adding a few commonly used collective communications to this layer, the entire Elemental library, which has functionality similar to ScaLAPACK [8] and PLAPACK [26], is successfully ported with relatively little effort. The conclusions we draw from this experience are:

- Message passing can be an effective way to avoid having to provide cache coherency in many-core architectures.
- Software that can be cast in terms of interleaved stages of computation and structured communication, namely collective communication, can be supported by distributed-memory, many-core architectures such as SCC. One collective communication not commonly contained in other message passing libraries, which we call *permutation* (see Section 4.3), was discovered in the process and added to our set of supported collectives. Its utility extends well beyond Elemental, enabling many advanced parallelization strategies.
- If one invests in learning from prior art, in our case the ScaLAPACK and PLAPACK libraries, and redesigns software with an eye on programmability and layering, shifting the burden of parallelization from the architecture to the programmer can be done while keeping programming many-core architectures manageable.

Together, these insights advance the understanding of the subject.

The rest of the paper is organized as follows. We provide a brief overview of SCC in Section 2. Section 3 discusses Elemental along with our efforts to port it to SCC. The implementation of collective communications on SCC is described in Section 4. Performance results are provided in Section 5. We discuss the lessons learned from our efforts and plans for future work in Section 6.

2 The SCC Processor

The Single-chip Cloud Computer (SCC) processor is an experimental processor [17] from Intel Labs. It uses an architecture that can scale to many hundreds of cores as the density of transistors that can be placed within a single chip continues to increase. Manufactured with Intel's production 45 nm process technology, the SCC project explores hardware questions such as low power routers, explicit power management, and scalable network-on-a-chip architectures. Its most important role, however, is as a platform for many-core software research. We do not investigate the power management capabilities of SCC in this paper. Instead, we explore the programmability and scalability features of such a chip.

The SCC processor was created through a software/hardware co-design process. As the processor was designed, a native message passing environment was developed for the chip [20]. By using a functional emulator, we were able to develop applications and propose changes to the processor architecture as it was being developed. In this section, we briefly review the architecture of the SCC processor and its native message passing environment.

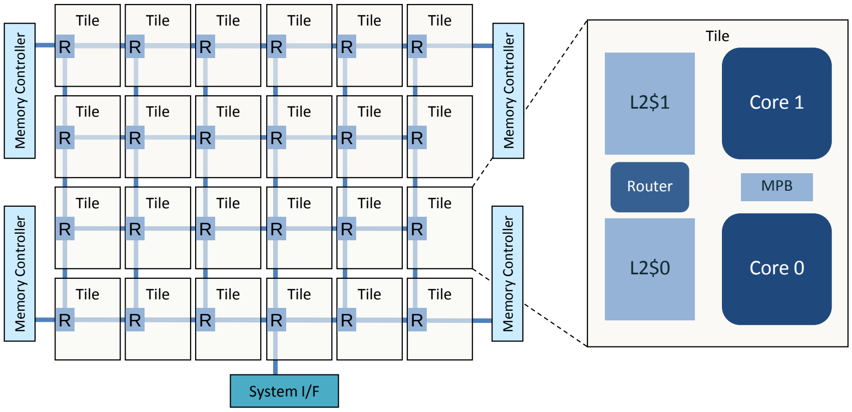

Figure 1: SCC architecture is comprised of a 4×6 grid of tiles where each tile contains a pair of cores (with L1 and L2 caches), a router, and 16 KB of shared SRAM to serve as a message passing buffer.

2.1 The SCC Architecture

The SCC processor architecture is shown in Figure 1. The processor consists of 24 tiles organized into a 4×6 grid. The routers implement fixed X-Y routing, which reduces the energy consumed by the network [17] compared to a more general adaptive routing. The on-die network extends off the grid at four locations on edges of the chips to connect to four DDR3 on-die memory controllers for 16 to 64 GB of off-die DRAM memory. It also extends off the edge of the chip at one point to provide a PCI interface.

A tile contains a pair of minimally modified P54C processor cores [2], each with an independent L1 (16 KB data and 16 KB instruction) and L2 (256 KB unified instruction/data) cache. The cores are second generation Pentium® processors selected because of their low power, in-order architecture and the fact that they were available as “off the shelf” designs that could be directly synthesized from RTL. The choice of this core seriously limits the raw performance of the chip but does not impede progress on the key research vectors for the project, e.g., programmability, scalability, power management.

Each tile also includes a router and a 16 KB block of SRAM. These memory blocks are organized into a shared address space visible to all cores on the chip. This memory was added to support the movement of L1 cache lines between cores and hence is called the “message passing buffer” (MPB). It is important to appreciate that the processor does not maintain cache coherency between cores for any memory region on the chip. All issues of coherency or consistency are managed explicitly by the programmer. When working with the SCC processors, programmers are exposed to three distinct address spaces:

- A private memory in off-chip DRAM for each core. This memory is cache coherent with an individual cores's L1 and L2 caches.
- The MPB that has 24×16 KB of shared memory in SRAM.
- A shared-memory off-chip address space in DRAM. This memory may be configured as uncached or cached, but in the latter case cache coherence between cores is not maintained by the SCC processor.

The MPB is an important feature of the SCC processor. Since the private memory associated with each core is a distinct address space, cores cannot exchange information by “passing pointers”. The MPB lets cores exchange information in the form of messages at the granularity of L1 cache lines. Because it is on-die, the MPB provides a low-overhead mechanism to move blocks of data from one core's L1 to another's L1 and ultimately between private memories. It would be possible to exchange data through the shared-memory in the off-chip DRAM, but this would suffer from higher latency and lower bandwidth.

The SCC processor lets programmers manipulate the details of how each core interacts with the different address spaces. This is done by modifying entries in an address translation lookup table. This capability would be too dangerous to expose in a processor product, but in a platform for software research it opens up a range of research opportunities on how to manage shared, non-coherent memory in a many-core platform.

SCC enables researchers to test programming a many-core processor using each of these options. Different memory models are expected to have different programmability issues and performance characteristics, which will be important to study as more cores are added to chips and software complexity increases.

In this paper, we view the SCC processor as a collection of cores with local memories, communicating through a message passing library described below, and we test its programmability as an integrated cluster. Future work by our group and others in the SCC research community will explore other programming models, in particular models that make direct use of the shared, off-chip memory available on the SCC processor.

2.2 RCCE Communication Library

RCCE (pronounced "rocky") [20, 28] is a light-weight communication library developed by Intel for the SCC processor. It defines low-latency mechanisms to move data stored in the private memory of one core to the private memory of another core. The most common usage model for RCCE assumes synchronous communication. Cores that need to exchange information wait for all participating cores to reach corresponding points in their execution. Then they cooperatively exchange data as needed. This approach is common with Message Passing Interface (MPI) [14] applications targeting cluster computers.

