A User's Experience with Parallel Sorting and OpenMP

Michael Süß and Claudia Leopold
University of Kassel
Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
{msuess, leopold}@uni-kassel.de

Abstract

Some algorithmic patterns are difficult to express in OpenMP. In this paper, we use a simple sorting algorithm to illustrate problems with recursion and the avoidance of busy waiting. We compare several solution approaches with respect to programming expense and performance: stacks, nesting and a workqueue (for recursion), as well as condition variables and the sched_yield–function (for busy waiting). Enhancements of the OpenMP–specification are suggested where we saw the need. Performance measurements are included to backup our claims.

1. Introduction

Parallel Programming is still a challenging task in our days. Although there are many powerful parallel programming systems, most of them operate on a relatively low abstraction level (e.g. *POSIX threads*, *MPI*). The specification of OpenMP [2] promised advances in this regard, and has provided a relatively smooth way to incrementally parallelize existing programs as well as to write powerful new applications on a high abstraction level, since its introduction in 1997. Its portability and vendor acceptance quickly helped the system to become a de facto standard for programming shared memory parallel machines.

Nevertheless, OpenMP is not without problems and rough edges, which version 2.0 of the specification was not able to straighten out fully. This paper uses a simple sorting algorithm to show some of these, as well as a couple of techniques developed to work around them. All examples were written in the *C++* programming language. Some suggestions for future enhancements to the specification are included.

The next section gives a short summary of the used sorting algorithm. Section 3 describes the two problematic areas that we encountered during the course of our work: *recursion* and *busy waiting*. Figure 1 depicts a diagram that

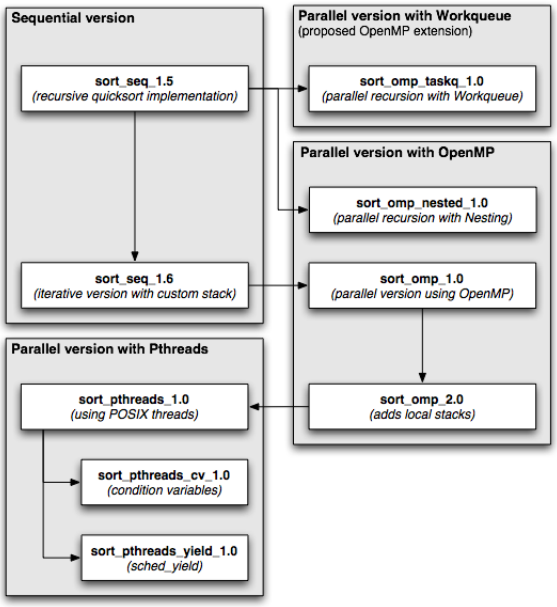

Figure 1. Revision tree of Sorting Program

illustrates the history of the different program versions we wrote to solve these problems (some less important versions are left out in the diagram, causing missing entries in the numbering scheme). Three different solution strategies are presented for recursion:

- an iterative approach (*sort_omp_1.0*)
- an advanced iterative version using local stacks (*sort_omp_2.0*)
- nested parallelism (*sort_omp_nested_1.0*)
- a *workqueue* (*sort_omp_taskq_1.0*)

The problem of busy waiting is approached with solutions borrowed from *POSIX threads*: *condition variables* (*sort_pthreads_cv_1.0*) and the *sched_yield*–function

(*sort_pthreads_yield_1.0*). Ideas for extensions to the OpenMP–specification presented in section 3 are the addition of an *omp_get_num_all_threads*–function, condition variables and an *omp_yield*–function. Note that at this point only data points are given for the inclusion of these extensions into the specification, as this paper merely describes our experiences with a certain problem area and does not include a complete proposal. Section 4 discusses the performance of our solutions. In section 5, we sum up our results and discuss further perspectives.

2. Sorting

Sorting data has always been one of the key problems of computer science. Many sequential algorithms have been suggested, of which *Quicksort*, invented and described by Hoare [4], is one of the most popular ones. It works recursively using the *divide and conquer* principle:

1. choose a pivot element, usually by just picking the last element out of the sorting area
2. iterate through the elements to be sorted, moving numbers smaller than the pivot to a position on its left, and numbers larger than the pivot to a position on its right, by swapping elements. After that the sorting area is divided into two subareas: the left one contains all numbers smaller than the pivot element and the right one contains all numbers larger than the pivot element.
3. goto step *1* for the left and right subareas (if there is more than one element left in that area)

The algorithm has an average time complexity of $O(n \cdot log(n))$ and a worst case time complexity of $O(n^2)$, making it one of the fastest sequential sorting algorithms available. Since we do not aim at producing the fastest parallel sorting algorithm ever, but merely try to show some problems and solutions with OpenMP, we chose this easy and widely used algorithm as the basis for our experiments, instead of a more advanced and complex (sequential or parallel) one. Previous experiments (although with a different focus) with quicksort and OpenMP have been conducted by Parikh [6].

