Scheduling Algorithms-by-blocks on Small Clusters

Francisco D. Igual¹, Gregorio Quintana-Ortí¹∗ and Robert van de Geijn²

¹ {figual,gquintan}@icc.uji.es
² rvdg@cs.utexas.edu.
Department of Computer Science, The University of Texas at Austin. Austin, TX.

SUMMARY
The arrival of multicore architectures has generated an interest in reformulating dense matrix computations as algorithms-by-blocks, where submatrices are units of data and computations with those blocks are units of computation. Rather than directly executing such an algorithm, a directed acyclic graph (DAG) is generated at runtime that is then scheduled by a runtime system like SuperMatrix. The benefit is a clear separation of concerns between the library and heuristics for scheduling. In this paper we show that this approach can be taken one step further, using the same methodology and an ad-hoc runtime to map algorithms-by-blocks to small clusters. The benefit is that with no change to the library code, and the application that uses it, the computational power of such small clusters can be utilized. Impressing performance on a number of small clusters is reported. We believe this to be a possible step towards programming many-core architectures, as demonstrated by a port of the solution to Intel’s Single-chip Cloud Computer (SCC).

1. Introduction
This paper is motivated by two insights: First, users often wish to perform dense matrix computations for problem sizes that are too small to warrant a large distributed memory architecture (cluster) yet too large for a current multicore architecture, or it may simply be the case that a small cluster is the most cost-effective solution. Also, multicore, where a few cores are available on a single chip, is becoming many-core, with a large number of conventional cores a single chip. Such many-core architectures may end up looking like distributed memory
architectures on a chip [17]. Second, especially on many-core architectures, it is highly likely that even if the cores are homogeneous they cannot be assumed to execute at equal clock speeds for example if “hot spots” on the chip require some cores to clock down. It would be convenient for a user to develop an application for multicore with the understanding that his/her solution can easily migrate to a distributed memory environment. We address these observations by extending the SuperMatrix runtime system [8, 21] developed as part of the FLAME project [11] to distributed memory environments with a relatively small aggregate number of cores.

There are a number of dense matrix libraries for distributed memory architectures. These include ScaLAPACK [2], a distributed memory extension of LAPACK, and PLAPACK [23], a package with similar functionality but that focuses more on abstractions that simplify development and maintainance. More recently, a new C++ library called Elemental was released that takes the insights from the PLAPACK project into the 21st century. What all three of these packages have in common is that an application needs to explicitly take distributed memory into account in order to utilize them. While this may be worthwhile for an application that targets high-end distributed memory architectures, the effort is considerable for those who just need a little extra performance.

A number of recent efforts have demonstrated that on multicore architectures it is convenient to formulate dense matrix computations as algorithms-by-blocks, viewing matrices as a collection of submatrices (blocks) [16]. Insights from architecture like out-of-order execution of scalar operation can then be applied to these algorithms in software by viewing blocks as units of data and computations with blocks as units of computation. Dependencies between computations can be easily identified at runtime by examining whether blocks are input and/or output to a computation. A directed acyclic graph (DAG) is constructed and a runtime system determines operations that are ready for execution and schedules these to cores using heuristics that take load-balance and locality of data into account. As part of the FLAME project, the sequential libflame library [24] code needs not change at all and the runtime scheduling is handled by the SuperMatrix runtime system [8, 21]. As part of the PLASMA/MAGMA/Quark project a similar effort is being pursued to extend LAPACK [6, 5]. We need better references for their junk. It differs in that the code does not resemble the original sequential code and there does not seem to be as clean a separation of concerns between the library code and the runtime system.

This paper extends libflame and the SuperMatrix runtime system so that it can target distributed memory architectures. Preliminary work in that direction was reported in [14]. This paper gives a more thorough treatment of the approach, offers new experimental results on a wide range of architectures (including shared and distributed-memory machines equipped with different interconnection networks) proposes and evaluates a new technique to transparently overlap communication and computation in the framework of linear algebra computations, and reports on a port of the developed runtime to a novel, experimental many-core architecture: the Intel SCC processor [13]. It should be noted that as part of the PLASMA project a similar effort targets massively parallel distributed memory architectures [3, 4]. In contrast, we believe the natural target of our approach is small clusters and focus heavily on providing a convenient path for the application.
The rest of the paper is organized as follows. Section 2 describes the main concepts that underlie the FLAME methodology for the automatic parallelization of sequential algorithms. Section 3 introduces the runtime-based approach to map sequential applications to parallel architectures. Section 4 describes in detail a basic runtime implementation targeting distributed-memory architectures, and successive improvements to improve performance. Section 5 shows the experimental study on several computers, including the Intel SCC chip. Finally, Section 6 summarizes concluding remarks and future work.

