Thread based OpenMP for nested parallelization

Ragnhild Blikberg[a]* and Tor Sørevik[b]†

[a]Parallab, BCCS, UNIFOB, University of Bergen, NORWAY

[b]Dept. of Informatics, University of Bergen, NORWAY

Many computational problems have multiple layers of parallelism. Here we argue and demonstrate that utilizing multiple layers of parallelism may give much better scaling than if one restrict oneself to only one level of parallelism.

Two important issues in this paper are *load balancing* and *implementation*. We provide an algorithm for finding good distributions of threads to outer-level tasks, when these are of uneven size, and discuss how to implement nested parallelism in OpenMP. Two different implementation strategies will be discussed: The preferred one, directive based parallelization, versus explicit thread programming.

Keywords: OpenMP, SMP-parallelism, nested parallelism

1. Introduction

Computational sciences have made quantum leaps forward over the past decades, due to impressive progress in algorithms, computer hardware and software. This has open up the possibility to tackle very complex problems by the assistance of computer simulations, and as more problems are being attacked the appetite for more compute power and more efficient methods seems to increase rather than decrease.

As the computational problems get larger their complexity also grows. A standard way to handle the increased complexity is to decompose the problem. This can be done functionally by splitting the problem in smaller problems, or *tasks*, to be solved independently or by domain splitting, dividing the computational domain into smaller parts. Next these subproblems are solved independently before they eventually are glued together. Such splitting provide an obvious coarse-grained parallelism. This parallelism has to be utilized. However, in many cases this easily available outer-level parallelism is not enough as it may consist of only (relatively) few tasks, and they may often be of unequal size. Thus parallelism has to be applied within each task as well.

In this paper we address the challenge of how to implement such multi-level parallelism in OpenMP. We also briefly describe how to allocate threads to tasks in order to achieve a good load balance.

The paper is organized as follows: The question of allocating threads to tasks is addressed in Section 2, where we give an algorithm which distributes threads to tasks. In Section 3, we discuss how to implement 2-level parallelism in OpenMP. The effect of 2-level parallelism on a real application, a wavelet based data compression code, has been tested and is reported in Section 4. Finally, the conclusions will be given.

2. The distribution algorithm

In this section we will present an algorithm for distributing threads to tasks. Different situation may occur depending on the number and sizes of tasks and the number of available threads. One immediately identifies two extreme cases; The case where all tasks need at least one thread, and the case where each thread should take one or more tasks. We first introduce algorithms for these two extreme cases, and then show how they can be combined to deal with all possible cases.

*http://www.ii.uib.no/~ragnhild
†http://www.ii.uib.no/~tors

Case 1: All tasks need at least one thread

The threads should be grouped together in teams, where each team is responsible for doing the work associated with one task. The distribution of threads to tasks can be done in the following way: First assign one thread to each task. Then find the task with highest 'work-to-thread-ratio' and assign an extra thread to this task. Repeat until all threads are assigned to a task.

A detailed description and analysis of this algorithm is given in [5]. It needs as input P (the number of threads available), N (the number of tasks) and a vector $\mathbf{w}$ of size N, giving the work-load estimate for the N tasks. The output is a vector $\mathbf{p}$ of size N containing the number of threads assigned to each task. This correspond to the following function call:

$$[\mathbf{p}] = \mathbf{Distribute1}(N, P, \mathbf{w})$$

Case 2: All threads do at least one task

The other extreme is when the tasks are many and small such that most tasks have to share one thread and under no circumstances will any task have more than one thread. In some sense this is the dual of the previous problem. Here we will assign tasks to threads.

This subproblem correspond to a bin-packing problem. One version of the problem is to assume we have a fixed number of bins (in our case threads) and seek to pack each bin such that the size of the largest bin is minimized. Alternatively, we can assume that the maximum bin-size is fixed and try to minimize the number of bins needed to pack all tasks.

In either case the problem is known to be NP-complete, and a good approximate solution can be found by the *best fit decreasing* method. For detailed description and analysis of the best fit decreasing algorithm see [2]. There are different advantages and problems with the two formulations, but these will not be discussed here.