3. A simple sorting algorithm and its problems

The base version for our tests was the simple quicksort algorithm described above, combined with *insertion sort* for small sorting areas. This algorithm (*sort_seq_1.5*) is sketched in Figure 2. We tried to speed up the algorithm through advanced (sequential) sorting techniques, but the performance gain was not worth the loss in simplicity.

Using this algorithm, we started our experiments with OpenMP, and soon after were confronted with our first problem: *recursion*.

```
template < typename T >
void myQuickSort(std::vector <T> &myVec, int q, int r)
{
        T pivot;
        int i, j;

        /* only small segment of numbers left -> use insertion sort! */
        if (r - q < SWITCH_THRESH) {
                myInsertSort(myVec, q, r);
                return;
        }

        /* choose pivot, initialize borders */
        pivot = myVec[r];
        i = q - 1;
        j = r;

        /* partition step, which moves smaller numbers to the left
                and larger numbers to the right of the pivot */
        while (true) {
                while (myVec[++i] < pivot);
                while (myVec[--j] > pivot);
                if (i >= j) break;
                std::swap(myVec[i], myVec[j]);
        }
        std::swap(myVec[i], myVec[r]);

        /* recursively call yourself with new subsegments,
                i is index of pivot */
        myQuickSort(myVec, q, i - 1);
        myQuickSort(myVec, i + 1, r);
}
```

Figure 2. Base version of the quicksort algorithm

3.1. Problem 1: Recursion

There is no easy and intuitive way to deal with recursion in OpenMP (yet), as the basic worksharing constructs provided by the specification (*for* and *sections*) do not seem to be well suited for recursive function calls. Our first solution to the problem involved getting rid of the recursion, as already suggested by an Mey [1]. It is a widely known fact that, by using a stack, every recursion can be changed into an iteration, and that is exactly what we did in *sort_seq_1.6*. This step was one of the most time consuming of all, as it involved the introduction of a new data structure, specially tailored to our problem. We called this structure *globalToDoStack*, it stores the intervals still to be sorted.

This program was later parallelized into *sort_omp_1.0*, making the following changes:

- a *parallel* region was added in *main*, around the first call of the *myQuickSort*–function
- a call to *omp_get_thread_num* was added, so that only one thread initially executes the *myQuickSort*–

function, all others wait until work for them is put on the *globalTodoStack*

- all accesses to shared variables (especially the *globalTodoStack*) were protected by *critical* sections to prevent multiple threads from accessing the variables at the same time

The resulting program is sketched in Figure 3.

In *sort_omp_2.0* the global stack was complemented with one local stack per thread. All new segments to be sorted are put on the local stacks per default. Only when one of the local stacks is empty, it needs to communicate with the global stack and poll for new work there. It works the other way around as well: when the global stack is getting empty, new work is pushed on it from a local stack. This modification led to a significant performance gain, because the need for synchronisation via critical sections dropped considerably. The effect is most visible for high numbers of working threads, because that is when a lot of work sharing needs to be done.

Our second solution to the recursion problem (*sort_omp_nested_1.0*) involves nested parallelism as illustrated in Figure 4. There are a few problems with this approach though. First of all, the OpenMP–specification allows compilers to serialize nested parallel directives, and many still do so. With these compilers, the version will not achieve considerable speedups, performance portability is not granted. Furthermore, nesting support in the specification seems to be a little immature. In particular, the *omp_get_num_threads*–function is useless for our example, as it always returns two threads per parallel region. The value we were really interested in is the number of all running threads (to limit the creation of new threads as we dive deeper into the recursion). Since this number is not available through a simple function call, we needed to track it ourselves, therefore introducing new (and performance hindering) critical sections into the code. Perhaps this could be taken care of by the introduction of an *omp_get_num_all_threads* function call or some similar mechanism into the next iteration of the specification.

