Solving Linear Algebra Problems on Distributed-Memory Computers using Serial Codes

Francisco D. Igual Gregorio Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
{figual,gquintan}@icc.uji.es

Technical Report DICC 2010-07-01

Published: July 31, 2010

Abstract

Programming on distributed-memory architectures is a complex task. Designing and developing the solution to a problem on a distributed-memory multicomputer requires an effort much higher than that of working on serial computers. Accumulated experience and new programming methodologies do not reduce the burden of this task. In this paper we propose a much easier method to generate programs for distributed-memory machines: we propose to make usual serial algorithms run above a run-time that carries out all the communications. This solution addresses the increasing necessity of porting problems that are to large for a single shared-memory machine, but do not need the potential of a huge cluster, without the necessity of dramatic changes in existing codes. In fact, our approach does not imply any change on existing, thoroughly-tested serial codes. The experimental results show that performances obtained by our new proposal are competitive with the top libraries on small distributed-memory architectures.

1 Introduction

Programming distributed-memory architectures is a very complex task even for experienced programmers. The difficulty has not been reduced despite the many resources and years spent on programming methodologies.

There exist some libraries for solving linear algebra problems on distributed-memory machines [3, 12]. Despite these projects started many years ago, they contain only a subset of the problems solved in usual libraries for serial computers, mainly because of the intrinsic difficulty of programming on those target architectures.

Recently, this programmability problem is becoming more important. Processor manufacturers are packing more and more cores in a single chip in an almost endless race. Though a limit of the number of cores in a single chip has not been defined, there exists a limit to the number of cores that a memory system can support. When that limit is reached, distributed-memory machines will be the main (and only) option. As the programming of those architectures is all but easy, this problem will have to be addressed really soon, when the number of cores runs into the limit the

memory can handle. Recent architectures like the Intel SCC [8] processor might be reaching that limit.

Serial algorithms for solving linear algebra problems have been designed and developed from the very beginning of computer science. Goldstine and von Neumann in the 1950s [1] wrote codes for computing the gaussian elimination, the inverse, and the eigenvalues of a matrix on their computer, one of the first ones with electronic technology and stored program. They were able to solve all those problems writing their codes in just machine language.

Many years afterwards, programming methodologies seem to have improved a lot: code reuse seems to be one of the main achievements of modern programming methodology.

In this paper we propose to employ code reuse. We follow a different methodology to traditional one: Use of usual serial codes running onto a run-time that takes care of all the communication and low-level tasks. Hence, usual serial (and thoroughly tested) algorithms and codes are employed without any modification, and thus programming difficulty is greatly reduced. We studied two very different factorizations that are representative of dense matrix computations: Cholesky and QR factorizations. The experimental study shows that our new approach gets performances similar to those of traditional methods on small-size architectures. We think this approach may be interesting when programming traditional and new distributed-memory architectures, such as the SCC processor.

The method of using serial algorithms and an underlying run-time has been successfully employed in the programming of multi-core and many-core architectures [6, 11], the programming of architectures with hardware accelerators [10], and the programming of out-of-core problems (large matrices stored in disk) [9]. A similar approach has been adopted by the PLASMA project [5, 4].

In this work we propose to extend this technique to distributed-memory architectures, so that its programming can be made easier. Problems that are too large for a single shared-memory machine but do not need the full potential of a huge distributed-memory cluster are becoming more and more popular nowadays. Our approach is of wide appealing for these type of problems, and reduces the programming effort to the minimum to adapt existing serial codes to these particular architectures.

The rest of the paper is organized as follows. Section 2 contains a brief description of the main libraries for programming distributed-memory computers. Section 3 describes our new approach to programming distributed-memory machines with serial codes. Section 4 shows the experimental study on several computers. Finally, Section 5 summarizes concluding remarks and future work.

2 Linear Algebra Libraries for Distributed-Memory Computers

A few libraries to solve linear algebra problems on distibuted-memory computers were designed and implemented years ago. Because of the programming effort invested is so much higher than when working on serial computers, the functionality of these libraries is not so complete: some interesting linear algebra problems have remained unimplemented for years despite its importance.

The building of ScaLAPACK started in 1990s, and most of it was written in Fortran-77. The design of this library tried to resemble LAPACK, but only a subset of LAPACK was implemented.