In our computation we have chosen the first formulation. The output of the algorithm is the vector **task_to_thread** of size N, telling which thread a task is distributed to. This correspond to the function call:

$$[\mathbf{task\_to\_thread}] = \mathbf{Distribute2}(N, P, \mathbf{w})$$

Our distribution algorithm works by simply sorting the tasks in two sets; one for those tasks large enough to rightfully ask for at least one thread for themselves, and a second set consisting of tasks so small that they must be prepared to share a thread.

Algorithm 1: The distribution algorithm

/ This algorithm finds a distribution of the work of N tasks to P threads.*
The tasks are divided in 2 sets, $\mathcal{L}$ ("large" tasks) and $\mathcal{S}$ ("small" tasks),
where one or more threads are working on the same task in $\mathcal{L}$,
*and where each thread has one or more tasks to work on in $\mathcal{S}$. */*

$[\mathcal{L}, \mathcal{S}, \mathbf{p}, \mathbf{task\_to\_thread}] = \mathbf{Distribute\_All}(N, P, \mathbf{w})$

$W = \sum_{i=1}^{N} \mathbf{w}_i$; $\quad \bar{w} = W/P$
$\mathcal{L} = \{\forall i; \mathbf{w}_i > \bar{w}\}$; $\quad \mathcal{S} = \{\forall i; \mathbf{w}_i \leq \bar{w}\}$;
$N_{\mathcal{L}} = |\mathcal{L}|$; $\quad N_{\mathcal{S}} = |\mathcal{S}|$;
$P_{\mathcal{L}} = \text{int}\,(W_{\mathcal{L}}/\bar{w})$; $\quad P_{\mathcal{S}} = \text{int}\,(W_{\mathcal{S}}/\bar{w})$;

/* *For tasks* $\in \mathcal{L}$: */
$[\mathbf{p}] = \mathbf{Distribute1}(N_{\mathcal{L}}, P_{\mathcal{L}}, \mathbf{w}_{\mathcal{L}})$

/* *For tasks* $\in \mathcal{S}$: */
$[\mathbf{task\_to\_thread}] = \mathbf{Distribute2}(N_{\mathcal{S}}, P_{\mathcal{S}}, \mathbf{w}_{\mathcal{S}})$

For this algorithm to work, an estimate of the workload, or weights, of the different tasks is needed. In the cases we have applied the technique to, a reasonable assumption is that the work is proportional to the amount of data. With data stored in 2d arrays workload estimate is than easily available.

3. Implementation of nested parallelism in OpenMP

Nested parallelism is possible to implement using message passing parallelization. In MPI [6], creating communicators will make it possible to form groups of threads working together in teams and sending messages to other members within the team for fine-grained parallelism, while the coarse-grained parallelism implies communication between communicators.

The distribution of work to multiple threads in SMP programming is usually done by the compiler. The programmer's job is only to insert directives in the code to assist the compiler. A more explicit approach, where the programmer explicitly distributes tasks to threads is also possible. The explicit approach gives the programmer full control, but does require a much higher level of programmer intervention. Therefor directive based SMP programming is usually the recommended approach. Below we discuss the possibilities and limitations of the two approaches for multi-level parallelism.

3.1. Directives for nested parallelism

Explicit construct for expressing multi-level parallelism is not usually found in directive based, multi-threaded programming for SMP. OpenMP [1] does however have constructs for nested parallelism. In OpenMP a parallel region in Fortran starts by the directive `!$OMP PARALLEL` and ends by `!$OMP END PARALLEL`. The standard allows these to be nested[3].

The more complicated situation where the tasks are split into the sets $\mathcal{L}$ and $\mathcal{S}$, and where nesting only should be executed for set $\mathcal{L}$, should also be possible to implement by nested directives. An example of how this situation can be implemented using nested directives, and a few changes in the code, is given in Example 1.