At the lowest level, RCCE provides a one-sided communication layer. The basic RCCE design treats the MPB as a set of 8 KB buffers, each designated to one core. To move a cache line from one core to another, the sending core "puts" (copies) a cache line into its own buffer from which the receiving core "gets" (copies) the cache line, thereby moving it into its own L1 cache. Programmers need to coordinate movement of cache lines into and out of the MPB. This is done with "flags", i.e., synchronization variables within RCCE.

The basic one-sided communication API within RCCE is flexible and can handle a wide range of communication patterns, but it can be a complicated approach. For example, if the message size exceeds the space for messages within a core's buffers (8 KB minus any space needed to support the synchronization flags), the programmer must decompose the messages into smaller packets that individually fit in a core's MPB. Moreover, messages must be L1 cache-line aligned and sized to a multiple of the cache line, or special precautions must be taken to avoid having data stuck in SCC's so-called write-combine buffer. We quickly recognized that unrestricted, higher level, two-sided communication primitives, much like the send and receive functions that MPI provides, were needed. Several more simplified, MPI-like functions were added on an as-needed basis, except asynchronous communication.

The exclusion of asynchronous communication in RCCE deserves further comment. The SCC processor typically executes with a Linux Kernel running on each core. Given Linux, we can execute with multiple threads on each core, thereby supporting asynchronous communication. An alternative mode of using SCC, however, uses a low level operating system-less mode, which we call *baremetal mode*. We designed RCCE so programs can be built and executed in Linux and baremetal mode without a change in source code. The cost is that programmers must convert asynchronous algorithms to ones that use synchronous communication. An important observation of this paper is that for the dense linear algebra functions we have explored, the restriction of synchronous communication is not a problem because of the way the ported library is programmed.

As mentioned earlier, RCCE was developed as part of a hardware/software co-design project. To support this effort, we created a functional emulator of RCCE execution on SCC. This functionality let us develop software and explore features of the SCC processor as it was being designed. The emulator used OpenMP [1] to model the MPB, so RCCE applications can run on any system that supports OpenMP. Once the SCC design was complete, the emulator proved to be of great value as a development platform for porting, debugging, and developing software for the SCC processor. We mention this because the dense linear algebra library described in this paper was ported to SCC using the emulator, and with few exceptions, we only had to relink our software to run on actual SCC hardware.

3 Elemental

In this section, we give a brief overview of the Elemental library. Cholesky factorization is used as a representative operation to illustrate some of the programming issues and how Elemental addresses them.

<table>
<tr><td colspan="3">Applications</td></tr>
<tr><td colspan="3">Elemental
Solvers</td></tr>
<tr><td colspan="3">Elemental
BLAS/Decomposition/Reduction/$\cdots$</td></tr>
<tr><td>Elemental
Local Operations</td><td colspan="2">Elemental
Redistribution Operations</td></tr>
<tr><td>Local Compute
Kernels
(BLAS/LAPACK)</td><td>Packing
Routines</td><td>Collective
Communication
(MPI or RCCE)</td></tr>
</table>

Figure 2: Layering of the Elemental library.

3.1 Background

LINPACK [9] can be considered the first numerical package that tried to address programmability in addition to functionality and numerical stability. Its developers adopted the Basic Linear Algebra Subprograms (BLAS) [19] interface for portable performance. Subsequently, the Linear Algebra PACKage (LAPACK) [3] was developed to provide higher performance on cache-based architectures by adopting new layers of the BLAS [10, 11] as well as added functionality and stability. As distributed-memory architectures became increasingly common, a level of abstraction was needed to support dense linear algebra on these systems. This led to ScaLAPACK [8], which extended LAPACK functionality to distributed-memory computer architectures.

The goal of ScaLAPACK was performance and functionality and on those fronts it has been a very successful package. However, the project did not place emphasis on programmability. PLAPACK [26], a dense linear algebra package for distributed-memory machines similar in functionality to ScaLAPACK, added programmability as a key focus to its design. The central idea is that the algorithms used for dense matrix computations should be apparent in the source code. The group behind PLAPACK has continued this line of research, exploring systematic derivation of linear algebra algorithms supported by clear abstractions to support their expression in source code. This led to a new, sequential dense linear algebra library, `libflame` [29], and more recently a new dense linear algebra library for distributed-memory architectures, Elemental [21]. It is this library that is at the heart of our experiment since it was designed for conventional clusters but appeared suitable to be ported to architectures like SCC.

Elemental solves scalability problems encountered in PLAPACK and programmability problems encountered in ScaLAPACK, which is explained in Section 5.3. For SCC Elemental has a few obvious advantages: 1) it uses a simple data distribution; 2) it is carefully layered, as shown in Figure 2; 3) it uses abstractions that allow the programmer to code at the level at which one reasons about the algorithm; and, importantly, 4) all communication is cast in term of collective communication. These features improve programmability of the library and greatly eased the porting effort.

3.2 A Motivating Example: Cholesky Factorization

If $A \in \mathbb{R}^{n \times n}$ is a symmetric, positive-definite matrix, then Cholesky factorization computes $A \rightarrow U^T U$ where U is an upper triangular matrix. One algorithm, often called the right-looking variant, is presented using FLAME notation [5, 15] in Figure 3. This figure presents a blocked algorithm that casts most computation in terms of matrix-matrix computations (level-3 BLAS), allowing it to attain high performance on cache-based architectures.

The FLAME notation helps solve the programmability problem by allowing algorithms to be derived to be correct [4], meaning that as functionality is added to libraries like Elemental and `libflame`, a high level of confidence can be placed in the correctness of the resulting algorithms.

3.3 Representing Algorithms in Code

The first feature of Elemental that aids programmability is that it is coded in C++ using modern, object-oriented coding practices, a deviation from the implementation of the alternative packages ScaLAPACK and PLAPACK, which were respectively coded in Fortran77 and C.

```
Algorithm: $A := \textsc{Chol\_blk}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
  Determine block size $b$
  Repartition
  $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
    where $A_{11}$ is $b \times b$

  $A_{11} := \textsc{Chol}(A_{11})$
  $A_{12} := A_{11}^{-H} A_{12}$            (TRSM)
  $A_{22} := A_{22} - A_{12}^{H} A_{12}$    (TRIANGULAR RANK-K)

  Continue with
  $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
endwhile
```

Figure 3: Blocked, right-looking algorithm for computing the Cholesky factorization. Note that the algorithm is for both a real and complex valued matrix A where A^H denotes conjugate transposition.

