A Framework for Dynamic Node-Scheduling of Two-Sided Blocked Matrix Computations

Lars Karlsson and Bo Kågström

Department of Computing Science and HPC2N, Umeå University,
S-901 87 Umeå, Sweden, {larsk,bokg}@cs.umu.se

Abstract. Blocked matrix algorithms are characterized by a high utilization of floating point units. Memory bandwidth is not a critical issue due to the surface-to-volume effect of level 3 algorithms. Factors limiting the performance of distributed algorithms include communication overhead and spurious synchronizations. Load balance is often achieved by using a 2D Block Cyclic Layout. To reduce communication overhead and synchronizations, a node algorithm is often rearranged into an efficient but more complicated variant. Frameworks for dynamic scheduling of node programs promise to remove much of the complexities while preserving most of the performance improvements. We present a design of a minimalistic framework for dynamic scheduling specifically targeting two-sided blocked matrix computations. A model algorithm, nonscalable in its straightforward implementation and with applications in modern algorithms for the nonsymmetric eigenvalue problem is shown to be scalable in practice. The scalability is enabled by the framework, specifically by the priority-based scheduling mechanism.

1 Introduction

Blocked matrix algorithms on possibly hybrid distributed memory machines (the nodes themselves may be shared address space parallel computers) usually utilize a large fraction of the machine's peak performance [8]. The main limiting factors are communication overhead and spurious synchronizations, both between and within nodes. Communication overhead can often be reduced by rearranging the node algorithm to expose overlap possibilities via nonblocking communication. The overlap is then exploited in hardware via network interfaces with Direct Memory Access (DMA) which ultimately reduces the communication overhead [5]. Spurious synchronizations are handled similarly; by rearranging the node algorithm, spurious synchronizations can be removed.

The added complexity increases the likelihood of programming errors. For some algorithms it is also difficult to find an almost optimal schedule, which typically depends on machine parameters, problem and block sizes.

To remove most of the extra complexity one may use *dynamic scheduling*. In such dynamic implementations the execution order of different portions of the node program (its *tasks*) is *non-deterministic*. Some recent work on dynamic scheduling on shared address space and multicore architectures are [6,7,3,2].

Typically, a new bookkeeping overhead is introduced and some programmer assistance in decomposing the program into tasks as well as guidance on enforcing dependencies is required. A dynamic scheduling approach mainly benefits programmer productivity, maintainability, quality assurance, and portability. Performance for particular problems may often be suboptimal due to unexploited problem-specific optimizations.

In this contribution, we present a design of a minimalistic and efficient dynamic scheduling framework. The design is intended for blocked matrix computations on hybrid distributed memory machines. The tasks are statically distributed to processes and dynamic scheduling is applied only on the nodes.

We introduce a model algorithm which appears as the computationally dominant part in several algorithms related to the nonsymmetric eigenvalue problem [4,10,9,1]. A straightforward parallelization on a 2D Block Cyclic Layout (BCL) is nonscalable. This is not caused by load imbalance but rather by a suboptimal execution order that introduces spurious synchronizations. We show that the priority-based scheduling in our framework allows flexible and more efficient execution orders to be imposed. The algorithm expressed using the framework has the same overall structure as the straightforward implementation, demonstrating a clear advantage of a dynamic approach compared with a manual, static approach that would require restructuring of the code.

2 Framework Design

Our framework design consists of four major parts:

1. An API to express a node program in a familiar sequential style while allowing it to be executed by multiple threads.
2. An efficient algorithm and API that at runtime identifies data dependencies from programmer declarations of read and write accesses to matrix blocks.
3. A scheduling and dependency tracking mechanism.
4. An API and scheduling mechanism for asynchronous matrix-based communication built on top of MPI [11].