2. Preparation: Algorithms-by-blocks

In this section, we review how as part of the FLAME project sequential algorithms are mapped to multithreaded architectures (e.g., multicore and SMP architectures). As we often do, we will use Cholesky factorization as a representative operation for our explanation.

2.1. Algorithms and serial code

Given symmetric positive-definite matrix $A \in \mathbb{R}^{n \times n}$, its Cholesky factorization is given by $A = LL^T$, where $L$ is lower triangular. Unblocked and blocked algorithms that overwrite $A$...
with $L$, presented in FLAME notation, are given in Figure 1. We often present a number of algorithms for this operation. However, as we will see, in the approach discussed in this paper this is not a relevant issue. Using the FLAME/C API, the algorithms in Figure 1 are easily translated to C code, where the C code closely resembles the algorithms, given in Figure 2.

### 2.2. Algorithm-by-blocks

A key insight behind the FLAME project's approach to parallelization for multicore architectures has been the idea of an algorithm-by-blocks. While the algorithm in Figure 1 is referred to as a blocked algorithm, an algorithm-by-blocks performs essentially the unblocked algorithm on the left of Figure 1, except that each element in the matrix becomes a block (submatrix). We will take these blocks to be square, except possible for the ones on the edge of the matrix.

Figure 2. FLAME/C implementations of the unblocked (left) and blocked (right) algorithms for the right-looking algorithmic variant of the Cholesky factorization.
Let us consider the case where $A$ is partitioned into blocks and consider the state of the matrix after the first iteration:

\[
\begin{pmatrix}
L_{00} & * & * & * \\
L_{10} & A_{11}^{(1)} & * & * \\
L_{20} & A_{21}^{(1)} & A_{22}^{(1)} & * \\
L_{30} & A_{31}^{(1)} & A_{32}^{(1)} & A_{33}^{(1)}
\end{pmatrix}.
\]

In the next iteration, $A_{11} := L_{11} = \text{Chol}(A_{11}^{(1)})$, $A_{i1} := L_{i1} = A_{i1}L_{11}^{-H}$, and $A_{ij}^{(2)} := A_{ij}^{(1)} - L_{ij}L_{ii}^H$, for $2 \leq i \leq 3$ and $i \leq j \leq 3$. In other words, if the matrix is stored by blocks, operations with blocks are to be performed in each iteration.

### 2.3. Other operations

As part of the FLAME project, an API (FLASH [16]) was developed so that the code that implements an algorithm-by-blocks is essentially identical to the code in Figure 2 (right). Algorithm-by-blocks for all level-3 BLAS and many important operations in LAPACK and libflame have so been implemented. For example, we have implemented incremental QR factorization [12], an operation that we will also use in our performance section.

### 3. Separation of Concerns: A Runtime-based Approach to Parallelization

In a number of papers, we introduced the idea of having a sequential library, libflame, automatically parallelize operations targeting multi-threaded architectures (SMP and/or multicore) [8] and/or multi-accelerator (multi-GPU) architectures [20] and even solving problems with data stored in disk [19].

The idea is to view blocks in the algorithm-by-blocks as a unit of data and operations with those blocks as units of computation (tasks). Instead of executing operations with blocks, the tasks are enqueued, together with their dependencies (a phase we call the examiner stage) thus forming a directed a-cyclic graph (DAG) of tasks. These tasks are then scheduled to cores via a runtime system which we call SuperMatrix. Figure 3 shows the traditional approach to serial programming (left), the traditional approach to parallel programming (center), and our new approach to program distributed-memory machines with serial codes by using a runtime (right).