Example 1:

```
      call compute_weights_of_tasks(w)
      call Distribute_All(N, P, w, p, task_to_thread) /* Distributing threads to tasks, Alg. 1 */

!$OMP  PARALLEL SECTIONS NUM_THREADS(2)

!$OMP  SECTION /* Set L */
!$OMP  PARALLEL DO PRIVATE(i) NUM_THREADS(N)
       do i = 1, N_L
!$OMP  PARALLEL DO PRIVATE(j) SHARED(i) NUM_THREADS(p(i))
          do j = 1, w(i)
             < WORK(j,i) >
          end do
       end do

!$OMP  SECTION /* Set S */
!$OMP  PARALLEL PRIVATE(i,j) NUM_THREADS(P_S)
       thread = OMP_GET_THREAD_NUM()
       do i = N_L+1, N
          if (thread /= task_to_thread(i)) cycle
          do j = 1, w(i)
             < WORK(j,i) >
          end do
       end do
!$OMP  END PARALLEL

!$OMP  END PARALLEL SECTIONS
```

The `$OMP SECTION` directive is used to separate between the two cases. This splitting will in most cases require some restructuring of the code.

In the first section the work in set $\mathcal{L}$ is carried out, and the directives `!$OMP PARALLEL DO` are nested. All variables used in a parallel region are by default `SHARED`. We declare the `i` index as `PRIVATE` in the outer loop, and as `SHARED` in the inner loop, since it should be private for each team and shared among the threads within the same team.

The `NUM_THREADS` clause, included in OpenMP 2.0, controls the number of threads in a team. If

[3]To enable the nesting one has to set the environment variable `OMP_NESTED` to `TRUE` or call the subroutine `OMP_SET_NESTED`.

NUM_THREADS is not set, it is implementation dependent how many threads will work in each (nested) parallel region. Assuming all tasks in set $\mathcal{L}$ and using Distribute1 to distribute threads to tasks, the natural choice is to use N threads on the outer loop, and p(i) threads on the inner loop.

In the second section, the work in set $\mathcal{S}$ is carried out, but here there is only one level of parallelization. Each thread checks for each task if the task is supposed to be done by itself, and executes the task if so.

When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation dependent. As a result, OpenMP-compliant implementations are allowed to serialize nested parallel regions even when nested parallelism is enabled. As of writing (summer 2003), most vendors appears to have an implementation of OpenMP version 2.0 for Fortran available. However, still most vendors seems to have chosen to serialize nested parallel regions.

3.2. Explicit thread programming

OpenMP also allows a more low level work distribution, where the programmer explicitly assigns work to threads. We illustrate how this technique can be used to obtain 2-level parallelism with the following simple example. Suppose a problem has been divided in set $\mathcal{L}$ and $\mathcal{S}$, and that set $\mathcal{L}$ consists of $N = 4$ independent outer-level tasks, where the work of the tasks is given by the weight vector $\mathbf{w} = \{10, 8, 2, 7\}$. Feeding this into Distribute1 with $P = 8$ will give the distribution $\mathbf{p} = \{3, 2, 1, 2\}$ of threads to the 4 outer-level tasks.

We want the threads within a team to divide the work equally among each other, to obtain a good load balance. If one weight unit equals the work associated with one inner-loop iteration, the threads will divide the iterations as shown in Table 1.

Table 1
`thread` *will work on* `task` *and do the iterations from* `jbegin` *to* `jend`.

`thread`	0	1	2	3	4	5	6	7
`task`	1	1	1	2	2	3	4	4
`jbegin`	1	5	8	1	5	1	1	5
`jend`	4	7	10	4	8	2	4	7

The information needed in the computation is given in Table 1 and is easily computed with the threads-to-task distribution available. We apply a simple service routine for this purpose (`find_jindexes`). Implementing the $\mathcal{L}$-section in Example 1 by explicit thread programming, can now be done as:

Example 2:

```
      call find_jindexes /* Construct Table 1 */
!$OMP  PARALLEL PRIVATE(thread,i,j) NUM_THREADS(N)
      thread = OMP_GET_THREAD_NUM()
      i = task(thread)
      do j = jbegin(thread), jend(thread)
         < WORK(j,i) >
      end do
!$OMP  END PARALLEL
```

Implementing the complete Example 1 by explicit thread programming, the `$OMP PARALLEL SECTION`-construct disappears. Instead, it is explicitly decided which threads should do the work in set $\mathcal{L}$ and which should do the work in set $\mathcal{S}$.