The Elemental implementation of the Cholesky algorithm is given in Figure 4. This code is representative of the layer that is labeled "BLAS/Decomposition/Reduction/$\cdots$" in Figure 2. Those familiar with the FLAME project will recognize that, like other FLAME related APIs, the code resembles the algorithm in Figure 3, hiding indexing details. In the case of Elemental, the code also encapsulates details about how the matrix is distributed among cores (or, in the case of MPI, processes). Thus, at the top level, Elemental partially solves the programmability problem by using an API that reduces the opportunity for introducing "bugs" when the algorithm in Figure 3 is translated into the code in Figure 4.

3.4 Distribution and Redistribution

To understand how parallelism is expressed in the Elemental code, one must first understand a little about the data distributions used by Elemental and how redistribution is expressed.

When parallelizing a sub-operation on distributed-memory architectures, data is initially distributed in some specified fashion among the processes. Ideally, each process has all data, so all can simultaneously perform independent local operations. In practice, this often requires data to be duplicated or redistributed among the processes before local computation commences. Often, the local computation is a contribution to a global result, and data must be reduced (e.g., summed) leaving it in some prescribed distribution.

To support scalability, dense linear algebra libraries often distribute matrices by viewing the processes as a logical two-dimensional mesh. These libraries attain load balance as the computation proceeds by wrapping matrices cyclically around the process mesh [16, 24, 25]. Elemental partially solves the programmability problem by choosing the simplest such distribution: the p processes are viewed as forming an $r \times c$ logical mesh, and the elements of a given matrix A are wrapped using an *elemental* 2D cyclic distribution, which means that element (i, j) is assigned to process $(i\%r, j\%c)$.

This is in contrast to PLAPACK and ScaLAPACK, which use a more complex block cyclic distribution. In those packages, the blocksize b_{distr} is used. Blocks of size b_{distr} by $r \times b_{\text{distr}}$ in PLAPACK and of size b_{distr} by b_{distr} in ScaLAPACK are wrapped around the processes grid in a 2D cyclic fashion. As a result, indexing and redistribution are more complicated because the "owning" process for element (i, j) is not simply $(i\%r, j\%c)$ as it is in Elemental. Thus, the code within Elemental related to the distribution and redistribution of data is much simpler than in the other packages. Furthermore, PLAPACK's distribution, tied to the number of rows in the process grid, makes code mildly non-scalable when the number of processors becomes large enough and the matrix fills all available memory. In [21] it is shown that Elemental's simplification does *not* adversely affect performance on traditional clusters.

```
1  template<typename T> void
2  elemental::lapack::internal::Chol_blk
3  ( DistMatrix<T,MC,MR>& A )
4  {
5     const Grid& g = A.GetGrid();
6
7     DistMatrix<T,MC,MR>
8        ATL(g), ATR(g),   A00(g), A01(g), A02(g),
9        ABL(g), ABR(g),   A10(g), A11(g), A12(g),
10                         A20(g), A21(g), A22(g);
11
12    DistMatrix<T,Star,Star> A11_Star_Star(g);
13    DistMatrix<T,Star,VR  > A12_Star_VR(g);
14    DistMatrix<T,Star,MC  > A12_Star_MC(g);
15    DistMatrix<T,Star,MR  > A12_Star_MR(g);
16
17    PartitionDownDiagonal
18    ( A, ATL, ATR,
19         ABL, ABR, 0 );
20
21    while( ABR.Height() > 0 )
22    {
23       RepartitionDownDiagonal
24       ( ATL, /**/ ATR,   A00, /**/ A01, A02,
25         /************/   /*****************/
26              /**/        A10, /**/ A11, A12,
27         ABL, /**/ ABR,   A20, /**/ A21, A22 );
28
29       A12_Star_MC.AlignWith( A22 );
30       A12_Star_MR.AlignWith( A22 );
31       A12_Star_VR.AlignWith( A22 );
32
33       //------------------------------------------//
34       A11_Star_Star = A11;
35       lapack::internal::LocalChol( Upper,
36                                    A11_Star_Star );
37       A11 = A11_Star_Star;
38
39       A12_Star_VR = A12;
40       blas::internal::LocalTrsm
41       ( Left, Upper, ConjugateTranspose, NonUnit,
42         (T)1, A11_Star_Star, A12_Star_VR );
43
44       A12_Star_MC = A12_Star_VR;
45       A12_Star_MR = A12_Star_VR;
46       blas::internal::LocalTriangularRankK
47       ( Upper, ConjugateTranspose,
48         (T)-1, A12_Star_MC, A12_Star_MR,
49         (T)1, A22 );
50       A12 = A12_Star_MR;
51       //------------------------------------------//
52
53       A12_Star_MC.FreeAlignments();
54       A12_Star_MR.FreeAlignments();
55       A12_Star_VR.FreeAlignments();
56
57       SlidePartitionDownDiagonal
58       ( ATL, /**/ ATR,   A00, A01, /**/ A02,
59              /**/        A10, A11, /**/ A12,
60         /************/   /*****************/
61         ABL, /**/ ABR,   A20, A21, /**/ A22 );
62    }
63    return;
64 }
```

Figure 4: Elemental implementation of the blocked, right-looking algorithm for the Cholesky factorization.

Now, let us turn to the first operation to be performed in the loop: $A_{11} := \textsc{Chol}(A_{11})$. The problem is that the elements of A_{11} are scattered among the processes. The command in line 34 of Figure 4 (together with the distribution indicated in line 12) redistributes this submatrix so that all processes own a copy in variable `A11_Star_Star` after which the processes locally and redundantly compute its Cholesky factorization. In this case, the data of A_{11} is stored such that process $(i\%r, j\%c)$ owns element (i, j) of the block. In order for all processes to own A_{11} redundantly, each must communicate its locally-stored portion of the block with all others. That collective communication operation is called allgather. Another redistribution used in other algorithms distributes element (i, j) to process $(i\%c, j\%r)$. Data can be redistributed to this format from $(i\%r, j\%c)$ using the routines alltoall and send-receive. We discuss the interface and implementation of these collective communication operations in Section 4.

Elemental supports a handful of other data distributions to enable algorithm coders to parallelize operations in many ways. To redistribute data among processes from one distribution to another, Elemental *only* uses collective communication. Figure 4 shows multiple examples of this. Where is the communication? Elemental partially solves the programmability problem by hiding the required collective communication in the overloaded = operator. Each of the commands in lines 34-50 in Figure 4 represent either a redistribution or local computation, but the specific details are hidden. Thus, the layering and abstractions in Elemental mirror the natural way in which parallel computation is decomposed. In [21] it is shown that that this layering and abstraction does *not* adversely affect performance on traditional clusters.