We are not alone in this approach: The PLASMA and MAGMA projects [1] pursue a similar approach. More recently, these have culminated in the Quark runtime system [15], which uses dynamic scheduling much like our SuperMatrix system has from its inception [7]. What makes our approach different is our focus on programmability, ensuring that the library that encodes the algorithms is completely separate from the runtime system that executes tasks, so that neither the application programmer nor the library developer needs not know about the parallelization, target architecture, or scheduling heuristics.

In this section, we explain how to easily port the FLASH/SuperMatrix approach to distributed memory architectures. The emphasis is on creating a simple and high performing...
implementation that allows an application to be easily ported from a multi-threaded environment to a small cluster. For this, we are willing to give up on scalability: this solution should not be used for large clusters.

3.1. Data distribution

Key to computing on distributed-memory architectures is the distribution of data so as to improve load balance while reducing communication local memory use. In traditional
distributed-memory libraries for dense linear algebra like ScaLAPACK [9], PLAPACK [23], and (more recently) Elemental [18], a 2D cyclic data distribution is commonly used, since it provides a more scalable solution. The idea is to view $p$ processes as a $r \times c$ mesh of processes. Each process is, for example, an MPI process and for our approach one would have one process per multicore node and obtain parallelism within that node via the runtime system that schedules task. More on this later. Partition an operand into square submatrices (blocks again)

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & \cdots \\ A_{1,0} & A_{1,1} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix},$$

where $A_{i,j}$ is $b \times b$ (except possibly for matrices on the right and bottom). A 2D block cyclic data distribution then assigns $A_{i,j}$ to process $(i \mod r, j \mod c)$, as illustrated in Figure 4.

Our approach borrows this cyclic distribution but in contrast to libraries like ScaLAPACK, PLAPACK, and Elemental, where the local matrix itself becomes a traditional matrix stored in column-major order, our blocks in the sequential algorithm are already stored as a matrix of blocks and hence only the placement of those blocks needs to be decided.

Now, FLASH already creates a matrix of matrix blocks (possibly hierarchically, but that is not exploited in this work). In other words, the top-level matrix is a matrix of descriptors (think: a matrix where each element describes a submatrix including a pointer to its data). Our new approach distributes duplicates the entire top level matrix to each of the processes in the $r \times c$ mesh. The descriptor for a block that a process owns (according to the 2D cyclic mapping) includes the data for that block. A descriptor for a block that a process doesn’t own consists of the information that describes the block, but does not include the actual data for the block. Thus, there is a duplication of the top-level matrix (of descriptors) to all processes, but a process only stores the part of the matrix that it owns.

Now, the top-level matrix locally on each processor requires $O(n^2/b^2)$ storage for the descriptors. The storage of the local matrix requires approximately $O(n^2/p)$ storage. If $n$ is large, as long as $b$ is reasonably large and $p$ reasonably small, the storage of the top-level matrix is not a limiting factor.
3.2. The examiner stage

In the FLASH/SuperMatrix approach, the library routine first gets executed. These algorithms perform operations with blocks. On a sequential architecture, these operations with blocks are simply calls to routines that execute at the time the task is encountered in the code. On a multithreaded architecture, this instead creates a DAG of tasks with dependencies that are then executed during the executor stage.

On distributed-memory machines the list of tasks is simply generated redundantly by every process. Since this only requires the information in the top-level matrix, no change is needed to the code that performs this function. An example of DAG for the Cholesky factorization of a matrix partitioned in $4 \times 4$ blocks is shown in Figure 5. In our approach, all processes will have a copy of this DAG.

Notice that a typical matrix operation with an $n \times n$ matrix requires $O(n^3)$ computation. We now execute an algorithm with $b \times b$ matrix blocks, generating $O((n/b)^3) = O(n^3/b^3)$ tasks. Thus, the storage required for the entire DAG is $O(n^3/b^3)$, larger than the $O(n^2/b^2)$ required for redundantly storing the top-level matrix. Thus, it is not the redundant storage of
the top-level matrix that is the major storage concern. Again, if \( b \) is large enough and \( p \) small enough, it is not a concern, period.