The kind of programmer interventions needed for these explicit thread programming changes are to some degree similar to the work needed when parallelizing using MPI [6]. However, no explicit communication is needed as a (virtual-)shared memory is assumed.

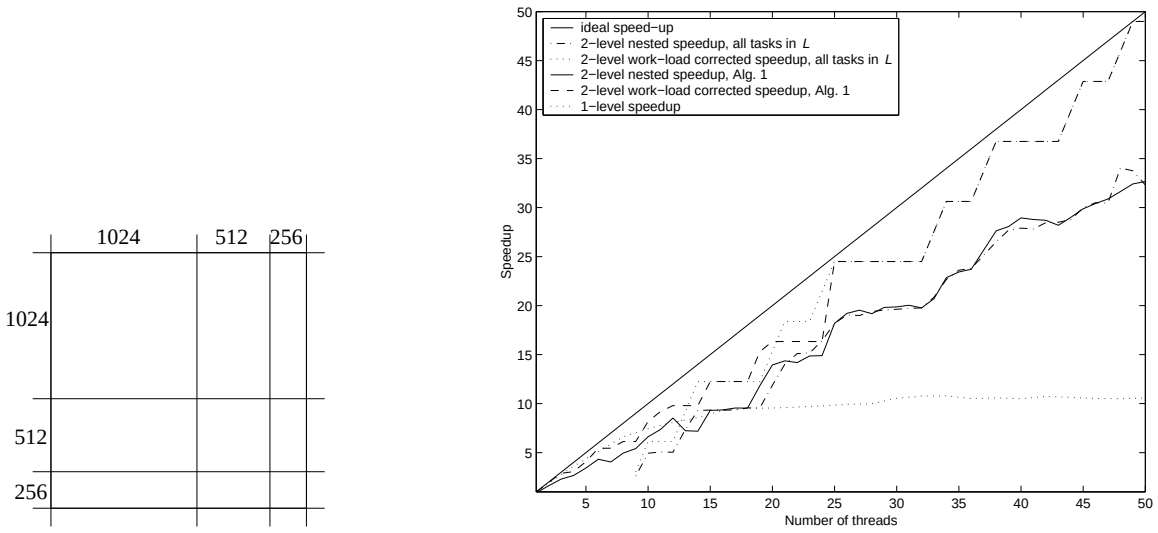

Figure 1. *a)* 1792×1792 *grid divided into 9 pieces. b) Ideal speedup, 2-level work-load corrected (theoretical) speedup, and obtained 2-level nested speedup for the data compression experiments*

4. Data compression experiments

In this section we report on experiences on 2-level parallelism applied to a real application, a wavelet-based data compression routine. The load balancing is done using Algorithm 1 in Section 2. For implementation we have used the explicit thread programming technique outlined in Section 3.2.

The wavelet-based data compression routine is used in an out-of-core earthquake simulator to minimize memory usage as well as disk-traffic [8]. In our experiments we run the compression routine as a stand alone routine. The routine first transforms the data into wavelet-space using a 2d-wavelet transform, and then it stores only the non-zero wavelet coefficients [7].

The wavelet routine only works for arrays $m \times n$ where m and n are integers power of 2. Thus we first divide the array in N blocks of (different) power of 2 sizes. For each of these blocks a 2d-wavelet transform is carried out. As our test case we have chosen a 2d array of size 1792×1792, which is divided into 9 pieces of unequal sizes, as shown in Figure 1a).

We can not expect perfect load balance due to the integer restriction on the number of threads. If we correct for the work imbalance between the outer-level tasks, and assume no extra parallel overhead and perfect load balancing within an outer-level task, we obtain a sharper bound on the achievable 2-level speedup. The formal definition is as follows:

Definition 4.1 *The work-load corrected 2-level speedup is defined as* $\hat{S}_p = T_1/\hat{T}_p$ *where* $T_1 = \sum_i w_i$ *and* $\hat{T}_p = \max\left(\max_{i \in N_{\mathcal{L}}}(w_i/p_i), \max_{i \in P_S} threadwork_i\right)$[4].

