Symmetric Rank-k Update on Clusters of Multicore Processors with SMPSs

Rosa M. Badia [a], Jesus Labarta [a], Vladimir Marjanovic [a], Alberto F. Martín [b], Rafael Mayo [c], Enrique S. Quintana-Ortí [c], and Ruymán Reyes [c]

[a] *Computer Sciences Department*
Barcelona Supercomputing Center (BSC-CNS), 08034-Barcelona, Spain
{*rosa.m.badia,jesus.labarta,vladimir.marjanovic*}*@bsc.es*

[b] *Centre Internacional de Mètodes Numèrics a l'Enginyeria (CIMNE)*
Univ. Politècnica de Catalunya, 08860–Castelldefels, Spain
amartin@cimne.upc.edu

[c] *Dept. de Ingeniería y Ciencia de Computadores*
Univ. Jaume I, 12.071–Castellón, Spain
{*mayo,quintana,rreyes*}*@icc.uji.es*

Abstract. We investigate the use of the SMPSs programming model to leverage task parallelism in the execution of a message-passing implementation of the symmetric rank-k update on clusters equipped with multicore processors. Our experience shows that the major difficulties to adapt the code to the MPI/SMPSs instance of this programming model are due to the usage of the conventional column-major layout of matrices in numerical libraries. On the other hand, the experimental results show a considerable increase in the performance and scalability of our solution when compared with the standard options based on the use of a pure MPI approach or a hybrid one that combines MPI/multi-threaded BLAS.

Keywords. Linear algebra, ScaLAPACK, clusters of multi-core processors, SMPSs, message passing numerical libraries

1. Introduction

The symmetric rank-k update (SYRK) is a specialized case of the matrix-matrix product, included in the specification of the *Basic Linear Algebra Subprograms* (BLAS), which plays an important role in the solution of symmetric positive definite (s.p.d.) linear systems via the Cholesky factorization [3]. The SYRK computes the upper (or lower) triangular part of the result of the matrix product

$$C := \beta C + \alpha A^T A, \tag{1}$$

where C is an $m \times m$ symmetric matrix, A is a $k \times m$ matrix, and α, β are scalars. (For simplicity, hereafter we assume that only the upper triangular part of C is updated/referenced.) Exploiting the symmetry of matrix C effectively reduces the cost of this operation to $m^2 k$ floating-point arithmetic operations (flops).

When the matrices involved in the operation are large ($m, k \approx O(10^4 - 10^5)$), as e.g. happens in the solution of generalized eigenvalue problems for the analysis and modeling of molecular structures, clusters of computers combined with parallel (message-passing) dense linear algebra libraries, such as ScaLAPACK [2] or PLAPACK [7], are usually necessary to speed-up the time-to-response. Current cluster platforms are equipped with one or more multicore processors per node. One conventional approach to exploit these two levels of hardware parallelism (inter-node and intra-node) from numerical linear algebra libraries such as ScaLAPACK/PLAPACK is to place one MPI process per core (i.e, use a pure MPI approach, with as many MPI processes as cores per node). Alternatively, one can run one MPI process per node and rely on a multi-threaded implementation of the BLAS (hybrid MPI/MT-BLAS) to exploit the intra-node hardware parallelism.

In this paper we describe our experience using SMPSs [6,4] to leverage intra-node hardware parallelism from within a message-passing implementation of SYRK. Concretely, we address the real, double-precision code for this operation in the ScaLAPACK library, PDSYRK, making the following two major contributions:

- We describe in detail the parallelization of the symmetric rank-k routine in ScaLAPACK using SMPSs, which serves as a case study to illustrate how to tackle other message-passing dense linear algebra codes, from ScaLAPACK or other libraries with similar functionality, with this framework. The description identifies relevant difficulties encountered during the parallelization, due e.g. to algorithmic restrictions embedded in the code and the semantics of the SMPSs programming model, as well as alternatives which may be used to overcome these problems.
- We demonstrate performance and scalability of the parallelization of PDSYRK using SMPSs substantially higher than those of the conventional pure MPI or MPI/MT-BLAS practices.

The rest of the paper is structured as follows. In Section 2 we briefly review the algorithm(s) underlying the implementation of PDSYRK in ScaLAPACK. The parallelization of this code using SMPSs is illustrated in Section 3. Numerical experiments on a cluster consisting of eight nodes, with 8 cores each, and connected a high-speed Infiniband interconnect are reported in Section 4. Finally, a few concluding remarks follow in Section 5.