### 3.3. The executor stage

It is in how the DAG is now scheduled that the difference lies. This means that only the mechanics of the executor (scheduler) needs to be changed inside the SuperMatrix runtime system. Whereas in previous approaches (multi-core architectures, multi-GPU architectures, and solving problems with data stored in disk) the DAG and its task are scheduled by one thread, on distributed-memory machines this is redundantly performed be every process. each task is processed by every process. In this case, one process will do the processing and some other processes may do some communications in case they are required.

The first question is where to execute a task. From experience with other architectures, e.g. multi-GPUS [20], we know that designating the owner of a given task to be the first owner of the first input/output operand (block in the matrix) yields results. Hence we adopt this rule in our current implementation.

The second question is when to execute a task and how to get the data to its owner. The approach is rather simple: The tasks, one by one, are inspected redundantly by all processes, following the ordering dictated by the sequential order of the serial program, and captured in the list of tasks. Then, for each task, a process takes action depending on whether it owns the task:

- If a given process is the owner, it will execute the task. If the data is already locally available, then the task is executed and the DAG updated upon completion with information regarding which tasks are ready for execution next. If the data is not locally available, it will acquire it from the owner.
- If a given process is not the owner, it will examine whether it owns any input data for the task. If so, the data is sent to the owner of the task. If the sent operand is also an output of the task, it will receive the modified result back. If the task does not contain any operand belonging to the process, it will just skip the task.

When all processes have processed all tasks, the program finishes, and the results have been obtained.

The only information that is required to orchestrate communications between tasks is the knowledge about tasks being input, output, or input/output operands.

### 3.4. An example

Let us illustrate the previous descriptions with an example. Let us suppose we want to compute the right-looking variant of the Cholesky factorization of a matrix of \( 4 \times 4 \) blocks on a \( 2 \times 2 \) mesh of processes. Consider a 2D block data distribution, as was illustrated in Figure 4. Thus, process 0 (\( P_{00} \)) will own blocks \( A_{00}, A_{02}, A_{20}, \) and \( A_{22} \); \( P_{10} \) will own blocks \( A_{10}, A_{12}, A_{30}, \) and \( A_{32} \); \( P_{01} \) will own blocks \( A_{01}, A_{03}, A_{21}, \) and \( A_{23} \); and \( P_{11} \) will own blocks \( A_{11}, A_{13}, A_{31}, \) and \( A_{33} \).

Every process will execute the serial algorithm to generate the list of tasks shown in Figure 6. Once the list is generated in every process, all of them start processing the full list, task by task.
At the very beginning, all processes examine the first task: $A_{00} := \text{Chol}(A_{00})$. $P_{00}$ checks that the task belongs to it and the required operand resides in its memory. Therefore, $P_{00}$ starts computing the Cholesky factorization of $A_{00}$. At the same time, the rest of the processes check that the task does not belong to them, and no operand in it belongs to them. Therefore, they skip it and start examining Task 2.

While $P_{00}$ is factorizing block $A_{00}$, the rest of the processes start analyzing Task 2. Process $P_{10}$ must execute Task 2, as it owns its output operand, $A_{10}$. This process analyzes the status of the operands of this task. It owns $A_{10}$ but the other block it needs, $A_{00}$, belongs to $P_{00}$ and it must wait for it to arrive. Processes $P_{01}$ and $P_{11}$ are not involved at all with this task and skip it, jumping to the next task.

When $P_{00}$ finishes factorizing block $A_{00}$, it moves on to Task 2. When examining that task, it checks that it owns block $A_{00}$, which is needed by Task 2, and concludes it must send that block to $P_{10}$, the owner of the task. Once sent, process $P_{10}$, which was waiting for it, receives it and commences with the execution of Task 2.

The process continues in this fashion with all tasks in the list.

4. Variations on a Theme: Implementation of the Runtime System

The last section emphasizes that the changes to the FLASH/SuperMatrix approach to generating and scheduling algorithm-by-blocks are completely isolated in the executor part
of the system, the runtime system. As a result, we can view the description of the runtime system in the last section as a simple first approach. In this section, we describe it in more detail, as well as two variations on the theme. For now, assume that MPI [22] is used to carry out communications between processes.