Comparing codes, barring blank lines and comments, of serial LAPACK and its counterpart for distributed-memory ScaLAPACK, Choleksy factorization in LAPACK is 141 line long, while in ScaLAPACK it is 270 line long. Codes for QR factorization consists of 108 lines in LAPACK, and 203 lines in ScaLAPACK. Roughly, ScaLAPACK is twice longer than LAPACK. However, the difficulty in programming ScaLAPACK is much higher, since ScaLAPACK requires concurrent programming with message-passing communications.

PLAPACK is a rather different approach. It is a library infrastructure for coding parallel linear algebra algorithms at a high level of abstraction on distributed-memory computers. This infrastructure allows programmers to exploit a natural approach to encoding so-called blocked algorithms, which achieve high performance by operating on submatrices and subvectors, withtout working with indices. However, concurrent programming is also required in this approach.

3 Our New Approach: Serial Algorithms on a Run-Time

In this section, we describe the traditional serial algorithms we used and the new run-times developed in our implementations.

3.1 Serial Algorithms

First, we briefly describe the serial algorithms used in our implementations. These serial algorithms using FLAME/C API are included in `libflame` library (it can be downloaded from `http://wwww.cs.utexas.edu/users/flame`). These algorithms have been thoroughly tested on different machines and even on different architectures.

3.1.1 Cholesky Factorization

Figure 1 shows the right-looking variants of both the unblocked and blocked algorithms for computing the Cholesky factorization of a symmetric positive definite matrix, specified using the FLAME notation. Those algorithms only update the lower triangular part of the initial matrix. Both of them are serial codes, but the blocked algorithm casts the bulk of the computation in terms of matrix-matrix products in order to exploit the multi-layered structure of the memory system by reusing data that are closer to the processor.

We have used the right-looking variant since it is usually the best one on parallel architectures. Note that we have used the *serial* right-looking algorithm for computing the Cholesky factorization.

FLAME includes a variety of application programming interfaces (APIs) that allow an easy transition from algorithm to code, reducing the possibility of introducing errors during this process.

We will use the blocked variant with no change at all, and we will run it above a run-time which captures all the calls.

Figure 2 shows the unblocked codes (left), and blocked codes (right) corresponding to the algorithmic right-looking variant of the Cholesky factorization of a lower triangular matrix using the FLAME/C API [2]. These codes are just an easy translation of algorithms in Figure 1.

We briefly show how the blocked algorithm works on a matrix A partitioned as:

$$A = \begin{pmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{pmatrix}.$$

At the beginning, A_{TL}, A_{TR}, and A_{BL} are empty, and A_{BR} contains all blocks (T_{ij}). When the first iteration of the loop starts, the repartitioning operation makes A_{11} become T_{00}, A_{12} become (T_{01}, T_{02}, T_{03}), A_{21} become $(T_{10}^T, T_{20}^T, T_{30}^T)^T$, and so on. After this repartitioning, the computing code starts. The first task to do is to compute the Cholesky factorization of A_{11}, which is T_{00} in this iteration. The second task to do is to compute a triangular system solve: $A_{21} := A_{21} A_{11}^{-T}$,