For this paper, it is not important to understand all of the distributions and communication necessary to redistribute data in Elemental. For details about the distributions and communication see [21]. It is important to understand that data is distributed among processes and is only communicated between processes using collective communication routines. Furthermore, communication details are handled by Elemental behind the code seen in Figure 4. This encapsulation aids programmability when implementing an algorithm using Elemental and aids programmability and portability within the library itself.

3.5 Retargeting to SCC

At the onset of this research, there were multiple reasons why retargeting Elemental to SCC appeared to be a natural fit. Since the primary purpose of SCC is to investigate programmability issues related to many-core architectures, it made sense to tap into the focus of the FLAME project on programmability. Elemental, which builds on the FLAME project, was developed for conventional clusters and one of the memory models of SCC allows us to view it as a distributed-memory system (cluster) on a single chip. Elemental uses collective communication, which maps well to the synchronous point-to-point communication supported by RCCE. Message passing, used indirectly via collective communication in the case of Elemental, is a model that avoids having to explicitly manage coherency between cores since this is handled within the message passing primitives. Finally, the FLAME project's emphasis on program correctness and the abstractions developed for Elemental gave a high degree of confidence in that code base, meaning that as the port proceeded there was never a question of whether there was a latent logic error in Elemental. If something did not work, the problem was always with the small number of routines that were tailored to SCC and RCCE.

To understand where changes had to be made requires an explanation of what happens when a redistribution is triggered by a command like the one in line 34 of Figure 4. The processes recognize the "before" and "after" distributions and determine that data from all processes must be communicated to all processes by an allgather. Before a routine that implements allgather can be called, though, the local data must be rearranged (packed) into a convenient format. After completion of the collective communication, it must again be locally rearranged (unpacked) by each process. In between is a call to an MPI collective communication on a conventional cluster. This call needed only to be replaced by a call to an equivalent RCCE collective communication. Thus, it is only in the box labeled "Collective Communication" in Figure 2 that changes were made. Said another way, the focus on programmability in designing and layering Elemental allowed us to retarget the library to a new architecture with minimal changes outside that layer in the code.

When this research commenced, no SCC processor was available for the port, so it was performed with the aid of the previously mentioned RCCE emulator. The major challenge was that only some of the collective communications used by Elemental were part of RCCE's collective communication library. We discuss how we dealt with that challenge in the next section. Conveniently, at the heart of the FLAME project is a methodology for systematically deriving different algorithmic variants for a given linear algebra operations [27]. Different variants often require different redistributions and hence even when initially only a few collective communications were available in RCCE, broad functionality of Elemental came on-line. The benefit of eventually having all collective communication used in Elemental available was that all variants for all operations supported by Elemental were also available, meaning that the best-performing algorithmic variant for distributed-memory computing could be employed. For example, there are

RCCE interface
Send
`int RCCE_send( char* buf, size_t num, int dest );`
Receive
`int RCCE_recv( char* buf, size_t num, int src );`
Allgather

`int RCCE_allgather( char* inbuf, char* outbuf,`
`size_t num, RCCE_COMM comm );`

Alltoall

`int RCCE_alltoall( char* inbuf, char* outbuf,`
`size_t num, RCCE_COMM comm );`

Send-receive

`int RCCE_sendrecv( char* inbuf, size_t in, int dest,`
`char* outbuf, size_t out, int src,`
`RCCE_COMM comm );`

Figure 5: RCCE interface for the communication routines used within Elemental.

three commonly used variants for Cholesky factorization. The initial port of Elemental to SCC only supported one of those variants, the left-looking variant, because RCCE does not implement all of the communication patterns in MPI or even all of those used by Elemental. Eventually, the right-looking variant in Figures 3 and 4, which parallelizes more naturally, was supported as we introduced more collective communication operations.

4 Collective Communication

All of the communication between processes in Elemental is cast in terms of collective communication. RCCE only provides a small set of unoptimized collective communication routines (collectives) such as broadcast, where a vector of data on one process is sent to all other processes. Elemental uses several standard collective communication routines provided by MPI that are not supported by RCCE, so we implemented these operations using only the simple, synchronous point-to-point communication routines (send and receive).

In Figure 5, we provide the interface for the communication routines in RCCE along with those we developed. Notice that we attempt to mimic the MPI interface [14] for these routines as closely as possible to ease the porting effort. We also provide an illustration of the collective communication routines we study in this section, allgather and alltoall, in Figure 6.

4.1 Allgather

For the allgather operation, initially each process p_i owns a subvector of data x_i where

$$x = \left(\begin{array}{c} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{array} \right)$$

and upon completion each of the p processes owns the entire vector x.

A relatively simple algorithm for allgather is the cyclic, or bucket, algorithm [6]:

Before			After		
Allgather					
p_0	p_1	p_2	p_0	p_1	p_2
x_0			x_0	x_0	x_0
	x_1		x_1	x_1	x_1
		x_2	x_2	x_2	x_2
Alltoall					
p_0	p_1	p_2	p_0	p_1	p_2
$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(0)}$	$x_1^{(0)}$	$x_2^{(0)}$
$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_0^{(1)}$	$x_1^{(1)}$	$x_2^{(1)}$
$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_0^{(2)}$	$x_1^{(2)}$	$x_2^{(2)}$

Figure 6: Illustrations of allgather and alltoall with three processes.

```
$l = (me - 1 + p)\%p$
$r = (me + 1)\%p$
$i = me$
for $j = 1, \ldots, p - 1$ do
    $k = (i + 1)\%p$
    Send $x_i$ to $p_l$
    Receive $x_k$ from $p_r$
    $i = k$
end
```

where me denotes the rank of the calling process. Within an iteration of this algorithm, each process sends its local contribution to its neighboring process on the (logical) left and receives from the (logical) right. This algorithm inherently sends data in a circular communication pattern. Since both the send and receive in RCCE are blocking, deadlock occurs with this cyclic algorithm if implemented in a single communication step. All processes first call the send routine and will block until the corresponding receive is posted, but no process will do so since all are blocked on the send.