In Figure 1b) we display the linear speedup and 2-level work-load corrected speedup, together with the speedup achieved for 2-level and 1-level parallelization. The runs are done on a dedicated Origin 2000 using MIPSpro Fortran Compilers, Version 7.3.1.3m.

We have applied two versions of our distribution algorithm; Algorithm 1 and a simplified version where all tasks are put in set $\mathcal{L}$ regardless of size. When the simplified version is used, it is not possible to run the nested version on less than 9 threads, which is the reason why the curve starts at this point. The 1-level parallelized code reaches its maximum speedup at about 20 threads, while the 2-level parallelized code increases its speedup up to at least 50 threads, where the speedup is 33. We find these results to be very encouraging.

[4] $threadwork_i$ is the amount of work distributed to thread i, and can be computed in Distribute2.

4.1. An adaptive mesh refinement application

We have also applied 2-level nested parallelism to an adaptive mesh refinement [3] application. In contrast to the compression routine test case, which have a static number of tasks, the number of tasks are here dynamically changing. The results can be found in [4].

5. Conclusions

The main purpose of this paper has been to examine the possible gain of utilizing nested parallelism when available in the problem. Our findings are very encouraging. Using the two levels of parallelism turned out to be imperative for good parallel performance on our test cases.

As always good load balancing is essential in achieving good scalability. This becomes more difficult when applying multi-level parallelism. We have presented a simple algorithm which distributes threads to tasks, computing near optimal solution to an NP-complete problem. We also show how 2-level parallelism can be implemented in OpenMP, using explicit thread programming.

The work of parallelizing in 2 levels in OpenMP can often be time consuming, mainly because directives appropriate for the team concept are missing. For instance, a team-aware directive like `!$OMP TEAMPRIVATE`, corresponding to `!$OMP THREADPRIVATE`, can save the programmer for extra changes in the code. This directive is not a part of OpenMP 2.0, even if nesting was implemented in the compiler.

The proper implementation of tasks in set $\mathcal{S}$ could have been made simpler and more intuitive with an `ON_THREAD` clause to the `$OMP PARALLEL DO` directive. Such a clause could give the programmer admission to dictate which thread should do which loop iteration. For debugging purpose, we also miss to use the `!$OMP BARRIER` within parallel do-loops.

As we see larger SMP-systems becomes more and more common, the scalability of OpenMP becomes more important. Utilizing multi-level parallelism will become an important issue in this context. The extension for OpenMP 2.0 points in the right direction. We are, however, very unhappy with the fact that serializing nested parallelism is still compliant with the OpenMP spec.

REFERENCES

[1] OpenMP. `http://www.openmp.org/`.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer, 1999.

[3] M. J. Berger and R. J. LeVeque. AMRCLAW, Adaptive Mesh Refinement + CLAWPACK. `http://www.amath.washington.edu/~rjl/amrclaw`, 1997.

[4] R. Blikberg. Nested parallelism applied to AMRCLAW. In *Proceedings of EWOMP'02, Fourth European Workshop on OpenMP*, 2002.

[5] R. Blikberg and T. Sørevik. Nested parallelism: Allocation of threads to tasks and OpenMP implementation. *Journal of Scientific Programming*, 9(2,3):185–194, 2001.

[6] W. Gropp and R. Lusk. *Using MPI, portable parallel programming with the Message Passing Interface.* The MIT Press, 1994.

[7] M. Lucká and T. Sørevik. Parallel wavelet based compression of two-dimensional data. In *Proceedings of Algoritmy 2000*, 2000.

[8] P. Moczo, M. Lucká, J. Kristek, and M. Kristeková. 3D displacement finite differences and a combined memory optimization. *Bull. Seism. Soc. Am.*, 89:69–79, 1999.