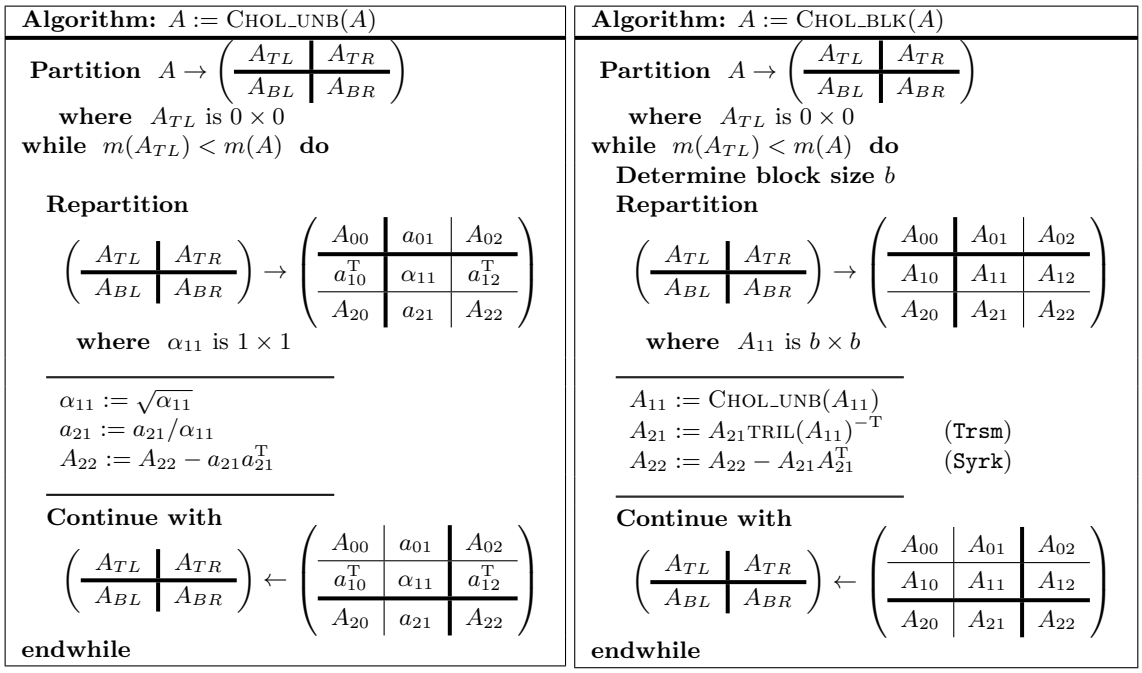

Figure 1: Rigth-looking variants of unblocked (left) and blocked (right) algorithms for computing the Cholesky factorization. In the notation, $m(A)$ stands for the number of rows of matrix A, and TRIL(A) denotes the matrix consisting of the elements in the lower triangular part of A.

which results in the processing of blocks : T_{10}, T_{20}, and T_{30}. The last computing operation of the iteration is: $A_{22} := A_{22} - A_{21}A_{21}^T$, which results in the processing of blocks: T_{11}, T_{12}, T_{13}, T_{21}, T_{22}, T_{23}, T_{31}, T_{32}, and T_{33}. The final repartitioning operation of this first iteration reorganizes blocks such that next iteration the algorithm will be processing the rest of the matrix. Figure 3 shows the shape of blocks after first partitioning (and just before the execution of computing operations) of first iteration, and the shape of blocks after first partitioning of second iteration. The figure shows how blocks (*views* in fact, that is, references to sub-matrices inside the original matrix) are shifted as the matrix is being processed. Second iteration will proceed in the same way as the first one, but in this case A_{11} will become T_{11}, etc. And so on.

3.1.2 QR Factorization

We have implemented incremental QR factorization since we already have serial thoroughly-tested codes and because our run-times do not allow macroblocks (aggregations of blocks) yet. See [7] for a detailed description of this factorization.

3.2 New Method

Many serial algorithms have been designed, developed, and intensively used in the field of linear algebra since the development of the first computers. Some of them have been implemented in many programming languages and have been thoroughly tested for decades.