This deadlock can be easily resolved using two steps for an even number of processes. In the first step, all the even numbered processes send while all odd numbered processes receive. In the second step, evens receive and odds send. A problem arises when the number of processes is odd, but we can avoid deadlock by introducing a third step. In the first step, all evens send to odds, excluding the wrap-around where process p_{p-1} sends to p_0. In the second step, all odds send to evens. Finally, the wrap-around occurs where the first and last ranked processes communicate with each other. As a result, deadlock can be avoided by detecting a ring communication pattern and performing this odd and even decomposition of the sends and receives. We illustrate deadlock prevention of these two cases in Figure 7 using four and five processes as examples.

4.2 Alltoall

The alltoall operation performs a permutation of the vector x on each process. Initially, each process p_i owns a vector $x^{(i)}$ that is partitioned where $x_j^{(i)}$, $j = 0, \ldots, p - 1$. Upon completion the process p_j owns the permuted vector $x_j^{(i)}$, $i = 0, \ldots, p - 1$.

A relatively simple algorithm for alltoall is a staged pairwise exchange algorithm [23]:

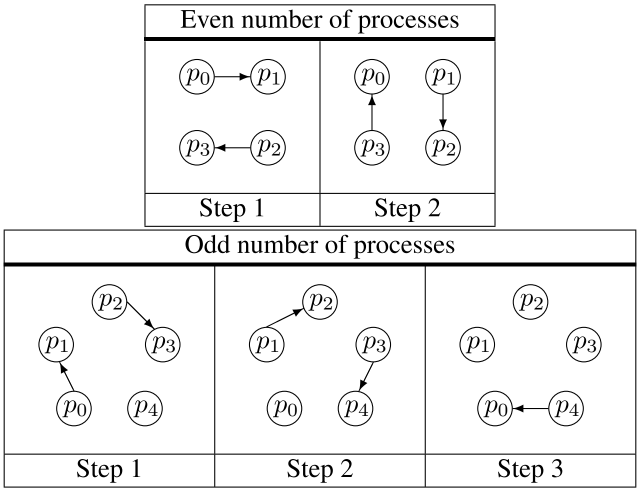

Figure 7: Deadlock prevention within a ring communication pattern when given even and odd number of processes.

```
for $j = 0, \ldots, p-1$ do
    $i = (j - me + p)\%p$
    if $i \neq me$ then
        Send $x_i$ to $p_i$
        Receive $x_i$ from $p_i$
    end
end
```

This algorithm is deadlock free because only distinct pairs of cores communicate with each other during each stage. We can play the trick of having the lower ranked process send first while the higher ranked process receives, and then reverse roles. The algorithm can be improved slightly for even numbers of processes [28], yielding a provably optimal schedule.

4.3 Send-receive

On the surface, one important redistribution in Elemental is not a collective communication. Instead, it requires point-to-point communication similar to MPI's send-receive. The MPI routine can be thought of as combining a send and a receive (possibly from a different process) into a single routine. In general this would be difficult to implement in terms of synchronous point-to-point communications. Fortunately, the communication that requires this in Elemental, and many other codes, can be viewed as a collective communication that implements a permutation where every process sends data to one other process and also receives data from one other process. It is this permutation functionality that we support.

We implement this routine by first distributing all the sending and receiving ranks for each process via an alltoall operation, so all processes know how all other processes will communicate. The resulting communication graph consists of a collection of linear (open) chains and/or cycles, each of which can always be implemented in a maximum of three communication stages (two in the case of a linear chain or cycle with an even number of nodes). We can "cache" this communication pattern so that the communication graph construction is only performed once and is subsequently reused in Elemental. As a result, we do not need to call alltoall each time the permutation is invoked.

5 Performance

We endeavored to test programmability and scalability of SCC, a possible look at the future of many-core processors. As explained above, the abstractions and layering used enabled us to retarget Elemental code to SCC with limited changes. As for scalability, we compare the performance on varying numbers of cores for three dense matrix operations: Cholesky factorization; LU factorization with partial pivoting, $PA := LU$ where $A \in \mathbb{R}^{n \times n}$, L is a lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix; and general matrix-matrix multiplication (GEMM), $C := \alpha AB + \beta C$ where $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, and $C \in \mathbb{R}^{m \times n}$.

We tune the algorithmic block size and the grid configuration within Elemental. We link Elemental with the Intel Math Kernel Library (MKL) 8.1.1 for the execution of the computational kernels on each core using double precision floating point real arithmetic. We compare the scalabilty of Elemental versus a sequential MKL implementation: `dpotrf` for CHOL, `dgetrf` for LU, and `dgemm` for GEMM. MKL is used for single-core performance.

Notice that using the default clock frequency of 533 MHz, each core has a theoretical peak performance of 533 MFLOPS giving SCC a total theoretical peak performance of 25.584 GFLOPS. However, this is not a realistic performance target given that MKL's `dgemm` on a single core only achieves around 120 MFLOPS. Hence, this is not an experiment on how to achieve near-peak performance.

5.1 Results

We provide performance results in Figures 8, 9, and 10. In the left of those figures, we compare the scalability of Elemental by fixing a few sample problem sizes and showing performance using varying numbers of cores. On the right, we show the breakdown of the different component costs within implementations using all 48 cores and varying the problem sizes. The computation cost only entails the time each process spends executing a sequential kernel when linking to MKL. The communication cost is the time spent in collective communication routines, which includes copying between private memory and MPB. The overhead contains all remaining components of the execution such as packing and unpacking data to and from contiguous application-level buffers.

Consider the performance of matrix multiplication for a single core, found on the left of Figure 10, which uses the sequential MKL `dgemm` kernel.[1] Given the cache-friendliness of GEMM, an ambitious speedup on n cores would be n times this single-core performance. As we increase the number of cores available to Elemental, performance increases with no obvious "knees" in the curves indicating diminishing marginal utility. Given the way Elemental distributes data and parallelizes algorithms, we believe it would scale well on processors similar to SCC with even more cores. Such scalability is seen in [21] on cluster computers composed of many cores.

Notice that scalability improves with larger problem sizes as the computational time on p processes is $O(n^3/p)$ while the communication and packing related time is $O(n^2/\sqrt{p})$ (and hence $O(n^3)$ versus $O(n^2)$ when p is fixed). This trend continues with larger problem sizes than those shown. This is typical behavior for these operations on clusters, as costly communication is a larger portion of execution time for smaller problem sizes than larger ones. Notice the decrease of the communication cost portion in the component graphs as the problem sizes increase, and the computational portion simultaneously increases. As communication and overhead costs are relatively smaller portions of overall performance for larger problem sizes, speedup improves as the problem size increases.