2.1 Application Programming Interface

The main API of the framework is used to annotate a correct sequential node program so that during execution it constructs a correct Directed Acyclic Graph (DAG) representation of the program. The DAG is then scheduled onto a team of threads consisting of one master thread (the thread that builds the DAG) and $w \geq 0$ worker threads. The master thread handles all calls to MPI functions which means that the MPI implementation does not need to be thread-safe.

The framework considers two types of tasks: computation tasks (user-defined) and communication tasks (fully integrated).

Computation tasks are constructed from a sequence of API function calls. First a DAG node and its payload (user-defined task-specific information) are

created, then all task dependencies are constructed, and finally the node is committed to the scheduler.

The scheduling and execution of communication tasks is fully integrated within the framework. Communication tasks are constructed using a separate API for matrix-based communication. The structure of task creation is the same as for computation tasks.

Data dependencies for a particular task are identified at runtime by the method described in Section 2.2. During task construction a sequence of API calls is used to specify read and write accesses to one or more matrix blocks.

2.2 Runtime Data Dependence Analysis

Due to the surface-to-volume effect in level 3 blocked matrix computations the number of floating point operations is often far greater than the number of memory references. The difference is even greater between floating point operations and *matrix block* references. This holds for the entire computation as well as for all of its level 3 subcomputations (such as `GEMM` updates, recursive panel factorizations, etc.) and is at the heart of the method described in this section.

Data dependencies may tightly couple otherwise independent parts of a large program. Manual analysis could therefore be prohibitively difficult. *Approximation* of data dependencies at *runtime* using matrix blocks as the unit of memory reference effectively solves the problem of coupling.

A task may request read-only access (read request) or read/write access (write request) to a block. Concurrent reads are allowed but writes serializes. Artificial DAG nodes (so called *join* nodes) are constructed to collect dependen-

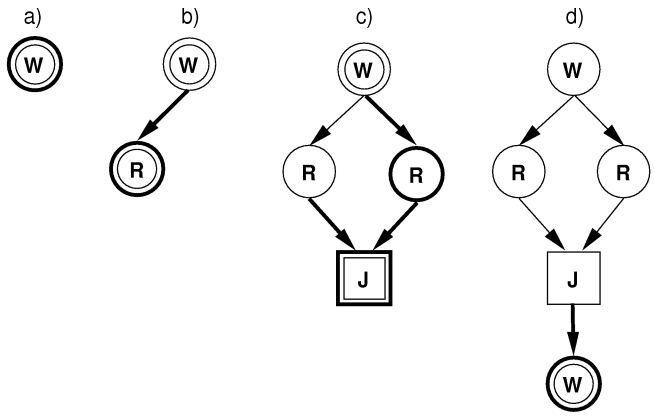

Fig. 1. Example showing how dependencies on a matrix block are efficiently constructed from a sequence of read and write requests (in this case: Write, Read, Read, Write). Double outlines denote nodes which may get new arcs and are hence retained as a part of the algorithm state. Bold arcs are the new dependencies that are added during the current step of the algorithm.

cies from concurrent reads to a single block in anticipation of a write request. This process is illustrated by example in Figure 1 and it allows the amount of memory required by the algorithm to remain constant.

3 Application Example

A model algorithm (Algorithm 1: `Sweep`) with applications in the nonsymmetric eigenvalue problem is described and analyzed in this section. We use our node-scheduling framework to effectively turn the straightforward implementation of `Sweep` into an efficient, scalable implementation which has two antidiagonal wavefronts running through the matrix.

3.1 Model Algorithm: Sweep

Algorithm 1 details the distributed algorithm from a global perspective. We use a special notation to specify which process or group of processes that are involved in each operation. With `P(i, j)` we mean the process that owns element `A(i, j)`. The notation `P(i, *)` refers to the *process row* that owns the *matrix row* `A(i, :)` and similarly `P(*, j)` refers to the *process column* that owns the *matrix column* `A(:, j)`. In each iteration, a diagonal block of size $n_b \times n_b$ is used to compute an $n_b \times n_b$ orthogonal matrix Q and is modified in the process by applying Q^T from the left and Q from the right. The iteration step is half the block size (the distribution block size is n_b) and the algorithm is assumed to start aligned with a block. At every other iteration the diagonal submatrix is local to one process (a *local* iteration, lines 7–15), whereas during the remaining iterations it is distributed onto four processes (a *cross-border* iteration, lines 17–39).