In this paper we propose to use those usual serial algorithms running above an underlying runtime that studies the data dependencies between tasks, and then carries out the transfers needed

```
FLA_Error FLA_Chol_unb_var1( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00,  a01,     A02,
          ABL, ABR,      a10t, alpha11, a12t,
                         A20,  a21,     A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {

    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00,  /**/ &a01, &A02,
    /* ************* */   /* ************************** */
                            &a10t, /**/ &alpha11, &a12t,
       ABL, /**/ ABR,       &A20,  /**/ &a21, &A22,
       1, 1, FLA_BR );
    /*-----------------------------------------*/
    FLA_Sqrt( alpha11 );
    FLA_Inv_scal( alpha11, a21 );
    FLA_Syr( FLA_LOWER_TRIANGULAR,
             FLA_MINUS_ONE, a21
             FLA_ONE,       A22 );
    /*-----------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, a01,      /**/ A02,
                               a10t, alpha11, /**/ a12t,
    /* *************** */   /* ************************ */
       &ABL, /**/ &ABR,        A20, a21,      /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
FLA_Error FLA_Chol_blk_var1( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       b, b, FLA_BR );
    /*-----------------------------------------*/
    FLA_Chol_unb_var1( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11,
                       A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21,
              FLA_ONE,       A22 );
    /*-----------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figure 2: FLAME/C implementations of the unblocked (left) and blocked (left) algorithms for Variant 1 of the Cholesky factorization.

for the processes to perform the tasks. No change at all had to be made to the serial algorithms we previously described.

Figure 4 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 run-time (right).

In our approach, the serial algorithm does not actually execute the tasks, but just generates a list of tasks to be executed. This can be easily done by inserting a layer below the serial algorithms.

As we told above, the technique of using serial codes and a run-time has been successfully employed in programming multi-core architectures, multi-GPU architectures, and solving problems with data stored in disk.

However, the new run-time for distributed-memory machines is very different from the previous ones.

The generation of the list of tasks is something different: Whereas in previous approaches (multi-core architectures, multi-GPU architectures, and solving problems with data stored in disk) the list of tasks were generated by only one thread, on distributed-memory machines the list of tasks is generated by every process due to the distributed nature of the architecture. All processes in the system generate the list of tasks, concurrently. This is not a true handicap because its cost is similar to that when only one node generates the list of tasks. Then, once the list of tasks has been generated in all processes, the execution of the algorithm may start.

The way of handling the tasks and its dependencies is also completely different in this case. Whereas in previous approaches (multi-core architectures, multi-GPU architectures, and solving problems with data stored in disk) each task was processed by only one thread, on distributed-memory machines each task is processed by every process. In this case, one process will do the