2. The ScaLAPACK Implementation of SYRK

The ScaLAPACK routine PDSYRK implements the symmetric rank-k update with data matrices distributed following a block-cyclic 2D layout among a $p \times q$ (logical) grid of processes. Thus, C and A are partitioned into square blocks, of size d_s (the data distribution blocking factor), which are then mapped to the 2D process grid in a block-cyclic fashion. The block-cyclic layout determines the communication pattern of the distributed algorithm. The algorithm employs standard kernels for local vector and matrix operations from BLAS and the message-passing communication layer tailored for dense linear algebra in BLACS.

Internally, PDSYRK encodes two distinct algorithmic variants, PB_CPSYRKAC and PB_CPSYRKA. The first communicates both matrix operands and is chosen when the volume of communication is estimated to be below 1.3× that of the second; in the latter, data transfers only involve matrix A. Here we will focus on this second routine as, except for the last iterations, this is the common case invoked from ScaLAPACK routine PDPOTRF for the Cholesky factorization of a s.p.d. matrix.

Assume for simplicity that $m = \mu \cdot d_s$ and $k = \kappa \cdot d_s$, with μ and κ integers, and consider the matrix partitionings $C \rightarrow (C_{ij})$ and $A \rightarrow (A_{ij})$ into blocks of dimension $d_s \times d_s$ induced by the block-cyclic 2D layout. Routine PB_CPSYRKA computes (1) in κ steps, where at each step i, with $i = 1, 2, \ldots, \kappa$, the following events occur:

1. Processes in the (i mod p)-th row first broadcast column-wise their local portions of the i-th block row of A, $A_i = [A_{i,1}, A_{i,2}, \ldots, A_{i,\mu}]$.
2. A_i is transposed onto the (i mod q)-th column of processes, yielding $\bar{A}_i = A_i^T$, and these processes then broadcast row-wise their local parts of $\bar{A}_i$.
3. All the processes in the grid update their local portions of C w.r.t. A_i^T and A_i, $C_{r,s} := C_{r,s} - \bar{A}_{i,r} \cdot A_{i,s}$.

In the general case ($p \neq q$) the transposition of A_i onto a column of processes essentially requires a point-to-point communication from each process of the (i mod p)-th row to the appropriate process of the (i mod q)-th column of processes. The blocks of A to be exchanged are packed by the source process into temporary buffers, and unpacked/transposed into an auxiliary workspace in the destination; for details, see [5].

The parallel update of C decouples the algorithmic blocking factor, a_s, from the data distribution blocking factor in order to increase the granularity of the invocations to the level-3 BLAS that are necessary to perform the local computations. The diagonal blocks induced by this algorithmic partitioning are updated via several fine-grained calls to the level-3 BLAS, possibly on more than one node. The update of the off-diagonal blocks, on the other hand, involves a single coarse-grain invocation to the level 3 BLAS per node.

3. Parallelization using SMPSs

3.1. Brief introduction to SMPSs

SMPSs combines a language with a reduced number of OpenMP-like pragmas, a source-to-source compiler, and a run-time system to leverage task-level parallelism in sequential codes. At the node level, SMPSs exploits task parallelism by (semi-)automatically decomposing the problem (i.e., code) into tasks, and dynamically identifying dependencies among these and issuing *ready tasks* (those with all dependencies satisfied) to be executed to the cores of the system.

SMPSs is an active project that targets multiple different hardware platforms (Grids; platforms with multiple hardware accelerators: GPUs, Cell B.E., Clearspeed boards; heterogeneous systems, etc.) with distinct implementations of the

framework. One particular appealing instance for our purposes is MPI/SMPSs, which provides specific support for MPI applications. In this particular version there exists the possibility of embedding calls to MPI communication primitives as SMPSs tasks, and a separate thread devoted to communication is created dynamically by the runtime.

3.2. Parallelizing PDSYRK *with SMPSs*

We next describe in detail the changes that were introduced into the original ScaLAPACK code for PDSYRK to accommodate SMPSs, the difficulties that we encountered, and how these were overcome.