4.1. Basic runtime

The first version of the runtime is rather simple simple. For each task, all blocks not owned by a task owner (alien blocks) are transferred. Input alien blocks are only sent by their owners before executing the task, output alien blocks are received by their owners after executing the task, and input/output alien blocks are sent and then received by their owners. Clearly, the same block with same contents may be transferred between processes multiple times.

4.2. Runtime with cache

A more sophisticated runtime tries to reduce the number of transfers between processes by using a cache of data blocks in every node to store the most recently used alien blocks. Before one block is sent by its process owner, both the block owner and task owner check whether it is already located in the cache of the task owner. If this is the case, no transfer is done.

Consider, as an illustrative example of the benefits of the cache system, the execution of tasks numbered as Task 2 and Task 4 in Figure 6. Both tasks are executed consecutively by the same process (P10). In addition, both tasks have a common input alien parameter A00. Thus, in the basic runtime, prior to the execution of Task 4, A00 must be transferred from its owner (P00), although an identical transfer has been performed prior to the execution of Task 2. With the introduction of a software cache, this transfer can be avoided, as operand A00 is still stored in the cache bound to process P10.

In the experiments reported in the next section, we employed a four-set associative cache with 32 blocks. This technique reduced the number of transfers between processes considerably. The total amount of blocks in cache memory, and the degree of associativity can be easily tuned in the developed runtime.

4.3. Runtime with cache and overlapping of computations and communications

The performance of the previous runtime can be improved by creating two concurrent threads per process. One thread will perform all the communications and the other thread will perform all the computations. The communication thread will process tasks in advance, that is, performing all the communications required. Once all the operands of one task are ready (have been transferred if owned by other processes), the task is sent to the computation thread to be executed.

In this way, communications and computations can be overlapped, thus improving performances.
4.4. Coding effort

One way to compare the programmability effort of the proposed approach versus that of, say, LAPACK or ScaLAPACK is to compare the number of lines of code.

The Cholesky factorization in LAPACK (serial library) is 141 line long (considering routines dpotrf and dpotf2, and excluding blank lines and comments). The Cholesky factorization in ScaLAPACK (distributed-memory library) is 270 line long (same routines and excluding comments). While the former is a serial code, the latter is a concurrent code, and thus more complex.

In our approach, the Cholesky factorization is about 70 lines of serial libflame code. In parallelizing it, none of these lines needed to be changed. While it is true that the runtime system had to be modified, this was itself a relatively straightforward task. The most complex version of the runtime (including cache management and overlapping of computations and communications) is roughly 1000 lines long. Importantly, those changes potentially allowed almost the entire libflame library to be parallelized for distributed memory architectures.

I wonder if we should have a section “Relation to SuperMatrix runtimes for other architectures” in which you describe similarly customizations for, say, GPU and/or OOC computation.

5. Experimental results

In this section, we report experimental results on different platforms. We illustrate the benefits of the more sophisticated runtime system implementations and compare against a distributed memory library, ScaLAPACK.

5.1. Setup

In recent reviews, referees complained that names for the systems are not very helpful when trying to look at graphs. Tesla, in particular, is confusing because of the NVIDIA use of that name.

We tested both ScaLAPACK codes and our codes on three very different parallel platforms, in an attempt to provide an extensive evaluation of a wide range of systems, and to illustrate the flexibility of the solution:

- **RA** is a distributed-memory machine equipped with Intel Xeon processors. Each node is a 32-bit Intel Xeon at 2.4 GHz with 512 MB of RAM. The peak speed of each processor is 4.8 GFlops (10^9 flops per second). The interconnection network is a Fast Ethernet (with a peak bandwidth of 100 Megabits/s). Therefore, this machine consists of slow processors interconnected with a very slow network.

- **TESLA2** is a shared-memory machine with two quad-cores Intel Xeon E5440 at 2.83 GHz (8 cores in total). The peak speed of each core is 22.6 GFlops. All communications are performed through the shared memory. Therefore, this machine consists of fast processors interconnected with a very fast network (shared memory).
PECO is a distributed-memory machine with 4 nodes. Each node has one Intel Xeon E5520 at 2.27 GHz (4 cores in total per node). The peak speed of each core is 18.2 GFlops. The interconnection network is a InfiniBand with a peak speed of 40 Gigabits/s. Characterize this machine like you did the others. Fast processors, medium network?

