Task-Queue Based Hybrid Parallelism: A Case Study

Karl Fürlinger[1], Olaf Schenk[2], and Michael Hagemann[2]

[1] Institut für Informatik,
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München
karl.fuerlinger@in.tum.de
[2] Departement Informatik,
Universität Basel
{olaf.schenk,michael.hagemann}@unibas.ch

Abstract. In this paper we report on our experiences with hybrid parallelism in PARDISO, a high-performance sparse linear solver. We start with the OpenMP-parallel numerical factorization algorithm and reorganize it using a central dynamic task queue to be able to add message passing functionality. The hybrid version allows the solver to run on a larger number of processors in a cost effective way with very reasonable performance. A speed-up of more than nine running on a four-node quad Itanium 2 SMP cluster is achieved in spite of the fact that a large potential to minimize MPI communication is not yet exploited in the first version of the implementation.

1 Introduction

Hybrid parallelism (i.e., the combined usage of shared memory and message passing programming paradigms) seems to be a perfect fit for the hierarchical organization of today's popular SMP cluster systems. In most previous reports on the merits of hybrid parallelism (e.g., [12, 2, 1, 8]), existing MPI applications have been extended with OpenMP shared memory programming constructs. Often the authors arrive at the conclusion that the performance of pure MPI programs is generally somewhat better than those of their hybrid counterparts. Still, a hybrid approach might be advantageous in certain circumstances, such as when the MPI code scales poorly or when replication of the dataset limits the number of MPI processes per node.

In this paper we report on our experiences with the *other* path to hybrid parallelism: We start with PARDISO, a shared memory parallel direct solver for large sparse systems of linear equations [7] that has recently been included into the Intel Math Kernel Library (MKL) [5]. In order to support message passing, we introduce a modified version of the OpenMP-parallel central numerical factorization algorithm that accounts for most of the solver's runtime. The modified version uses a dynamic task-queue instead of OpenMP's `parallel for` construct for work sharing. The introduction of message passing functionality is then straightforward by adding new tasks related to message passing.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 624–631, 2004.

The rest of the paper is organized as follows: Section 2 provides some general information about PARDISO and describes the current OpenMP implementation of the central numerical factorization algorithm. Section 3 describes the new task-queue based implementation of this algorithm which is the basis for the message passing extensions described in Section 4. Then, in Section 5, we compare the performance of the different incarnations of our implementation (sequential, pure OpenMP, pure MPI, and hybrid OpenMP+MPI) on a number of test matrices. Finally, Section 6 concludes and discusses future work.

2 The Sparse Direct Solver PARDISO

PARDISO is a high performance direct solver for sparse systems of linear equations which arise in application areas such as semiconductor device simulation.

In this work we are only concerned with PARDISO's supernodal[1] left-right looking numerical factorization algorithm that accounts for most of the runtime of the solver. For a comprehensive discussion of PARDISO please consult [9–11].

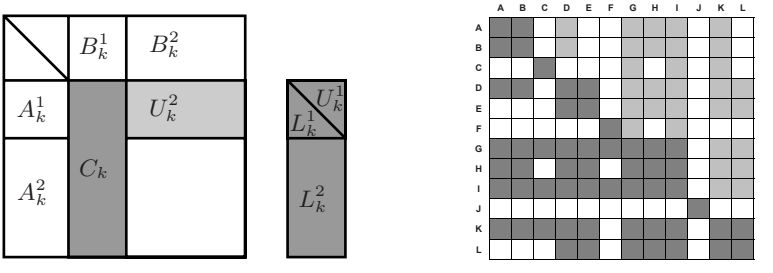

Fig. 1. Block-based LU factorization (left) and example sparse matrix with supernodes {A,B}, {C}, {D,E}, {F}, {G,H,I}, {J}, {K,L} (right).

To motivate the algorithm used by PARDISO consider the block-based LU factorization depicted in Fig. 1. In the k-th elimination step, the k-th block row and the k-th block column are computed by performing the following operations:

1. The external modifications ('*updates*') of the block columns of L and block rows of U:
$$C_k \leftarrow C_k - \begin{pmatrix} A_k^1 \\ A_k^2 \end{pmatrix} B_k^1, \quad U_k^2 \leftarrow U_k^2 - A_k^1 B_k^2. \tag{1}$$
2. The internal *factorization* of the diagonal block of C_k, to obtain the factors L_k^1 and U_k^1, followed by the internal factorization of the block columns of L and rows of U:
$$L_k^2 \leftarrow L_k^2 (U_k^1)^{-1}, \quad U_k^2 \leftarrow (L_k^1)^{-1} U_k^2. \tag{2}$$

[1] A supernode is defined as a group of consecutive rows/columns with similar non-zero structure.