After the orthogonal matrix Q has been computed and applied to the diagonal block (line 8 or 18) the corresponding block row and column must also be updated in order to complete the orthogonal similarity transformation

$$A \leftarrow \begin{bmatrix} I_{i-1} & & \\ & Q & \\ & & I \end{bmatrix}^T A \begin{bmatrix} I_{i-1} & & \\ & Q & \\ & & I \end{bmatrix}.$$

Figure 2 shows the affected portions of the matrix during two iterations of the loop (one local and one cross-border).

All of the block row and column operations (copy, update, and send/receive) are partitioned into independent suboperations at block boundaries. Each suboperation is regarded as an atomic unit of computation (copy and update) or communication (send/receive) and maps to one task.

3.2 Analysis

We derive an approximate upper bound on the efficiency of the straightforward implementation. We assume that Q can be computed for free, that all updates

Algorithm 1 Sweep: Distributed Memory Model Algorithm

```
1:  local = true
2:  for i = 1 to N-nb+1 step nb/2
3:     a = i
4:     b = i + nb - 1
5:     c = i + nb/2 - 1
6:     if local then
7:        Copy A(a:b, a:b) to C on P(a, a)
8:        Compute Q from C on P(a, a)
9:        Copy C to A(a:b, a:b) on P(a, a)
10:       Broadcast Q to P(a, *) from P(a, a)
11:       Broadcast Q to P(*, a) from P(a, a)
12:       Copy A(1:a-1, a:b  ) to W on P(*, a)
13:       Copy A(a:b,   b+1:N) to S on P(a, *)
14:       Update A(1:a-1, a:b  ) = W*Q  on P(*, a)
15:       Update A(a:b,   b+1:N) = Q'*S on P(a, *)
16:    else
17:       Gather A(a:b, a:b) to C on P(a, a)
18:       Compute Q from C on P(a, a)
19:       Scatter C to A(a:b, a:b) from P(a, a)
20:       Partition Q into two column blocks: Q = [Q1, Q2]
21:       Send Q2 from P(a, a) to P(b, b)
22:       Broadcast Q1 to P(a, *) from P(a, a)
23:       Broadcast Q1 to P(*, a) from P(a, a)
24:       Broadcast Q2 to P(b, *) from P(b, b)
25:       Broadcast Q2 to P(*, b) from P(b, b)
26:       Partition W into two equal column blocks: W = [W1, W2]
27:       Partition S into two equal row    blocks: S = [S1; S2]
28:       Copy A(1:a-1, a:c  ) to W1 on P(*, a)
29:       Send A(1:a-1, a:c  ) to W1 on P(*, b) from P(*, a)
30:       Copy A(1:a-1, c+1:b) to W2 on P(*, b)
31:       Send A(1:a-1, c+1:b) to W2 on P(*, a) from P(*, b)
32:       Copy A(a:c,   b+1:N) to S1 on P(a, *)
33:       Send A(a:c,   b+1:N) to S1 on P(b, *) from P(a, *)
34:       Copy A(c+1:b, b+1:N) to S2 on P(b, *)
35:       Send A(c+1:b, b+1:N) to S2 on P(a, *) from P(b, *)
36:       Update A(1:a-1, a:c  ) = W*Q1  on P(*, a)
37:       Update A(1:a-1, c+1:b) = W*Q2  on P(*, b)
38:       Update A(a:c,   b+1:N) = Q1'*S on P(a, *)
39:       Update A(c+1:b, b+1:N) = Q2'*S on P(b, *)
40:    end if
41:    local = not local
42: end for
```