3.2.1. Capturing data dependencies

Our present parallelization of the ScaLAPACK routine using the MPI/SMPSs programming model is greatly influenced by two current restrictions[1] of this programming model. First, dependencies among tasks are identified by comparing the base address of the corresponding operands, so that memory *regions* corresponding to different operands cannot overlap. Second, data for matrix/vector operands accessed by tasks must be stored contiguously in memory. This second constraint has important implications in our case because ScaLAPACK/MPI/BLAS employ column-major storage and, therefore, computational/communication kernels used from within PDSYRK indeed access noncontinuous data. Among the different solutions that were considered, we decided to adopt the usage of "sentinels" or "representants" [4], as it is applicable to most dense linear algebra codes and avoids significant recoding [1]. In this particular technique, the top-left (first) entry of a matrix (vector) operand acts as the representant for the whole data contained in it so that the runtime identifies dependencies based solely on its address.

Notice that the use of sentinels implies that we must enforce that (regions in memory corresponding to) distinct operands accessed by tasks are separated, with no overlapping among them. In the next two subsections, we will see how we had to further adapt the codes to enforce this.

3.2.2. Local computations

Routine PDSYRK decouples the algorithmic and distribution block sizes with the purpose of casting all local computations in terms of a reduced number of coarse-grain level 3 BLAS. Thus, in order to parallelize the local computations, we could simply encapsulate the calls to the level 3 BLAS from within PDSYRK as SMPSs tasks. However, this straight-forward approach has two major drawbacks, related with the algorithmic partitioning and load balancing, which we describe next.

In routine PDSYRK, local updates to the off-diagonal blocks are aggregated into one large `dgemm` (matrix-matrix product), while diagonal blocks are updated via repeated invocations to the `dsyrk` kernel. This is illustrated in Figure 1 (left), which shows the operations performed in a single node to update the local portions of C with the received parts of A_i and $\bar{A}_i$ (in the figure, we refer to these operands

[1] While other variants of SMPSs do not suffer from these limitations, they cannot be used in the parallelization of PDSYRK as they provide no support for MPI applications.

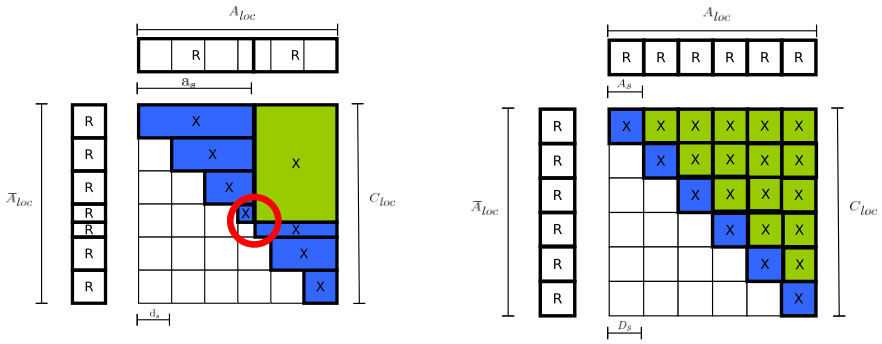

Figure 1. Example of the update of local blocks performed in a single process during the i-th step of PDSYRK. Blocks labelled with an "`R`" are only read, while blocks containing an "`X`" are both read and written. The update of the green region(s) is done via call(s) to `dgemm` (matrix-matrix product) while blue regions are updated via invocations to `dsyrk` (symmetric rank-k update). Left: update with unaligned partitionings shows an overlapped region (highlighted in red) which leads to a false dependence detection. Right: Aligned partitionings easily solve this problem and taskification of the updates yields a more balanced distribution.

as C_{loc}, A_{loc} and $\bar{A}_{loc}$, respectively). In principle, the update of each one of the blocks in C_{loc} marked with an "`X`" corresponds to an SMPSs task. The problem here is that employing the top-left entry of the block of C_{loc} highlighted with a red circle as a sentinel (representant) for the two `dsyrk` operations that update part of its data leads to erroneously identifying a dependency between these two tasks. In other words, the base-address-match dependency test performed by the runtime detects a false data dependency between these two operations, serializes their execution and, in consequence, unnecessarily limits the parallelism intrinsic to the update.