Similar operations are performed in the supernodal approach used by PARDISO. However, instead of *one* external update involving all previously factored blocks (all blocks to the left), the update is split into several smaller supernode updates, of course only the non-zero supernodes need to be considered in this case. For example, in the sparse matrix shown in the right part of Fig. 1, supernode {D,E} is updated by {A,B} but not by {C}.

The original implementation of this algorithm is shown in Fig. 2 where nup(j) denotes the number of outstanding external updates on supernode j, $\mathcal{U}(j)$ is the set of all supernodes that are updated *by* supernode j and $\mathcal{L}(j)$ holds all 'ready' supernodes that update supernode j.

The algorithm is called left-right looking, because for the factorization of supernode j, all updates from 'left' supernodes are considered (i.e, those with smaller index) and as soon as the factorization of j is finished, all 'right' supernodes that are updated by j are informed that j is ready.

```
1: #pragma omp parallel for
2: for j = 1 ... N_supernodes do
3:     while nup(j) > 0 do
4:         wait for i ∈ L(j)
5:         L ← L\{i}
6:         perform j ← i supernode update
7:         nup(j) ← nup(j) −1
8:     end while
9:     for all k ∈ U(j) do
10:        L(k) ← L(k) ∪ {j}
11:    end for
12: end for
```

Fig. 2. PARDISO's original factorization algorithm.

It is not obvious how the algorithm shown in Fig. 2 can be extended with message passing functionality. The easiest way would be to use one OpenMP thread for MPI communication that could not participate in the main worksharing `for` loop. Our approach is instead the re-organization of the algorithm using a central task-queue as described in the next section. Sending and receiving messages can then be handled quite easily by adding new task types related to message passing.

3 Task-Queue Based Re-organization

The central data structure of the new OpenMP version is a task-queue (a linked list of task descriptors). Fig. 3 shows a statechart of the new algorithm. The task-queue is initialized with all supernodes that are ready to be factored (i.e., those that do not require any external updates). Then, until all supernodes are

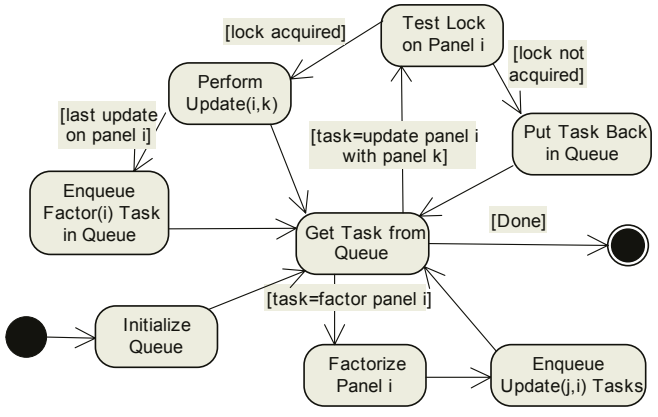

Fig. 3. Statechart of the new OpenMP factorization algorithm.

factored, each thread fetches a task descriptor from the head of the queue and performs the associated action.

There are two types of tasks: FACTOR(j) is the internal factorization of the supernode j. After the factorization of supernode j is finished, UPDATE(k,j) tasks are enqueued for all supernodes k that are updated by supernode j.

The UPDATE(k,j) tasks are the second type of tasks. A thread that performs an UPDATE(k,j) task checks if there are further outstanding UPDATE(k,$\cdot$) tasks. If this is not the case, the supernode is ready to be factored and a FACTOR(k) task is enqueued.

In contrast to the original version, the new OpenMP version has the advantage of better load balancing since the fine grained tasks are dynamically distributed to the whole thread set. A potential disadvantage is the increased requirement for thread synchronization. Access to the task-queue is protected by using OpenMP's critical section construct. Furthermore, concurrent updates to the same supernode by different threads are avoided by using OpenMP locks.

Table 1 shows a performance comparison between the original and the new OpenMP version for the six test matrices listed in Tab. 2. Evidently, the new version is about ten percent slower for two threads and around 20 percent slower for four threads. The difference can be attributed to the increased synchronization requirements mentioned and a less advantageous order of the supernode updates with respect to cache misses.

4 Adding Message Passing Functionality

It is straightforward to add message passing functionality to the implementation described in section 3. The statechart of the hybrid version is shown in Figure 4, the new states related to MPI are shown in grey. Each MPI process consists of a number of OpenMP threads and "owns" a number of supernodes, implying its responsibility to perform all external updates on those supernodes as well

Table 1. Factorization time in seconds for the original and the new OpenMP algorithm for two and four threads on a quad Intel Itanium 2 SMP system. Sequential denotes the sequential factorization time of the new algorithm.