We employed GNU compilers, the GotoBLAS library which version, the MPICH2 implementation of the MPI standard (MVAPICH2 release ...for the Infiniband network in PECO), and libflame (Release r3979).

We tested all the implementations for 4, and 8, 9, or 16 processors/cores, as available. For both ScaLAPACK and our approach we tested most configurations. On 4 processors we tested the following mesh configurations: $1 \times 4$, $2 \times 2$, and $4 \times 1$. On 8 processors we tested the following mesh configurations: $1 \times 8$, $2 \times 4$, $4 \times 2$, and $8 \times 1$. On 9 processors we tested the following mesh configurations: $1 \times 9$, $3 \times 3$, and $9 \times 1$. On 16 processors we tested the following mesh configurations: $1 \times 16$, $2 \times 8$, $4 \times 4$, $8 \times 2$, and $16 \times 1$. Only results for the best mesh configurations in each platform are shown in the figures.

Block sizes employed in ScaLAPACK implementations and in our new codes were the following: 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, and 384. No other block sizes appear to be better than those. Only results for best block sizes are shown in the figures.

The figures report performance in GFLOPS ($10^9$ flops per second) as a function of matrix size. In all figures, DRT stands for Distributed RunTime and refers to our serial algorithms linked to an implementation of the described distributed memory runtime system.

5.2. Results

Figure 7 compares the three runtime system implementations described in the previous section when computing the Cholesky factorization. As expected, the basic implementation...
delivers the worst performance due to its higher communication cost. Adding a software cache usually offers the best performances. The reduction in the number of data transfers is the main reason underlying this performance improvement. Adding both a software cache and computation/communication overlapping yields the best performance for some mesh configurations and large matrix sizes.

In Figures 8 and 9 we compare the best implementation of the runtime system with ScaLAPACK for the Cholesky factorization and QR factorization, respectively. In the case of QR factorization a variant on that operation that performs an algorithm-by-blocks is used for our implementation while ScaLAPACK implements the classic blocked algorithm. These graphs show that performance of our new codes is competitive with ScaLAPACK. We believe the above results to be representative of performance that would be attained by other linear algebra operations.

Note that we are not out to outperform ScaLAPACK: the important thing is that the coding effort for our approach is much less.

5.3. Experimental results on novel architectures. The SCC chip

As a demonstration of the flexibility of our approach, we adapted it to a novel architecture: the Intel Single-chip Cloud Computer (SCC) research processor [13]. For that effort, rather than focusing on performance, our aim was to illustrate how runtime-based systems are an appealing solution not only for current distributed-memory architectures, but also to future many-core processors.

The Intel SCC is an experimental research processor designed to explore future trends in terms of hardware (power consumption management, many-core integration, network-on-chip architectures, etc.) and software. From the software perspective, it can be seen as a platform to explore the programming techniques and models that will fit better to future many-core architectures.

An overview of the SCC architecture is shown in Figure 10. The processor consists of 48 cores, organized in a mesh of 4 × 6 tiles. Each tile groups two P54c (Intel Pentium) processors, each with independent L1 and L2 caches. The tile includes a small amount (16 KBytes) of SRAM seen as a shared memory space by all the cores in the chip, usually referred as MPB (Message-Passing Buffer). Each tile includes a router; the on-chip interconnection network is linked to four DDR3 memory controllers supporting up to 64 GBytes of DRAM memory. When considering the processor as a distributed-memory architecture, the programmer views this memory divided as separate private memory spaces to each core in the processor. There is no support for cache coherence between cores for all memory spaces in the architecture. Therefore, it is the software writing by a programmer that must explicitly manage data coherence and consistency explicitly; this type of features allow the usage of novel programming techniques and strategies that are usually difficult to apply in commercial processors.

Considering the SCC processor as a “cluster on chip”, it can be seen as a purely message-passing architecture. To support this programming paradigm, Intel supplies the RCCE [17, 10] interface, an experimental light-weight communication library for the SCC processor. RCCE provides blocking point-to-point communication primitives, with semantics similar to those offered by the MPI standard [22]. Provided that our runtime only uses blocking point-to-
Figure 8. Performance the best runtime system implementation and ScaLAPACK for the Cholesky factorization on three different systems. Only results for best mesh configurations and best block sizes are shown.
Figure 9. Performance the best runtime system implementation and ScaLAPACK for the QR factorization on three different systems. Only results for best mesh configurations and best block sizes are shown.
point communications, the port of existing message-passing application to the SCC chip is straightforward.