In Figure 11 (left), we show the performance of a representative set of operations supported in Elemental that have been ported to SCC. These include all level-3 BLAS and several LAPACK-level operations. This graph illustrates the benefits of programmability when porting a wide range of operations to this new architecture. For each of these operations, Elemental contains multiple algorithmic variants because Elemental addresses programmability. Different variants exhibit constrasting performance characteristics and use different communication patterns. Without the communication routines described in Section 4, only two of the operations in this graph work on SCC. If only a single variant of each operation were available, we could not test correctness of early ports of Elemental as easily because we would have to implement new algorithm variants in addition to porting the library. Instead, we are able to choose variants of those operations that only call collective communication functions available in RCCE. With the additional communication routines of Section 4, all variants of the remaining operations work immediately. We show the default variant of each operation in this graph.

The absolute performance of Elemental on SCC shown in these graphs is rather poor, even considering the weak P54C cores. This is largely the result of unoptimized sequential BLAS and LAPACK implementations. Furthermore, these are early performance results, and we have not spent much time yet optimizing Elemental for this architecture. Our initial goals were to test scalability and programmability for which we show good results.

[1] Due to per-core memory limits, only two of the problem sizes fit on a single core.

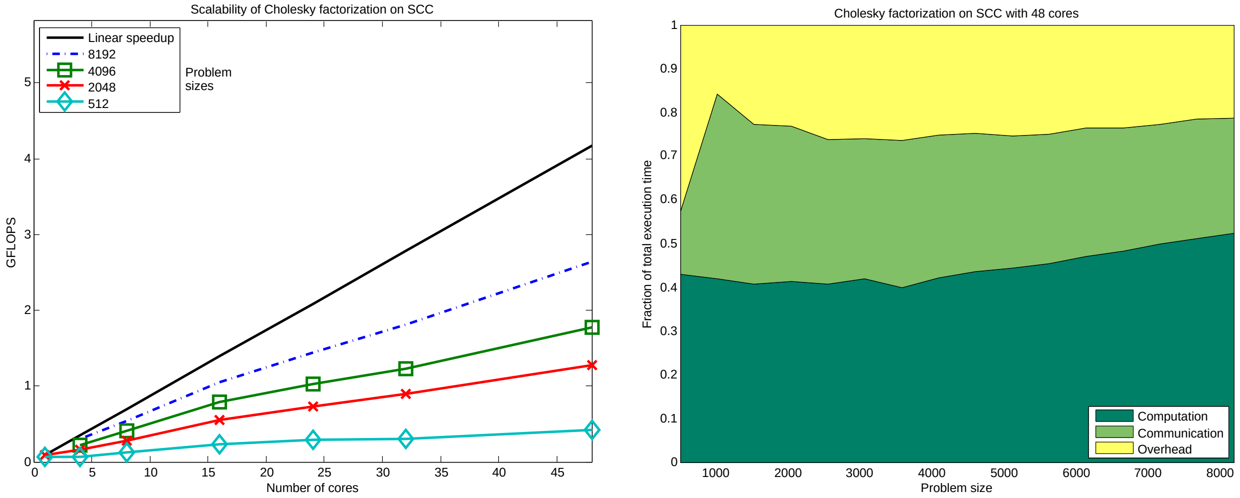

Figure 8: Scalability (left) and cost breakdown for 48 cores (right) of Elemental's implementation of Cholesky factorization.

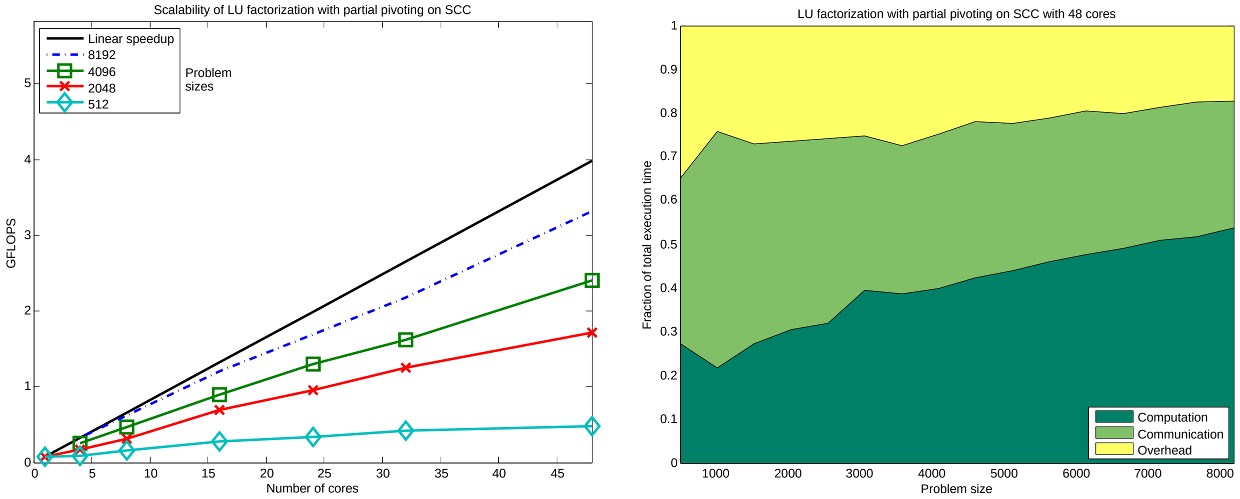

Figure 9: Scalability (left) and cost breakdown for 48 cores (right) of Elemental's implementation of LU factorization with partial pivoting.

5.2 High-Performance LINPACK

The High-Performance LINPACK (HPL) is a highly specialized implementation of the LINPACK benchmark [12] for massively parallel distributed-memory systems that was partially ported to SCC by one of the co-authors. HPL performs a large LU factorization with partial pivoting, and much like ScaLAPACK, it uses a block cyclic data distribution and fundamentally does not address programmability. Details such as pipelining where communication and computations are overlapped are exposed directly within the code.

In order to port HPL to SCC, we replaced all the asynchronous MPI communication calls with synchronous RCCE routines. Since RCCE only provides blocking calls, deadlock had to be detected and explicitly avoided. This required non-trivial analysis of the HPL communication patterns underlying the point-to-point messages, thereby greatly complicating the port.

Elemental's implementation of LU factorization with partial pivoting is compared against HPL in Figure 11 (right).