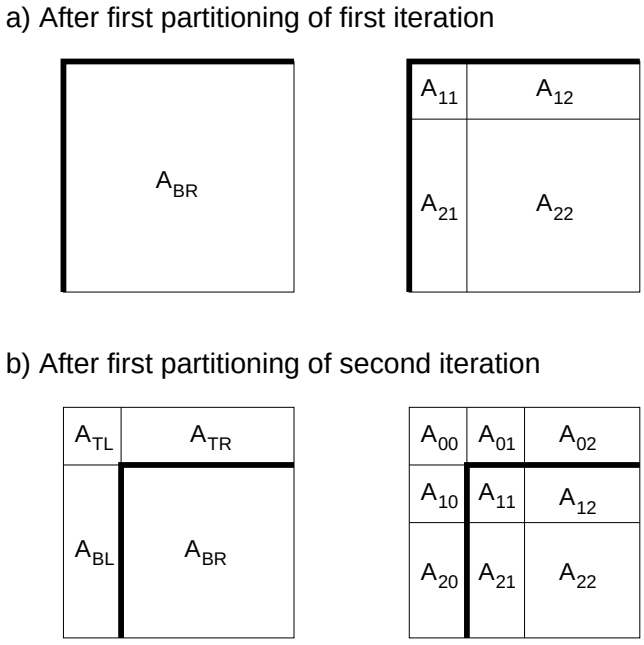

Figure 3: Top, shape of views of A after first partitioning of first iteration. Bottom, shape of views of A after first partitioning of second iteration. As it can be seen, blocks are shifted as the matrix is being processed.

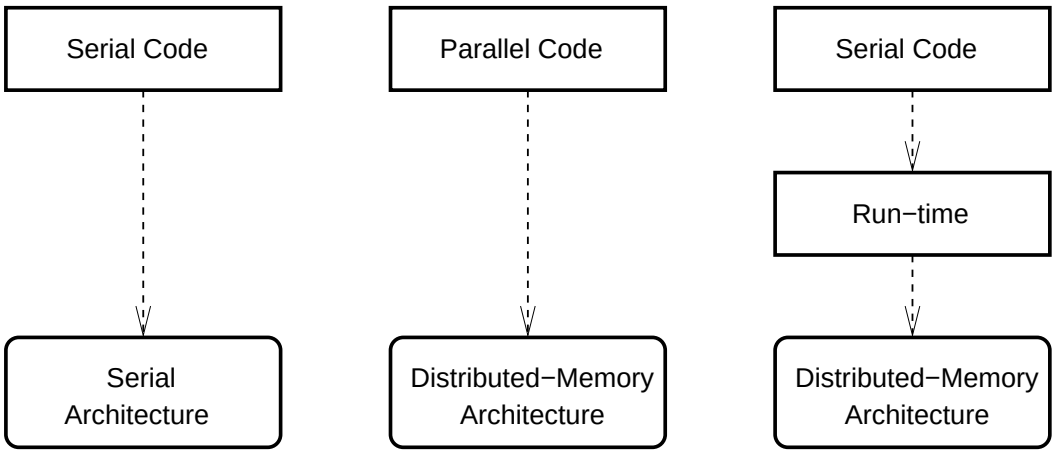

Figure 4: Left, programming serial architectures with serial codes; center, programming distributed-memory architectures with parallel codes; right, programming distributed-memory architectures with serial codes by using a run-time.

processing and some other processes may do some communications.

As usual in programming on distributed-memory architectures, at the beginning of the application data must be partitioned and distributed between processes. That means it must be defined which data block will be stored in each node. In usual distributed-memory libraries, a 2D block

cyclic data distribution is commonly used, since it is more scalable and it is a generalization of 1D row block cyclic and 1D column block cyclic (and it can be transformed into those by adjusting some input parameters).

We used that same distribution: 2D block cyclic data distribution. In contrast to the traditional libraries for distributed-memory architectures, where the data distribution strongly influences and determines data structures, algorithms and codes, the data distribution in our approach does not have so much influence, and it can be easily modified with no change at all in the serial code, and with no major changes in the run-time.

This data distribution of blocks among processes makes that each data block can be viewed by a process as either a *own block* (if it belongs to it) or an *alien block* (if it does not belong to it and thus is stored into some other process).

When having a list of tasks and a set of processes, it must also be defined which process will execute each task. We assume that the owner of the task is the process that owns the first output operand since it rendered good results on other architectures. The method to execute the list of tasks is rather simple: Every process processes all tasks, one by one. A detailed procedure is shown next:

- If a process owns a task, that process will execute the task. To be able to do it, it must get all the operands needed. To get all operands, it must received the alien operands (not owned by it), since it has already got its own operands.

- If a process does not own a task, it will not execute it, but it must check whether that task contains some operands owned by it. If so, it will send them to the owner of the task. If the sent operand is an output operand, it must receive it back, since it has been modified by the task.

- If a process does not own the task, and the task does not contain any operand belonging to the process, the process will not do anything and will jump to next task.

As it can be seen in the above method, the way of handling dependencies is rather different to the method used on multi-core computers. In fact, dependencies between tasks are not considered at all.

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