	Matrix	1	2	3	4	5	6
Sequential		64.17	158.60	116.32	109.93	398.65	314.12
2 Threads	Original Version	32.58	76.41	56.02	53.05	192.88	153.30
	New Version	36.12	84.09	62.28	58.55	210.48	165.04
	Ratio	1.11	1.10	1.11	1.10	1.09	1.08
4 Threads	Original Version	17.91	41.24	30.44	28.78	103.23	82.91
	New Version	21.65	48.75	36.59	34.66	119.18	93.61
	Ratio	1.21	1.18	1.20	1.20	1.15	1.13

as the internal factorization. The assignment of supernodes to MPI processes is currently a simple static round-robin assignment.

Similar to the OpenMP version, the task-queue of each process in initialized with the supernodes that are ready to be factored. However, in the MPI version, each MPI process only adds the supernodes it owns. Furthermore, when process P_1 factors supernode k it enqueues only those UPDATE(j,k) tasks that affect the supernodes j owned by P_1. In addition P_1 sends the supernode k (i.e., the L and U factors and pivoting data) to all other MPI processes that own supernodes that are updated by k.

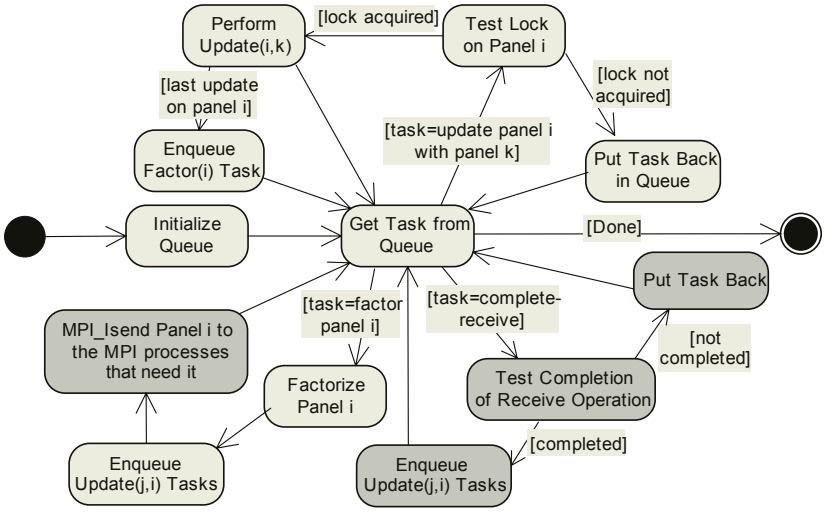

Fig. 4. Statechart of the hybrid OpenMP/MPI factorization algorithm.

To overlap communication with computation, asynchronous MPI operations are used. When an MPI process P_1 transfers a supernode to process P_2 it first sends an integer denoting the affected panel. Then it posts asynchronous send (`MPI_Isend()`) operations that transfer the actual panel data (the L and U factors and pivoting data).

On the receiving side, on each iteration of the main work loop, a thread first checks for incoming MPI messages. If an `MPI_Iprobe()` indicates an incoming supernode j, the thread posts corresponding asynchronous receive operations (`MPI_Irecv()`) for the L and U factors of the supernode and for the pivoting data. It then creates a COMPLETE_RECEIVE(j) task and places it on the task-queue. Then the thread proceeds as usual with its iteration of the work loop.

When a thread fetches a COMPLETE_RECEIVE(j) task from the queue, it uses `MPI_Testall()` to check whether all asynchronous operations related to panel j have finished. If this is not the case, the task is put back in the queue. Otherwise, all local UPDATE($\cdot$,j) tasks are enqueued just as if the panel would have been factored locally.

5 Results

The new OpenMP approach and the message passing extensions have been implemented in a way that allows us to derive four different versions of the application from the same source code. `seq` is the sequential version, `omp.`a is the pure OpenMP version with a threads, `mpi.`a (n) is a pure MPI version with a MPI processes on each of n nodes and `hyb.`a`:`b (n) is the hybrid version running on n nodes with a MPI processes on each node that consist of b OpenMP threads.

The programs were tested on an Itanium 2 cluster with four nodes. Each node ist built up of four Itanium 2 ('Madison') Processors with 1.3 GHz, 3 MB third level cache and 8 GB of main memory. The nodes are connected by Mellanox 4x Infiniband [3] HCAs. We used MVAPICH (based on MPICH 1.2.5, OSU patch 0.9.2) and the Intel C and Fortran compilers in version 7.1. The tests were performed with the matrices shown in Tab. 2.

Table 2. The test matrices. N is the order of the matrix, nnz is the total number of non-zeros and n_{super} is the number of supernodes identified by the pre-processing stages of PARDISO.