The third solution to the recursion problem has been first suggested by Shah et al. [7]. It involves usage of the *workqueuing model* (*sort_omp_taskq_1.0*, depicted in Figure 5). This model has been proposed as an OpenMP–extension, but has not been accepted yet. To the authors knowledge, only two compilers understand the new pragmas, and so this solution has the drawback that the code presented here is not portable. This might change quickly though, if the *workqueuing*–proposal is accepted. Except for this drawback, the solution is easy and elegant. This version lacks the ability to add *local queues* (similar to the local stacks that brought a performance improvement in *sort_omp_2.0*), but a smart compiler might recognize this

```
template < typename T >
void myQuickSort(std::vector <T> &myVec, int q, int r,
      std::stack <std::pair <int, int>>&globalTodoStack,
      int &numBusyThreads, const int numThreads)
{
     bool idle = true;

     /* Skipped: Initialisation */

     while (true) {

          /* only small segment of numbers left ->use insertion sort! */
          if (r - q < SWITCH_THRESH) {
               myInsertSort(myVec, q, r);
               /* and mark the region as sorted, by setting q to r */
               q = r;
          }

          while (q >= r) {   /* Thread needs new work */

               /* only one thread at a time should access the
                  globalTodoStack, numBusyThreads and idle variables */
               #pragma omp critical
               {
                    /* something left on the global stack to do? */
                    if (false == globalTodoStack.empty()) {
                         if (true == idle) ++numBusyThreads;
                         idle = false;
                         /* Skipped: Pop a new segment off the stack */
                    } else {
                         if (false == idle) --numBusyThreads;
                         idle = true;
                    }
               }

               /* if all threads are done, break out of this function.
                  note, that the value of numBusyThreads is current, as there
                  is a flush implied at the end of the last critical section */
               if (numBusyThreads == 0) {
                    return;
               }
          }

          /* Skipped: choose pivot and do partitioning step */

          #pragma omp critical
          {
               globalTodoStack.push(pair(q, i - 1));
          }

          /* iteratively sort elements right of pivot */
          q = i + 1;
     }
}

int main(int argc, char *argv[])
{
     /* Skipped: Program Initialisation */

     #pragma omp parallel shared(myVec, \
       globalTodoStack, numThreads, numBusyThreads)
     {
          /* start sorting with one thread, the others wait for the stack to fill up */
          if (0 == omp_get_thread_num()) {
               myQuickSort(myVec, 0, myVec.size() - 1,
                 globalTodoStack, numBusyThreads,
                 numThreads);
          } else {
               myQuickSort(myVec, 0, 0,
                 globalTodoStack, numBusyThreads,
                 numThreads);
          }
     }
     /* Skipped: Tests and Program output */
}
```

Figure 3. Parallel part of sort_omp_1.0

```
template < typename T >
void myQuickSort(std::vector < T > &myVec, int q,
        int r, int &numBusyThreads, const int numThreads)
{
    /* Skipped: Initialisation + Partitioning step */

    /* do not nest, if there are too many threads already */
    if (numBusyThreads >= numThreads) {
        myQuickSort(myVec, q, i - 1, numBusyThreads,
          numThreads);
        myQuickSort(myVec, i + 1, r, numBusyThreads,
          numThreads);
    } else {
        #pragma omp atomic
        numBusyThreads += 2;

        #pragma omp parallel shared(myVec, numThreads, \
         numBusyThreads, q, i, r)
        {
            #pragma omp sections nowait
            {
                #pragma omp section
                {
                    myQuickSort(myVec, q, i - 1,
                      numBusyThreads, numThreads);
                    #pragma omp atomic
                    numBusyThreads--;
                }
                #pragma omp section
                {
                    myQuickSort(myVec, i + 1, r,
                      numBusyThreads, numThreads);
                    #pragma omp atomic
                    numBusyThreads--;
                }
            }
        }
    }
}
```

Figure 4. Parallel part of sort_omp_nested_1.0

```
template < typename T >
void myQuickSort(std::vector <T> &myVec, int q, int r)
{
    /* Skipped: Initialisation + Partitioning step */

    #pragma omp taskq
    {
        #pragma omp task
        {
            myQuickSort(myVec, q, i - 1);
        }
        #pragma omp task
        {
            myQuickSort(myVec, i + 1, r);
        }
    }
}

int main(int argc, char *argv[])
{
    /* Skipped: Program Initialisation */

    #pragma omp parallel shared (myVec)
    {
        #pragma omp taskq
        {
            #pragma omp task
            {
                myQuickSort(myVec, 0, myVec.size() - 1);
            }
        }
    }

    /* Skipped: Tests and Program output */
}
```

Figure 5. Parallel part of sort_omp_taskq_1.0

performance potential and insert them automatically.

Having solved the recursion problem, the next one appeared: *busy waiting*.

3.2. Problem 2: Busy Waiting

When a thread has nothing to do in our sorting application (because both its local stack and the global stack are presently empty, e.g. at the beginning), it does not mean that it is allowed to quit. New tasks can be put on the global stack at any time and should of course be processed as soon as possible. The only way this can be accomplished with OpenMP is *busy waiting*, which means that the thread is constantly polling for work, wasting processor cycles that could be better spent in another thread.