Let us illustrate the previous descriptions with one example. Let us suppose we want to compute the right-looking variant of the Cholesky factorization a matrix of 4×4 blocks on a 2×2 mesh of processes. If the mesh is column-wise numbered, process 0 (P_0) will own blocks A_{00}, A_{02}, A_{20}, and A_{22}; P_1 will own blocks A_{10}, A_{12}, A_{30}, and A_{32}; P_2 will own blocks A_{01}, A_{03}, A_{21}, and A_{23}; and P_3 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 5. Once the list is generated in every process, all of them start processing the full list, task after task.

At the very beginning, all processes examine first task: $A_{00} := \text{CHOL}(A_{00})$. P_0 checks that the task belongs to it and then that it has got all the operands needed. Therefore, P_0 starts computing the Cholesky factorization of A_{00}. At the same time, the rest of processes check that the task does not belong to them, and no operand in it belong to them. Therefore, they must skip it (by doing nothing), and jump to examine the task 2.

While P_0 is factorizing block A_{00}, the rest of processes start checking task 2. Process P_1 checks that it must execute task 2, and then it checks the status of the operands of this task. As A_{10}

#	Task	Output Operands	Input Operands
1	CHOL	A_{00}	A_{00}
2	TRSM	A_{10}	A_{10} A_{00}
3	TRSM	A_{20}	A_{20} A_{00}
4	TRSM	A_{30}	A_{30} A_{00}
5	SYRK	A_{11}	A_{11} A_{10}
6	GEMM	A_{21}	A_{21} A_{20} A_{10}
7	GEMM	A_{31}	A_{31} A_{30} A_{10}
8	SYRK	A_{22}	A_{22} A_{20}
9	GEMM	A_{32}	A_{32} A_{30} A_{20}
10	SYRK	A_{33}	A_{33} A_{30}
11	CHOL	A_{11}	A_{11}
12	TRSM	A_{21}	A_{21}
13	TRSM	A_{31}	A_{31}
14	SYRK	A_{22}	A_{22} A_{21}
15	...	...	...

Figure 5: List of first tasks needed to factorize a matrix partitioned in 4×4 blocks.

belongs to itself, there is not any problem about it. But as the other block it needs, A_{00}, belongs to P_0 (an alien block for P_1), it must wait for it. Processes P_2 and P_3 check that they are not involved at all with this task and then jump to next task.

When P_0 finishes factorizing block A_{00}, it jumps to task 2. When examining that task, it checks that its own block A_{00} is needed by that task and therefore it must send it to P_1, the owner of the task. Once sent, process P_1, which was waiting for it, receives it and then executes task 2. And so on.

3.2.1 Implemented Run-Times

The technique of using serial codes and a run-time has been successfully employed in programming multi-core architectures, multi-GPU architectures, and solving problems with data stored in disk.

The generation of the list of tasks is similar for all those approaches, with some differences: No task dependencies are needed and the list of tasks must be generated by every process.

The run-time for distributed-memory machines is completely different, since the handling of dependencies is very different: In this case, all processess must analyze all tasks, due to the distributed nature of the architecture.

We implemented and evaluated two different run-times.

The first one is a very simple one. For each task, all alien blocks (not owned by task owner) are transferred. Input alien blocks are only sent by their owners, and output alien blocks are sent and then received by their owners, since they have been modified.