To deal with this problem, we propose to align both partitionings by readjusting the internal algorithmic block size a_s to the closest multiple of the data distribution block size. Furthermore, to solve the imbalance due to the invocation to coarse-grain kernels and the mapping of these into SMPSs tasks, we decompose these calls into multiple finer-grain tasks. The result from these two techniques is illustrated in Figure 1 (right): no overlapping occurs now, which allows the runtime to correctly identify dependencies among the tasks which operate on these blocks; and the granularity of tasks is much smaller and regular, easing a balanced workload distribution. In practice, the optimal distribution block size is relatively small (in our experiments, in the range of 128 to 384) so that we do not expect that the restriction on the algorithmic block size poses a major obstacle for the performance of the routine.

3.2.3. Communication kernels

Two types of BLACS primitives are invoked from the PDSYRK routine: point-to-point messages (exchanged to transpose A_i) and broadcasts (to propagate copies of the information within the row/column of processors); see section 2. However, in BLACS the broadcast is cast in terms of point-to-point primitives so that, in

practice, it suffices to develop a single strategy for the taskification of point-to-point send/receive calls.

To identify data dependencies among communication and computation tasks in MPI/SMPSs, the former need to be blocked conformally with the data distribution partitioning. Although this can be achieved by decomposing a BLACS send/receive invocation into a number of sends/receives (as was done for the computational kernels), this option was discarded as we instead preferred to preserve the communication pattern of the initial distributed algorithm in ScaLAPACK and, for programmability and simplicity, avoid recoding (taskifying) the communication primitives in BLACS.

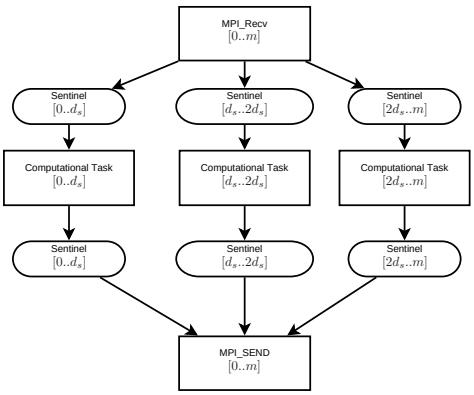

Figure 2. Example of taskification of communication kernels. Each node represents a SMPSs task. Assume that communication kernels operate with a buffer of m elements. Computational tasks operate with m/d_s elements ($m/d_s = 3$ in current example). To track of the dependencies, artificial tasks are created for each portion of the buffer, thus maintaining only one communication task.

The communication-preserving taskification scheme is illustrated in Figure 2. A receive call is annotated as a receive task plus several artificial tasks, one per block of the reception buffer. Both classes of tasks are mapped to the communication thread. The receive task actually receives the message, while the artificial tasks do nothing, they only receive the address of the corresponding block as an argument. (The overhead due to the introduction of these artificial tasks is negligible.) By specifying this parameter as an output to the corresponding task, the correct data dependency is created between computation and communication tasks. When the actual data-flow execution takes place, the communication thread first issues the receive task. Once the execution of this task is completed (i.e., the point-to-point communication is done), the data dependencies of the artificial tasks are satisfied, so that they are immediately executed by the communication thread. A similar strategy was developed for the send calls, but in this case the directionality of the data dependency was reversed using the input clause for the artificial tasks.

4. Experimental Results

All experiments reported next were obtained using IEEE double-precision arithmetic on an Infiniband-interconnected cluster composed of 8 nodes, with 8 cores (two Intel Xeon QuadCore processors E5410 –Nehalem– at 2.27 GHz; peak performance of 9.08 GFLOPS = $9.08 \cdot 10^9$ flops/sec. per core) and 24 Gbytes per node. The following software libraries were employed: ScaLAPACK 1.8.0, BLACS 1.1 on top of OpenMPI 1.4 for native message-passing, and BLAS in Intel MKL 10.3. Our prototype was compiled using an up-to-date version of the MPI/SMPSs tools (v2.3). Finally, we used Intel C/Fortran77 11.1 as the native compilers.

Three different implementations are evaluated in the experiments:

- `Reference.` This corresponds to a pure MPI implementation based on the use of the original PDSYRK routine from ScaLAPACK, employing a $p \times q$ grid of processes with one MPI process per core. For the experiments with 4 nodes and 32 cores, the grid was set to $p \times q = 4 \times 8$. In the experiments with 8 nodes and 64 cores, the grid was 4×16.
- `MT-BLAS.` This is a hybrid MPI/MT-BLAS parallel approach, with one MPI process per node and parallelism extracted at the node level with an implementation of MT-BLAS. The grids for the experiments with 4 and 8 nodes (32 and 64 cores, resp.) were $p \times q = 2 \times 2$ and 2×4.
- `SMPSs.` This is the version that employs SMPSs to extract parallelism at the node level from the ScaLAPACK code, configured with one MPI process per node, and the same grids as in the `MT-BLAS` implementation.