For possible solutions, one must only look as far as to the *POSIX threads* standard. A synchronisation primitive called *condition variable* is implemented there and solves the problem in a quite hard to understand way (when one is looking at it through the eyes of a beginner to parallel programming), but nevertheless fully. Butenhof describes condition variables like this:

> *A condition variable is a "signaling mechanism" associated with a mutex and by extension is also associated with the shared data protected by the mutex.* Waiting *on a condition variable atomically releases the associated mutex and waits until another thread* signals *(to wake up one waiter) or* broadcasts *(to wake all waiters) the condition variable. The mutex must always be locked when you wait on a condition variable and, when a thread wakes up from a condition variable wait, it always resumes with the mutex locked.* ([3, p. 72])

The introduction of this concept into the OpenMP–specification has already been suggested by Lu et al. [5], and we would also like to encourage the inclusion of this or a similar mechanism into the specification. To illustrate the savings possible when using condition variables, we ported our sorting application to *POSIX threads* (staying as close to the original version as possible). The resulting program is *sort_pthreads_1.0*, which should perform about equal to *sort_omp_2.0* (see section 4 for details). Then, *sort_pthreads_1.0* was enhanced with condition variables (*sort_pthreads_cv_1.0*). Every time a thread finishes

his work and finds nothing else to do on the stacks, it puts itself to sleep and is woken up by another thread only when there is new work to be done. This may lead to a significant performance gain on a heavily loaded machine.

A second and somewhat easier solution to the problem of busy waiting can be observed in *POSIX.1b* (realtime extensions). This standard defines a function *sched_yield*, which puts the calling thread at the end of the *ready–queue* of the operating system scheduler and selects a new thread to run. If there is no other thread waiting, the function returns immediately. The same could be done with a new function *omp_yield* in OpenMP. Whenever a thread runs out of work in our example program and the stacks are empty, it calls the suggested function. If other threads are waiting to be processed (which might not be out of work yet), these get a chance to run and produce more work for all idle threads. Though less powerfull than condition variables, our experiments with *sort_pthreads_yield_1.0* (a version incorporating *sched_yield*) suggest that this function is able to reduce busy waiting under heavy load for our problem as well (see Table 3).

4. Performance results

Performance tests were carried out on an otherwise unloaded node of an *AMD Opteron 848* class computer with 4 processors at 2.2 GHz, located at the RWTH Aachen. Programs were compiled with the *Intel C++ Compiler 8.1* using the *-O3 -openmp* compiler options. Further experiments were carried out on an otherwise unloaded node of a *Sun Fire 6800* class computer with a maximum of 8 *Ultra Sparc III* processors at 900MHz, also located at the RWTH Aachen. Here, the *Guide Compiler* by *Kuck & Assoc. Inc. (KAI)* was used with options: *-fast –backend -xchip=ultra3cu - -backend -xcache=64/32/4:8192/512/2 - -backend -xarch=v8plusb*.

Tables 1 and 2 show wall–clock time in seconds for all versions of our sorting program, with different numbers of threads. Only the time needed to actually perform the sorting algorithms was measured. All experiments were repeated at least three times, each time sorting 100 million random integers. For Tables 1 and 2, the best time achieved in each test was chosen.

Table 1 shows good speedups for all parallel program versions. The best performing solutions are *sort_omp_2.0* and the programs using Pthreads. Program *sort_omp_2.0* outperforms *sort_omp_1.0*, which shows the relevance of the local stacks. The programs with nesting and the workqueue are slower than the iterative programs, but this might be due to the relative immaturity of both options in the specification. The results for *sort_omp_nested_1.0* are to be taken with a grain of salt, since we were not able to fully control the number of threads used. Nevertheless, the best result we were able to achieve when running with different numbers of threads was 11.7 seconds, which is still slower than the other programs. No difference in speed is noticeable between the different versions using Pthreads. This is to be expected, as the advantages of these solutions will only show on heavily loaded systems or when looking at the CPU–time.