Number	Matrix	N	nnz	n_{super}
1	3D_75932_sef3D	75932	88,528,964	23,112
2	barrier2-4	113076	175,824,468	22,352
3	matrix-ibm-watson	125329	147,872,909	27,825
4	matrix_9	103430	129,396,404	23,447
5	ohne1	181343	417,744,155	24,099
6	para-2	153226	287,734,810	29,781

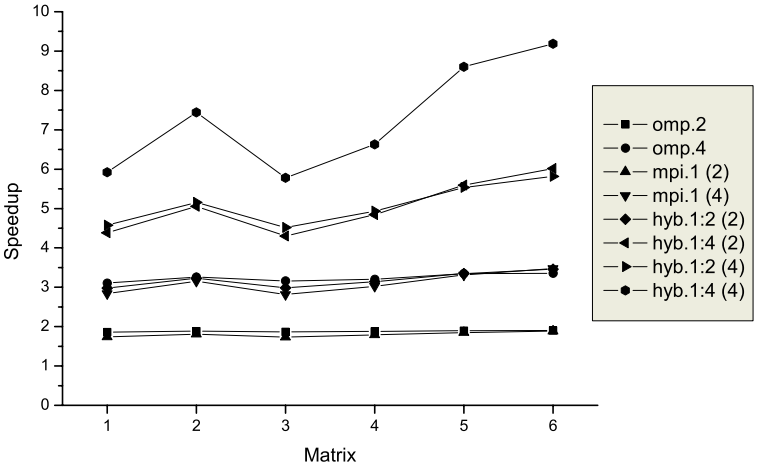

Fig. 5. Speedup relative to the sequential version.

Fig. 5 shows the speedup of various versions of our programs relative to the sequential version. The following conclusions can be derived:

- The MPI version is usually slightly slower than the OpenMP version on the same number of processors, the hybrid version is in between.
- The larger processor configurations perform better on the larger matrices.
- The `hyb.1:2 (4)` variant is usually slightly faster than the `hyb.1.4 (2)` version. This indicates that the decreased memory bus load (for two threads on one node) outweighs the additional message passing requirements.
- A speedup of more than nine can be achieved using a hybrid version running on all 16 processors.
- MPI variants with more than one MPI process on a node (not shown in Fig. 5) usually perform worse than variants with only one MPI process per node (e.g., `mpi.1 (4)` vs. `mpi.2 (2)`). This is due to the large memory requirement of the program (more than five GB for the largest matrices) and the fact that all data is replicated by the MPI processes.
- We additionally measured the communication time using mpiP [6]. Depending on the particular input matrix, the total aggregated time spent in MPI functions is in the range of 5–9%, 7–15% and 10–20% for 2, 3 and 4 MPI processes, respectively. The number is typically smaller for larger matrices and higher for the smaller exemplars. The most MPI time (around 30%) is spent in the `MPI_Iprobe` function (which is called very frequently). The receive operations for the panel (despite being asynchronous) and the `MPI_Testall` to check whether the panel has been transferred also contribute significantly.

6 Conclusion and Future Work

We have presented our experiences with the hybridization of an application using OpenMP work-sharing constructs. Our approach was to re-organize the application to use a central task-queue instead of the work-sharing constructs. This yields a very flexible solution and it is then straightforward to add message-passing extensions.

Adding message passing capabilities allows the solver to run on a much larger number of processors beyond the confines of a single SMP system. We have shown that reasonable performance can be achieved with a speedup of more than nine on 16 processors.

A number of optimizations have not yet been implemented in the current version of our hybrid code. Most notable, PARDISO already uses METIS [4] for graph-partitioning in its two-level scheduling approach [9] to optimize cache misses. This partitioning will be very effective to minimize the number of messages sent between processes once it is included in the hybrid version.

References

1. Edmond Chow and David Hysom. Assessing performance of hybrid MPI/OpenMP programs on SMP clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001. submitted to J. Parallel and Distributed Computing.
2. D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2000.
3. Infiniband Trade Association. `http://www.infinibandta.org/home`.
4. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
5. Intel Math Kernel Library. `http://www.intel.com/software/products/mkl/beta/features.htm`.
6. mpip: Lightweight, scalable mpi profiling. `http://www.llnl.gov/CASC/mpip/`.
7. Pardiso website. `http://www.computational.unibas.ch/computer_science/scicomp/software/pardiso/`.
8. Rolf Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *Proc. 45th Cray Users's Group (CUG) Meeting*, May 2003.
9. Olaf Schenk and Klaus Gärtner. Two-level scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems. *Parallel Computing*, 28:187–197, 2002.
10. Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 2003.
11. Olaf Schenk, Klaus Gärtner, and Wolfgang Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.
12. Lorna Smith. Mixed mode MPI/OpenMP programming. Technical Report EH9 3JZ, Edinburgh Parallel Computing Centre, 2000.