The second run-time tries to reduce the number of transfers between processes, by using a cache of data blocks in every node, to store most recently used alien blocks. Before one block is sent by its process owner, it is checked if the receiver (the task owner) has got it in its cache. If so, no transfer is done. We employed a four-set associative cache with 16 blocks. This technique reduced number of transfers between processes in a significant amount.

At the moment, both run-time versions only use point-to-point communications and thus they are not fully scalable. We expect that including collective communication, as distributed-memory architectures do, will overcome this in the future.

3.3 Comparison of Code Lengths

As it has been above mentioned, Choleksy factorization in LAPACK (serial library) is 141 line long, while the same factorization in ScaLAPACK (distributed-memory library) is 270 line long. While the former is a serial code, the latter is a concurrent code, and thus more complex. In our approach, our Cholesky factorization is about 70 line long of serial code. That length is much shorter than previous approaches. Therefore, in our new approach the programming task is much less complex both in quantity (shorter codes) and in quality (reuse of serial programs and intensively tested codes).

4 Experimental Results

We tested both ScaLAPACK codes and our codes on three very different machines, trying to evaluate them on a wide range of systems:

- Ra is a distributed-memory machine equipped with Xeon processors. Each node is a 32-bit Intel Xeon at 2.4 GHz with 512 GB 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 speed of 100 Megabits/s). Therefore, this machine consists of slow processors inteconnected 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 22.64 GFlops. All communications are performed through the shared memory. Therefore, this machine consists of very fast processors inteconnected with a very fast network (shared memory).

- Peco is a distributed-memory machine with 4 nodes. Each node has two Intel Xeon E5520 at 2.27 GHz (8 cores in total per node). The peak speed of each core is 18.16 GFlops. The interconnection network is a InfiniBand with a peak speed of 40 Gigabits/s.

We employed GNU compilers, GotoBLAS library, the MPICH implementation of the MPI standard, and libflame r1737.

We tested all the implementations in 4 and 8 (or 9) processors/cores. For both ScaLAPACK and our approach we tested most configurations. On 4 processors we tested the following mesh configurations: 1×4, 2×2, and 4×1. On 8 processors we tested the following mesh configurations: 1×8, 2×4, 4×2, and 8×1. On 9 processors we tested the following mesh configurations: 1×9, 3×3, and 9×1. Only results for best mesh configurations are shown in the figures.

Block sizes employed in ScaLAPACK implementations were: 32, 48, 64, 96, 128, 192, and 256. Block sizes employed in our new codes were: 64, 96, 128, 160, 192, 224, 256, and 288. Only results for best block sizes are shown in the figures.

Figure 6 contains the experimentals results obtained for the factorization of Cholesky. Figure 7 contains the experimentals results obtained for the QR factorization. Both figures show performances in GFlops against matrix size.

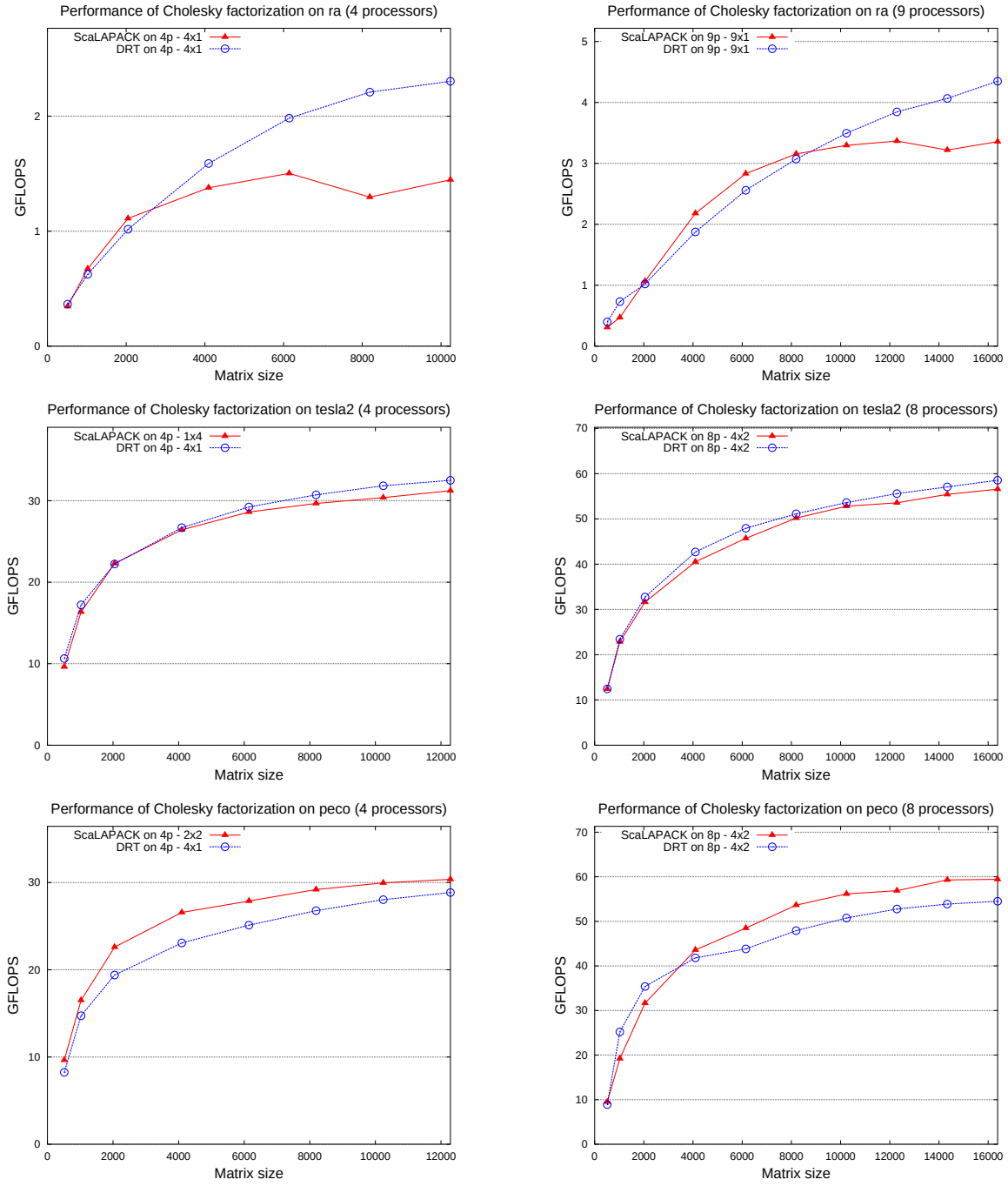

Figure 6: Performances of ScaLAPACK codes and the new codes (DRT) for computing the Cholesky factorization on three different systems. Only results for best mesh configurations and best block-sizes are shown.