Program	Wall–clock time (sec.)		
	1Th.	2Th.	4Th
sort_seq_1.5	23.8	23.8	23.8
sort_seq_1.6	23.6	23.6	23.6
sort_omp_1.0	24.0	13.7	8.1
sort_omp_2.0	24.3	12.6	7.5
sort_omp_nested_1.0	23.9	21.4	12.4
sort_omp_taskq_1.0	29.8	16.3	9.1
sort_pthreads_1.0	24.0	12.7	7.5
sort_pthreads_cv_1.0	24.8	12.9	7.6
sort_pthreads_yield_1.0	24.5	12.7	7.5

Table 1. Wall–clock time for sorting 100 million integers on an AMD Opteron 2200 in seconds

Program	Wall–clock time (sec.)			
	1Th.	2Th.	4Th.	8Th.
sort_seq_1.5	36.8	36.8	36.8	36.8
sort_seq_1.6	37.4	37.4	37.4	37.4
sort_omp_1.0	38.2	23.9	15.7	11.0
sort_omp_2.0	37.9	21.4	13.1	10.0
sort_omp_nested_1.0	43.4	25.2	25.2	25.2
sort_omp_taskq_1.0	n.A.	n.A.	n.A.	n.A.
sort_pthreads_1.0	37.6	21.2	13.3	10.1
sort_pthreads_cv_1.0	37.2	20.3	13.3	9.6
sort_pthreads_yield_1.0	37.2	22.1	14.2	10.9

Table 2. Wall–clock time for sorting 100 million integers on a Sun Fire 6800 in seconds

The results in Table 2 look similar, except for two differences:

- nested parallel regions are serialized by the *Guide Compiler*, therefore no speedup beyond 2 is possible for *sort_omp_nested_1.0*
- we were not able to get *sort_omp_taskq* to work reliably on this platform (we have no idea if this is a compiler problem or a subtle bug in our implementation, but as soon as more than one thread was employed, it would either crash or run forever), therefore no results for this platform are provided

Table 3 demonstrates what happens on a heavily loaded

Program	SUN / 96Th.	AMD / 16Th.
sort_omp_1.0	> 600	20.5
sort_omp_2.0	> 600	19.0
sort_pthreads_1.0	15.9	7.8
sort_pthreads_cv_1.0	10.6	7.7
sort_pthreads_yield_1.0	11.4	7.9

Table 3. Average wall–clock time for sorting 100 million integers on heavily loaded systems in seconds

system with and without *busy waiting*. The heavy load was built up by using four times as many threads as there are processors available on the machine. It shows average wall–clock time to reduce the chance of lucky scheduling decisions.

On the SUN platform, the Pthreads solutions show the results we expected: Program *sort_pthreads_1.0* takes considerably longer than *sort_pthreads_cv* and *sort_pthreads_yield*. There are two relatively big surprises for us though: First, the OpenMP–versions are slow as compared to the *Pthreads*–versions (in case of the SUN machine so slow that we decided to cancel the runs after 10 minutes). The reasons for this are not yet clear to us and still under investigation. Second, on the *AMD*–machine, no performance difference is noticeable between the different *Pthreads*–versions. A better scheduler might account for this, but we are still investigating this question as well.

5. Concluding remarks and perspectives

In this paper, we have used several versions of a simple parallel sorting program to show some weaknesses of the OpenMP–specification and a couple of ways to address them. Suggestions for enhancements to the specification were made whenever it seemed appropriate.

Section 3.1 addressed the problem of *recursion*. Three solutions to it were presented, only one of which could be portably implemented with the present state of the specification, while the other, more elegant but less performant, two required additional support for the *workqueue* extension, or for nested parallelism, respectively.

In section 3.2, we have discussed the problem of wasted processor cycles (*busy waiting*). As possible solutions we suggest the introduction of *condition variables* and a new function *omp_yield*. Performance measurements showed that both approaches may provide adequate savings in processor time.

In the future, we plan to implement and test some of the ideas and additions to the specification we have suggested into an actual compiler, as well as investigate other algorithmic problems beyond sorting.

6. Acknowledgments

We are grateful to Björn Knafla and Beliz Senyüz for proofreading the paper. We thank the *Center for Computing and Communication* (especially Dieter an Mey) at the RWTH Aachen and the *University Computing Center* at the Technical University of Darmstadt for providing the computing facilities used to carry out our performance measurements.

References

[1] D. an Mey. Two OpenMP programming patterns. In *Proceedings of the Fifth European Workshop on OpenMP - EWOMP'03*, September 2003.

[2] O. A. R. Board. OpenMP specifications. `http://www.openmp.org/specs`.

[3] D. R. Butenhof. *Programming with POSIX Threads*. Addison–Wesley, 1997.

[4] C. Hoare. Quicksort. *The Computer Journal*, 5:10–15, 1962.

[5] H. Lu, C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In *Proc. of Supercomputing'98*, 1998.

[6] R. Parikh. Accelerating quicksort on the intel pentium 4 processor with hyper–threading technology. www.intel.com/cd/ids/developer/asmo-na/eng/technologies/threading/hyperthreading/20372.htm, 2003.

[7] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP - EWOMP'02*, September 2002.