More specifically, the porting of our runtime system to the SCC chip required less than one hour to adapt it to the new infrastructure. Furthermore, no changes were required in the user’s code to run them on the SCC processor. The abstraction of the underlying architecture provided by the developed runtime system makes it possible to apply further optimizations to the system without changing a single line of the thoroughly tested, serial libflame codes.

As an illustrative example, we show results for the right-looking variant of the Cholesky factorization on the SCC chip. Figure 11 reports the experimental results for an increasing number of cores (from 1 to 48) in terms of raw performance (left-hand side plot) and speedup (right-hand side plot). Results correspond to the runtime version including cache. No overlapping was implemented due to the lack of asynchronous point-to-point communications in RCCE or hyprethreading support at core level in the SCC chip.

We were not able to compare the performances of our approach with those of ScaLAPACK since it had not been ported to the SCC chip, since there was not yet an MPI implementation for the SCC chip and considerable changes would need to be made to that library in order to only use blocking communications. However, experimental results show a reasonable scalability in our solution, considering that point-to-point blocking communication patterns are exclusively used. We note that this processor was not built for high performance. No conclusions about performance should therefore be drawn.

Few other works have gained insights into the port of distributed-memory dense linear algebra implementations to many-core chips like the Intel SCC. Recently, a port of the full Elemental library was proposed in the framework of the FLAME project We need
Figure 11. Performances of our new codes (DRT) for the Cholesky factorization on the SCC chip. Only results for best mesh configurations and best block-sizes are shown.

A reference here. Although the target functionality of both approaches is similar, the philosophy behind both frameworks is essentially different. One of the main benefits of our runtime-based system is the total absence of explicit communications in the user codes. From this perspective, user codes are merely sequential, requiring no modifications at all to run on distributed-memory architectures like the SCC chip. The development of new routines in Elemental requires a careful design of the communication patterns for each routine. This complexity in the development process of new implementations offers further benefits from the performance perspective, due to the possibility of applying optimizations that are hard to automatically apply on runtime-based systems. We conceive both systems as different (but complementary complementary) solutions to solve the programmability and performance problem present in novel many-core architectures like the Intel SCC chip.

6. Conclusions

We have demonstrated how, with relatively small effort, algorithm-by-blocks in conjunction with a runtime system can retarget sequential libflame code to distributed memory architectures. On small to medium sized clusters, the performance is comparable to that of ScaLAPACK. The building of traditional distributed-memory libraries took many man-years, while the effort to port a runtime system that targeted multithreaded architectures to a cluster using our approach requires an effort measured in man-days. That effort is amortized over the functionality of much of the libflame library.

Our approach is of special appeal given the increasing interest in porting existing applications for which current single shared-memory machines are not enough from the memory perspective.
and small clusters become the only realistic solution. By relying on a runtime system, this transition becomes straightforward for the application programmer. In addition, this work can give insight into how processors with very large number of cores, like the SCC processor, may be programmed in the future. Runtime systems seem the natural way to attain high performance in future many-core architectures, with minor impact from the programmer’s perspective.

Once the runtime has been developed, porting any other variants (e.g., left-looking, bordered, etc.) of the discussed factorizations and operations not previously included in the library is much easier than for traditional libraries.

Future research lines are focused on rewriting the runtime to increase its scalability. Though a rewriting of some aspects of the runtime will be needed, the clear and main advantage is that the basic serial algorithms will still not have to be modified at all. An additional interesting improvement is to use macro-blocks (or aggregations of blocks) to allow the execution of factorizations where a panel of columns needs to be factored (e.g., traditional LU factorization).

Acknowledgements

This research is sponsored by Microsoft Corporation. We thank Intel for granting access to the SCC platform. Francisco Igual is supported by a research fellowship from the University Jaume I (PREDOC/2006/02).

REFERENCES