In all cases, a best-effort was done to identify the optimal distribution size d_s for each problem size and parallel implementation. In the former two cases, the algorithmic size a_s was that set internally in ScaLAPACK PDSYRK. In the latter case, this value for a_s was rounded to the closest integer multiple of d_s. A best effort was also conducted to determine the optimal grid configuration for each implementation, leading to the parameters indicated above.

Figure 3 (left) reports the performance attained by the three parallel implementations using 4 nodes/32 processor cores. The results show that the `SMPSs` version outperforms both the `MT-BLAS` and `Reference` implementations, achieving an asymptotic efficiency close to 87% of the theoretical peak (9.08 GFLOPS/-core $\times$ 8 cores/node $\times$ 4 nodes =290.56 GFLOPS) already for problems of moderate size, while `MT-BLAS` and `Reference` implementations require much larger problems to deliver asymptotic efficiencies of 78% and 84%, respectively. Figure 3 (right) shows how the performance evolves when the number of nodes and cores are doubled, to 8 and 64 respectively. In this case the `SMPSs` version delivers nearly 85% of efficiency (peak on 64 cores is 581.12 GFLOPS) for the largest problem size, while `MT-Reference` peaks at a low 70% and the `MT-BLAS` implementation yields an even lower 72%.

5. Concluding Remarks and Future Work

This paper describes the major difficulties encountered during the port of the message-passing implementation of a well-known dense linear algebra kernel, the

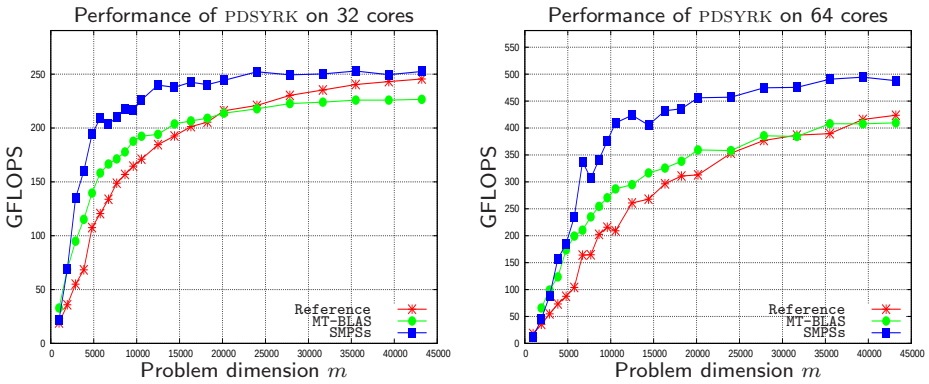

Figure 3. Performance of the symmetric k-rank update on 4 nodes/32 cores (left) and 8 nodes/64 cores (right), with fixed $k = 1024$ and varying m.

symmetric rank-k update, to the MPI/SMPSs programming model, and the solutions adopted to overcome these problems. Experimental results on a platform consisting of 8 nodes equipped with state-of-the-art multicore technology and communication interconnect report remarkable performance and scalability gains enabled by the use of this programming model.

Ongoing work on a new version of MPI/SMPSs that can transparently deal with noncontiguous data regions is expected to ease the migration of these libraries while maintaining performance similar to that reported here for the MPI/SMPSs approach.

Acknowledgements

This research was supported by Project EU INFRA-2010-1.2.2 "TEXT: Towards EXaflop applicaTions". The researchers at Universidad Jaume I were supported by project CICYT TIN2008-06570-C04-01 and FEDER.

References

[1] R. M. Badia, Jose R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21:2438–2456, 2009.

[2] 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.

[3] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[4] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008.

[5] A.P. Petitet and J.J. Dongarra. Algorithmic redistribution methods for block-cyclic decompositions. *IEEE Trans. Parallel and Distributed Systems*, 10(12):1201 –1216, dec 1999.

[6] SMP superscalar project home page. `http://www.bsc.es/plantillaG.php?cat_id=385`.

[7] R. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.