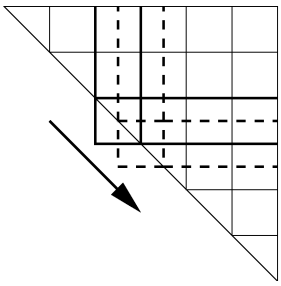

Fig. 2. Illustration of the model algorithm on a sample block matrix with $N = 6n_b$. Solid lines mark the region affected by iteration $i = 2n_b$. The diagonal block is used to compute Q and the corresponding block row and column are modified by the subsequent updates. Dashed lines show the half-block step taken to the next iteration ($i = 2.5n_b$) and is also an example of a cross-border iteration.

perform at a fixed performance of α flops/s. Communication and other sources of overhead are not taken into consideration.

To perform all local block row updates (lines 14–15) over the course of the algorithm *sequentially* requires

$$T_{\text{local}} \approx \frac{N_b n_b^3 (N_b - 1)}{\alpha}$$

seconds. Similarly, to perform all cross-border block row updates (lines 36–39) over the course of the algorithm *sequentially* requires

$$T_{\text{cross}} \approx \frac{N_b n_b^3 (N_b - 2) + n_b^3}{\alpha}$$

seconds. The same timing models hold for block column updates.

When the algorithm executes in parallel on a $P_r \times P_c$ mesh the updates are assumed to be perfectly load balanced across the involved processes. This means that local block row and column updates are parallelized over P_c and P_r processes, respectively. The cross-border block row and column updates are parallelized onto $2P_c$ and $2P_r$ processes, respectively. Since, in every iteration, one process has to participate in both a block row and a block column update these two operations are effectively serialized. The estimate of the parallel execution time of the straightforward implementation is therefore approximated by

$$T_p \approx \frac{T_{\text{local}}}{P_c} + \frac{T_{\text{local}}}{P_r} + \frac{T_{\text{cross}}}{2P_c} + \frac{T_{\text{cross}}}{2P_r} \leq \frac{3T_{\text{local}}}{2} \frac{P_r + P_c}{P_r P_c}.$$

The sequential execution time is $T_s = 2T_{\text{local}} + 2T_{\text{cross}} \leq 4T_{\text{local}}$ which gives the following approximate bound on the efficiency.

$$E_p \leq \frac{8}{3(P_r + P_c)}.$$

3.3 Dual Wavefront Implementation

In order to appreciate the number of allowed schedules we look at a single block column of the matrix and consider a sequence of updates from the left. During the first iteration, `i=1` and the block column is updated on rows `1:nb`. During the second iteration, `i=1+nb` and the block column is updated on rows `1+nb:2*nb`. The affected rows overlap and there is a true data dependence that serializes the updates. The key thing to notice is that, while serialized on a *particular* block column, all block columns are independent. The same reasoning holds for the updates from the right, replacing block column with block row.

This allows for an algorithm where the number of applied updates differ between each block column/row at any given time. The *dual antidiagonal wavefront* implementation described visually in Figure 3 is one example obtainable from the straightforward implementation by the framework and suitable priority assignments.

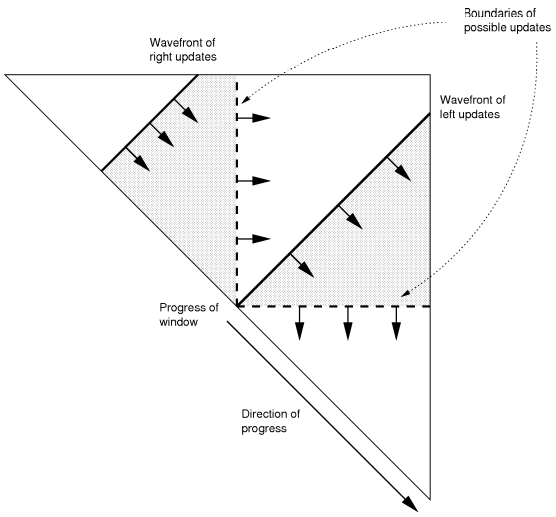

Fig. 3. The two antidiagonal wavefronts (the upper one corresponds to updates from the right, the lower one to updates from the left). The grey areas represent the parts of the matrix that have pending updates at this point.