In both figures, DRT means Distributed Run-Time and refers to our combination of serial algorithms running onto a distributed run-time.

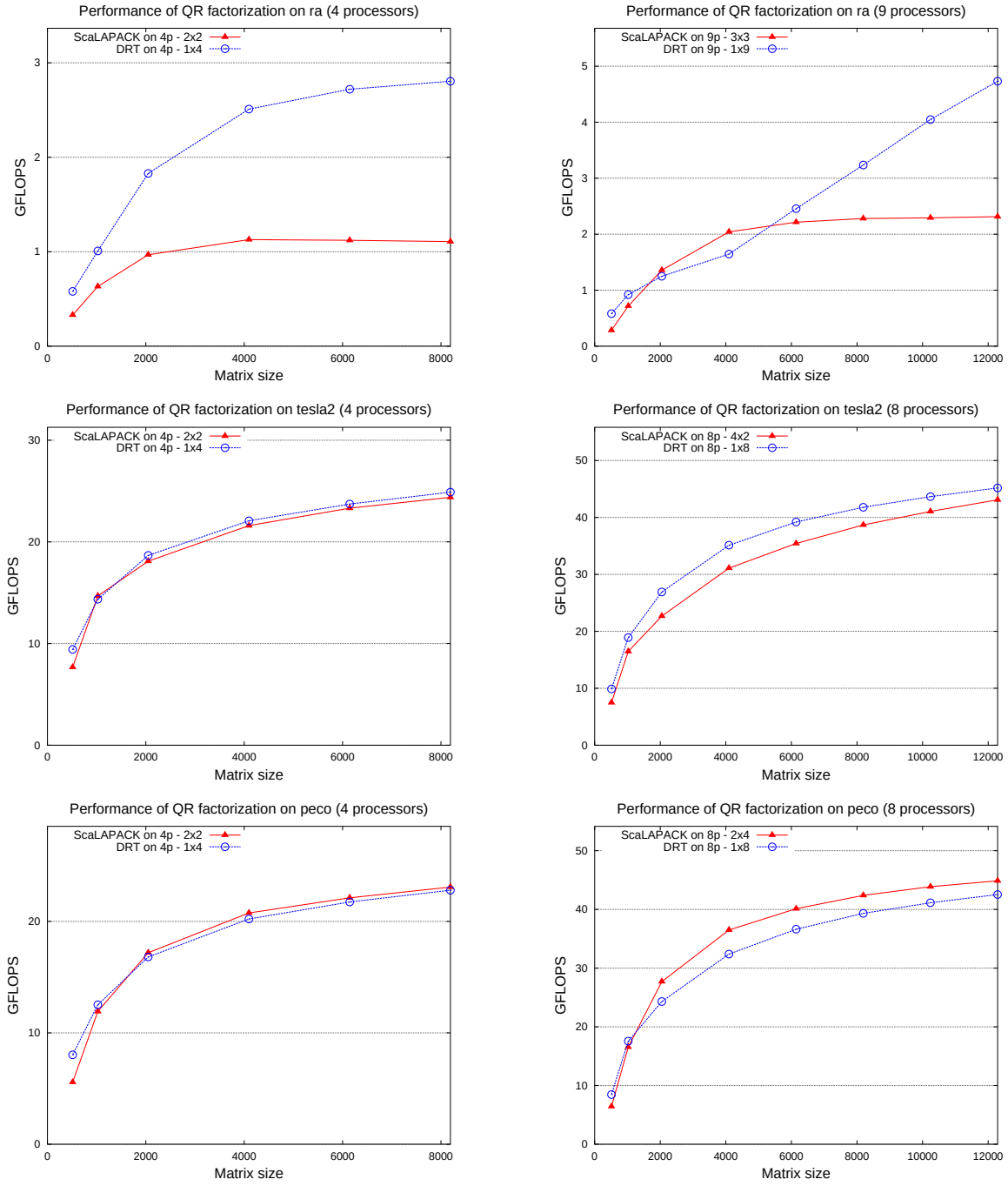

Figure 7: Performances of ScaLAPACK codes and the new codes (DRT) for computing the QR factorization on three different systems. Only results for best mesh configurations and best block-sizes are shown. Note that ScaLAPACK computes the traditional QR factorization, whereas our new approach computes the incremental QR factorization.

The above figures show that performances of our new codes are competitive to those of well-known libraries. However, the developing time is neither the same nor similar, but much shorter.

We believe the above results to be representative of other linear algebra problems. The improvement of performances could have been even larger if we had used a full-scalable run-time.

5 Conclusion

We have demonstrated how with a relatively little effort, programmers can get similar performance to those of well-known libraries on distributed-memory machines. The attained results are of wide appealing for small to medium sized clusters. The building of traditional distributed-memory libraries took many man-years, while our project has been much much shorter: The programs we wrote for this paper required only a few afternoons.

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 run-time system, this transition becomes straightforward for the programmer.

Once the run-time has been developed, porting another variants of the above factorizations (left-looking Cholesky, lazy Cholesky, left-looking QR factorization, etc.) or even new factorizations is much easier than with traditional libraries.

We are working now in rewriting the run-time to increase its scalability. Though a rewriting of some aspects of the run-time will be needed, the clear advantage is that the basic serial algorithms will not have to be modified at all. Another interesting improvement is to use macroblocks or aggregations of blocks to be able to use non-tiled factorizations.

We think our work can give insight into how processors with very large number of cores, like the SCC processor, may be programmed in the future.

Acknowledgements

This research is sponsored by Microsoft Corporation.

References

[1] W. Aspray. *John von Neumann and the Origins of Modern Computing*. The MIT Press, 1990.

[2] 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.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[4] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 191 UT-CS-07-600, University of Tennessee, September 2007.

[5] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.

[6] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007a. ACM.

[7] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.

[8] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference*, February 2010.

[9] Gregorio Quintana-Ortí, Francisco D. Igual, Mercedes Marqués, Enrique S. Quintana-Ortí, and Robert van de Geijn. A run-time system for programming out-of-core algorithms-by-tiles on multithreaded architectures. *ACM Transactions on Mathematical Software*, submitted, 2010.

[10] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*, pages 121–129, 2009.

[11] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, 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):14:1–14:26, July 2009.

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