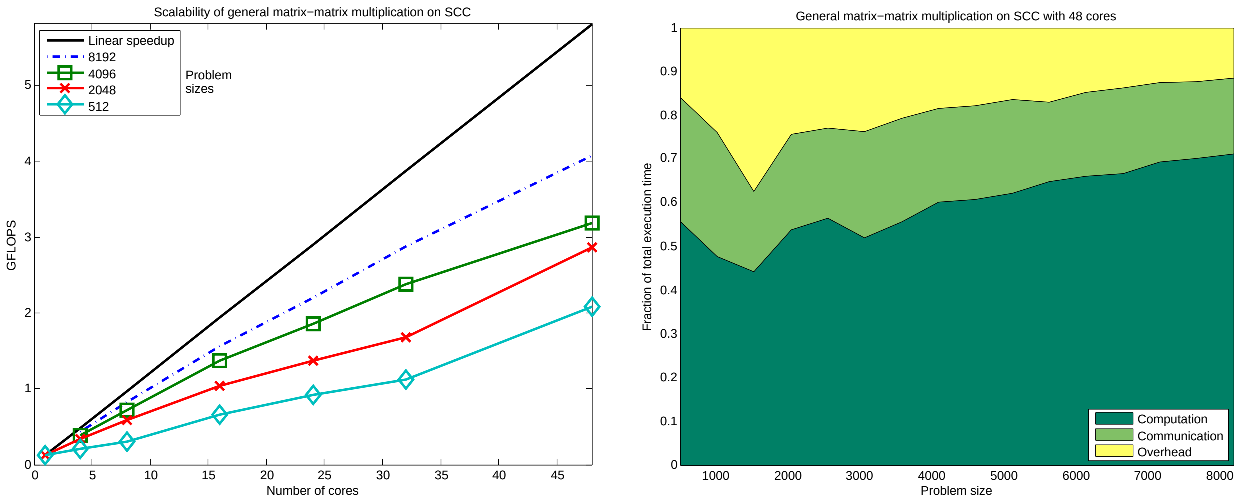

Figure 10: Scalability (left) and cost breakdown for 48 cores (right) of Elemental's implementation of general matrix-matrix multiplication where the matrix dimensions are $m = n$ and $k = 1280$.

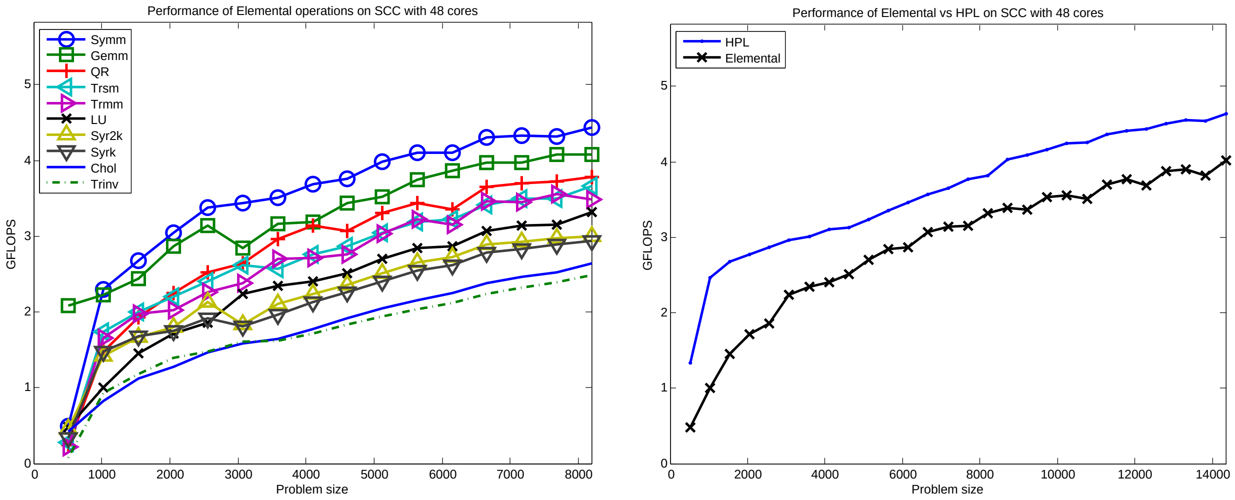

Figure 11: Performance of a cross-section of operations supported by Elemental that have been ported to SCC (left) and LU factorization with partial pivoting for Elemental versus HPL (right) using all 48 cores.

HPL requires tuning of a number of parameters. We are not able to tune all parameters because HPL is not completely ported to SCC. The tuning parameters, which often involve algorithmic variants, create a large quantity of code to port, and the effort required cannot be justified merely for the purpose of comparison. We tune as much as possible for a single problem size and use the best choices for all runs with the exception of the block size for which the optimal setting changes for small and large problems.

Although Elemental's performance is lower than HPL's, the difference is fairly modest, especially for larger problem sizes and is expected to narrow even further as Elemental is further optimized. We are already investigating improvements to Elemental's LU factorization implementation and describe some promising optimizations to the port in Section 6.2. Moreover, Elemental is a general-purpose library created to enable many algorithms to be developed for distributed-memory computers whereas HPL is a benchmark meant solely to achieve good performance for this one particular operation. We consider Elemental's relative performance deficit the result of a reasonable compromise between speed and versatility/programmability.

Regarding the latter, we note that even the incomplete port of HPL to RCCE required the assistance of the author of the original code because of the complicating effect of the point-to-point communications. Elemental, in contrast, was much easier to port completely, as it fully isolates the required data transport in a modest collection of generic collective communications [**?**]. Although the main author of Elemental is a co-author of this paper, the port was accomplished by another co-author of this paper who had little experience with Elemental and distributed-memory computing and received virtually no help from the author of Elemental.

5.3 Porting ScaLAPACK

One could argue that a comparison between Elemental and ScaLAPACK would have been a better experiment. However, ScaLAPACK contains many point-to-point communications and a much larger body of code than HPL, which prevented us from attempting the port.

To quantify this last statement, we point out a few key issues. First, major design decisions regarding ScaLAPACK were made prior to the arrival of MPI. Second, ScaLAPACK, by design, is layered and coded to closely resemble LAPACK. As a result, the library-level code is layered upon the parallel BLAS (PBLAS) layer, which itself is layered upon standard (local) BLAS and the Basic Linear Algebra Communication Subprograms (BLACS), a communication layer that has an interface that resembles the BLAS interface [13]. The BLACS themselves are coded in terms of what, at the time, were a myriad of native communication libraries. The most commonly used implementation is now layered upon MPI. The BLACS include both point-to-point and collective primitives.

In principle the BLACS collective communications should be easy to port. In practice the BLACS implement an array of algorithms for collective communication without relying on the MPI interface. Still, it would be a matter of simplifying this implementation so that they call only the collective communications that were developed as part of our effort. This might possibly go at the expense of performance since ScaLAPACK depends on pipelining between communication and computation in a number of important routines in order to reduce communication overhead.