3.4 Some Computational Results

In this section, we present measurements of execution time in the form of parallel efficiency. We do not yet have an equivalent sequential implementation of the `Sweep` algorithm but as an estimate of the sequential cost we accumulated the time spent in kernel computations (copy and update) in each process. All efficiencies presented in this section are derived in this fashion. In Figure 4 and

Figure 5, we present results for the straightforward implementation and the dual antidiagonal wavefront implementation, respectively. In these figures, the

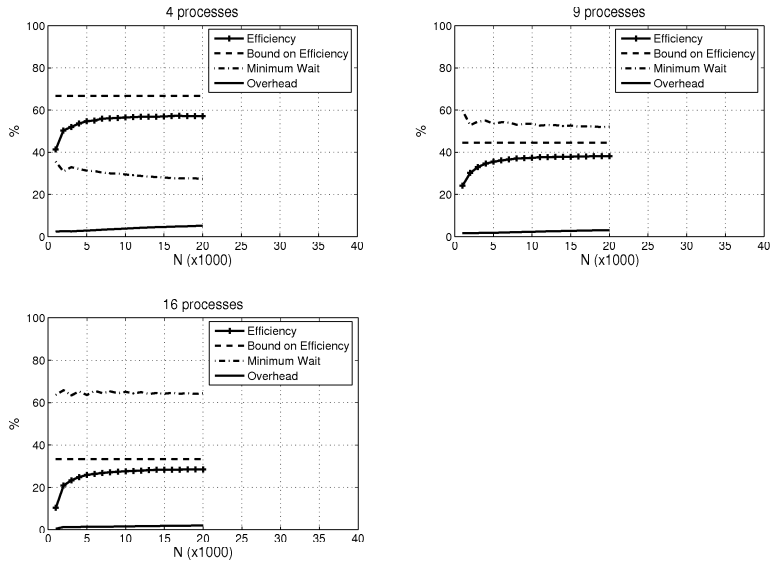

Fig. 4. Efficiency, upper bound on efficiency, overhead, and minimum synchronization overhead (Minimum Wait) for the straightforward implementation of `Sweep` ($n_b = 100$).

following information is displayed.

- *Efficiency.* The relative efficiency ($T^{(i)}_{\text{kernel}}$ refers to the time spent in kernel computations in process i and T_p is the parallel execution time)

$$E_r = \frac{\sum_{i=1}^{P_r P_c} T^{(i)}_{\text{kernel}}}{P_r P_c T_p}.$$

- *Bound on efficiency.* The approximate theoretical upper bound on efficiency,

$$\frac{8}{3(P_r + P_c)},$$

  as derived in Section 3.2.
- *Minimum wait.* On process i, the time spent in blocking MPI functions is referred to as $T^{(i)}_{\text{wait}}$. The minimum wait curve shows the quantity

$$\frac{\min_i T^{(i)}_{\text{wait}}}{T_p}.$$

This metric measures the impact of synchronization overhead on parallel execution time.

– *Overhead.* The fraction of the parallel cost ($P_r P_c T_p$) that is made up of everything except kernel computations, initialization of sends and receives, and time spent in blocking MPI functions. This is the combined overhead of the implementation and includes everything from building and maintaining the DAG to polling the MPI library to miscellaneous unaccounted operations.

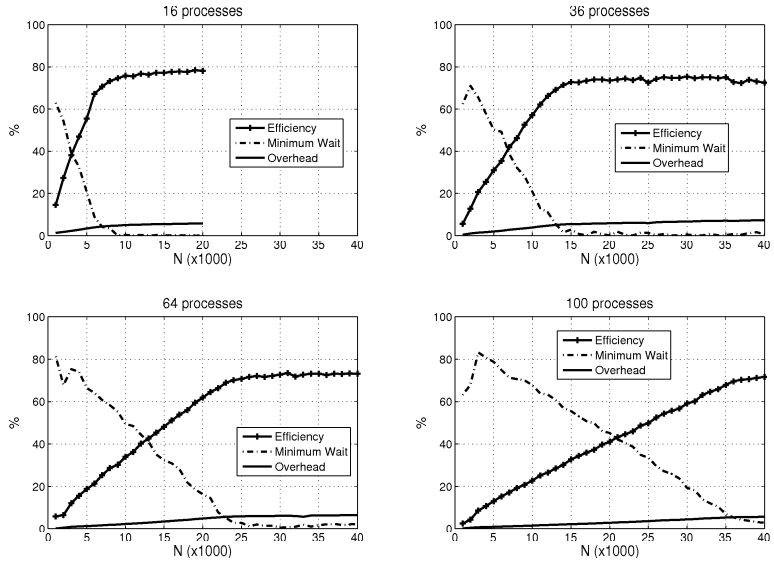

Fig. 5. Efficiency, overhead, and minimum synchronization overhead (Minimum Wait) for the more efficient dual wavefront implementation of `Sweep` ($n_b = 100$).

In Figure 4 (straightforward implementation), the relative efficiency comes close to the theoretical upper bound. The overhead is a few percent and the minimum wait is almost constant but still very significant. The straightforward implementation is not scalable and the reason is synchronization overhead. In Figure 5 (dual wavefront implementation), the relative efficiency peaks at around 70–80%. A strong negative correlation between the minimum wait and the efficiency shows that synchronization overhead is virtually eliminated by the dual wavefront algorithm for large enough problems.

4 Conclusions

We have presented a design of a framework for dynamic node-scheduling of blocked matrix computations on hybrid distributed memory machines. The framework uses an efficient runtime data dependence analysis method.Priority-based scheduling has been shown to extract much more of the available parallelism than standard FIFO scheduling. For example, comparing the timings in Figures 4 and 5 for 16 processes and $N = 20000$ (the largest problem solved on 16 processes), we get a speedup of 2.75 in favor for the wavefront implementation of `Sweep`. Going from 16 to 64 processes in Figure 5, we get another speedup of 3.32. We also see that $N = 20000$ is not large enough to reach practical peak on 64 processes.

References

1. B. Adlerborn, B. Kågström, and D. Kressner. Parallel Variants of the Multishift QZ Algorithm with Advanced Deflation Techniques . In B. Kågström et al., editors, *Applied Parallel Computing: State of the Art in Scientific Computing, PARA 2006*, Lecture Notes in Computer Science, LNCS 4699, pages 117–126. Springer, 2007.
2. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM.
3. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.
4. K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIAM J. Matrix Anal. Applics.*, 23:929–947, 2001.
5. R. Brightwell and K. D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proc. of the 18th annual international conference on Supercomputing*, pages 298–305, New York, NY, USA, 2004. ACM Press.
6. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Technical Report UT-CS-07-600, University of Tennessee at Knoxville, 2007. Also as LAPACK Working Note 191.
7. E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Super-Matrix 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 2007.
8. G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
9. R. Granat, D. Kressner, and B. Kågström. Parallel Eigenvalue Reordering in Real Schur Forms. *Concurrency and Computation: Practice and Experience*, submitted 2007. Also as LAPACK Working Note 192.
10. B. Kågström and D. Kressner. Multishift Variants of the QZ Algorithm with Aggressive Early Deflation. *SIAM J. Matrix Anal. Applics.*, 29:199–227, 2006.
11. MPI: A Message Passing Interface Standard. `http://www.mpi-forum.org/`, 1995.