The more troublesome aspect of a port of ScaLAPACK comes from its use of point-point communications. In the PBLAS we found 37 instances of calls to `DGESD2D`, the BLACS send primitive for communicating double precision data. At the library level (LAPACK-level functionality), we found 168 such instances. Each of these may need to be examined to determine whether the communication can be performed synchronously and possibly reimplemented so that it can be performed synchronously. Not counted here are a large number of calls in the ScaLAPACK test suite and redistribution routines.

The point is that porting ScaLAPACK is possible but labor intensive. By comparison, the only place where point-to-point communications are called by Elemental is in its communication layer where we automatically avoid deadlock and communication serialization. On conventional architectures, Elemental delivers performance that is competitive with, and often exceeds, that of ScaLAPACK [21].

6 Conclusion

In this paper, we have described our experiences related to the porting of a major software library, Elemental, to the SCC research processor. We started with the conjecture that for some problem domains software supported coherency of data on many-core architectures can be achieved by viewing the architecture as a distributed-memory parallel computer architecture and communicating data via message passing constructs. For the domain of dense matrix computations, the results provide early evidence that this is indeed the case when one starts with a library that already targets distributed-memory architectures and is very carefully layered. It is shown that a minimal set of communication primitives is needed to support this, namely collective communication.

6.1 Insights

We targeted a problem domain that is thought to be well-understood but has struggled with the complications of parallel computing for two decades. Fortunately, that struggle allowed insight to be gained from legacy libraries, ScaLAPACK and PLAPACK, yielding a highly layered library, Elemental, that fundamentally addresses the programmability problem for the domain of dense matrix computations. As a result, this library ported naturally to SCC processor, building on the RCCE communication library.

A question is how representative the domain of dense matrix computations is of other software libraries and "real" applications. A careful look at our results shows that any application that casts its communication in terms of stages of computation interleaved with stages of communication that can be implemented with synchronous communication should port to this kind of platform. One may argue that few applications fall into this category, but notice that one could have come to the same conclusion for the domain of dense matrix computations had one started with ScaLAPACK. Thus, the real story is that by building on prior art like ScaLAPACK and PLAPACK, we managed to effectively layer a new library for the domain of dense matrix computations that had this desired property. Similarly, there are likely other domains that can be recast in such a way. The point is that the arrival of many-core architectures is an opportunity to reexamine and rearchitect existing software.

6.2 Future Directions

Very little effort has been made to optimize the Elemental port to SCC. We would especially like to optimize the bottom layer of Figure 2 to improve perfromance. Some of the collective communication routines have opportunities for optimization. For example, they view the process grid as a linear array of processes, which it is not. Furthermore, the sequential MKL libary is unoptimized for SCC. It should be updated to take advantage of the L1 and L2 caches of the Pentium processor to improve the base performance for single-core computation. Lastly, the packing operations of Elemental copy data into contiguous memory buffers to call RCCE communication routines, which subsequently copy data from those buffers to the MPB in roughly 8 KB chunks. By breaking this boundary to provide communication routines that skip this intermediate copy, we can substantially reduce the overhead cost seen in the component graphs above.

More generally, it is unwise to bet on only one solution, given the uncertainty of future architectures. As part of the FLAME project, a number of solutions have been developed for parallelizing dense linear algebra libraries. First, the sequential `libflame` library can be linked to multithreaded BLAS. Second, computation is mapped to a directed acyclic graph (DAG) where the nodes are operations on submatrix blocks (tasks) and the edges are data dependencies between tasks, and a runtime scheduler called SuperMatrix schedules tasks from the resulting DAG in parallel [22]. The Elemental library is the third solution.

The SuperMatrix scheduler is being ported to the SCC architecture by the authors. For this solution, the cores are viewed as threads on a multithreaded architecture, and data is shared via off-chip shared memory. The on-chip communication buffers are only used to manage the queue of tasks that are ready to execute. Programming constructs that supplement RCCE and allow memory to be viewed more like a shared adress space are being developed in support of this effort. A second solution views the cores as a distributed-memory architecture and uses message passing to implement the SuperMatrix scheduler and the passing of blocks of data [18]. Interestingly, the second approach, which uses message passing and the RCCE communication library, again yielded a relatively simple port. In the future, we intend to compare these alternative approaches to the one presented in this paper.

Additional Information

For additional information on the Formal Linear Algebra Methods Environment (FLAME), visit

`http://www.cs.utexas.edu/users/flame/`.

For further information on Elemental, visit

`http://code.google.com/p/elemental/`.

7 Acknowledgments

We thank the other members of the FLAME team for their support. We also like to thank Jesper Larsson Traff from the University of Vienna for his help with the implementation of alltoall and Antoine Petitet from the University of Tennessee for his help with the HPL port.

This research is funded by Intel Corporation along with access to an SCC processor. Bryan Marker is funded by a Sandia National Laboratories fellowship, and Jack Poulson is funded by a Computational Applied Mathematics fellowship from The University of Texas at Austin.

References

[1] www.openmp.org.

[2] D. Anderson and T. Shanley. *Pentium processor system architecture*. Addison Wesley, 1995.

[3] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[4] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[5] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 19:1749–1783, September 2007.

[7] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. Optimizing power using transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):12–31, January 1995.

[8] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[9] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[11] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[12] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[13] Jack J. Dongarra, Robert A. van de Geijn, and R. Clint Whaley. Two dimensional basic linear algebra communication subprograms. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[15] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[16] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[17] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Van al, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core ia-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC '10: Proceedings of the International SolidState Circuits Conference*, 2010.

[18] Francisco D. Igual and Gregorio Quintana-Ortí. Solving linear algebra problems on distributed-memory computers using serial codes. FLAME Working Note #48 DICC 2010-07-01, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores, July 2010.

[19] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[20] Timothy G. Mattson, Rob F. Van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Van al, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC processor: The programmer's view. In *SC'10: Proceedings of the 2010 ACM/IEEE Conference on Supercomputing*, New Orleans, LA, USA, 2010.

[21] Jack Poulson, Bryan Marker, Jeff R. Hammond, Nichols A. Romero, and Robert van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. submitted.

[22] Gregorio Quintana-Orti, Enrique S. Quintana-Orti, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3), 2009.

[23] Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication (research note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 799–804, London, UK, 2002. Springer-Verlag.

[24] R. Schreiber. Scalability of sparse direct solvers. *Graph Theory and Sparse Matrix Computations*, 56, 1992.

[25] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[26] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[27] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.

[28] R. Van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *ACM Operating Systems Review*, 2011. in press.

[29] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.