Available online at www.sciencedirect.com
SCIENCE DIRECT®

Parallel Computing 31 (2005) 984–998

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Load balancing and OpenMP implementation of nested parallelism

R. Blikberg [a], T. Sørevik [b,*]

[a] Parallab, BCCS, University of Bergen, Norway
[b] Department of Mathematics, University of Bergen, Johs. Brunsgt. 12, N-5008 Bergen, Norway

Received 9 September 2004; accepted 5 March 2005
Available online 17 October 2005

Abstract

Many problems have multiple layers of parallelism. The outer-level may consist of few and coarse-grained tasks. Next, each of these tasks may also be rich in parallelism, and be split into a number of fine-grained tasks, which again may consist of even finer subtasks, and so on. Here we argue and demonstrate by examples that utilizing multiple layers of parallelism may give much better scaling than if one restricts oneself to only one level of parallelism.

Two non-trivial issues for multi-level parallelism are *load balancing* and *implementation*. In this paper we provide an algorithm for finding good distributions of threads to tasks and discuss how to implement nested parallelism in OpenMP.
© 2005 Elsevier B.V. All rights reserved.

Keywords: OpenMP; SMP-programming; Nested parallelism; Work distribution

1. Introduction

Computational demanding problems, where parallel computing is needed to find a solution within reasonable time, also tend to be complex problems. When breaking a complex problem into smaller independent subproblems for parallel processing,

[*] Corresponding author.
E-mail addresses: ragnhild@ii.uib.no (R. Blikberg), tors@ii.uib.no (T. Sørevik).
URLs: http://www.ii.uib.no/~ragnhild (R. Blikberg), http://www.ii.uib.no/~tors (T. Sørevik).

0167-8191/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2005.03.018

one typically finds several layers of parallelism. The first level or outer-level may consist of few, but large tasks. Each of these outer-level tasks may be split into a number of fine-grained tasks, which again may consist of even finer subtasks, and so on.

If the outer-level tasks are independent, they provide an obvious, coarse-grained parallelism. Unfortunately, coarse-grained parallelism may give poor scalability, as the tasks often are few and of unequal sizes. Consequently, many applications are parallelized at inner-level, exploiting parallelism within each task. Fine-grained parallelism quite often suffers from large parallel overhead and synchronization cost when the number of processes increases, resulting in limited scalability. When scalability is limited for coarse- as well as fine-grained parallelism, then combining the two still can provide reasonable scalability as demonstrated in our examples in Section 4.

In this paper we investigate the advantages and challenges of implementing multi-level parallelism. The great advantage we claim is enhanced scalability. However, we see two main challenges. As is illustrated by our examples, the cases where an outer-level parallelism is not sufficient, is in particular where the outer-level tasks are of unequal sizes. This makes *load balancing* non-trivial, but not less important. Thus, finding a good distribution of threads to outer-level tasks is very important for good efficiency. This question is addressed in Section 2, where we provide an algorithm which distributes threads to tasks.

The second challenge is that of the increased complexity of the *implementation*. We investigate the increased complexity in implementation in the context of OpenMP [1], the standard for SMP-programming. The popularity of SMP-programming is mainly due to its ease of programming as the programmer usually only inserts directives in front of the most time consuming loops. In the case of multi-level parallelism one would like it to be sufficient to simply apply nested sets of directives to achieve the necessary effect, and contrary to most of its predecessors OpenMP allows nested directives.

However, a complicating fact is that the effect of the OpenMP directives for nesting, is not well defined. It is, for instance, compliant with the standard to serialize nested directives. Thus there is no guaranty that multiple levels of parallelism actually are used even if the programmer specifies so by inserting directives. Consequently, the only way to make truly portable OpenMP-code, which faithfully executes the multi-level parallelism intended by the programmer, is the more cumbersome way of explicitly assigning tasks to threads. We therefore demonstrate how this can be done in Section 3.

The effect of 2-level parallelism on two real applications, a wavelet based data compression code and an adaptive mesh refinement code, have been tested and is reported in Section 4. We conclude in Section 5 with some remarks on the limitations of the current OpenMP standard.

2. The distribution algorithm

In this section we will present an algorithm for the distribution of threads to tasks. We will focus our discussion on the 2-level case and try to be exhaustive for this case. We also briefly explain how our algorithms can be extended to multiple levels. The computational units at the outer-level will be called *tasks*. These are assumed to be independent, and of unequal sizes. Each task is supposed to possess an internal fine-grained parallelism. For our algorithms to work, an estimate of the workload, or weight, of a specific task 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.

For load balancing different situations 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 design algorithms for these two extreme cases, and then show how they can be combined to deal with all possible cases.

2.1. Case 1: all tasks need at least one thread

The first extreme case is when the tasks are large and need at least one thread. Obviously this case can only be realized when the number of available threads is larger than the number of outer-level tasks.

The threads should be grouped together in teams, where each team is responsible for doing the work associated with one task. The allocation of threads to tasks can be done in the following way: first assign one thread to each task. Then find the task with the highest 'work-to-thread-ratio' and assign an extra thread to this task. Repeat this process until all threads are assigned to a task. A formal description of this algorithm is given in Algorithm 1.

Algorithm 1. Distribution of threads to tasks

```
/* This algorithm finds a distribution of P threads to the work of N tasks, for
P ⩾ N. The weight of task i is denoted by $\mathbf{w}_i$, while the number of threads distributed
to task i is given by $\mathbf{p}_i$. */

[p] = Distribute1(N, P, w)
$\mathbf{p}_{1:N} = 1$;
for $j = N + 1, P$
    Find $k \in [1:N]$ such that $\frac{\mathbf{w}_k}{\mathbf{p}_k} = \max_{i\in[1,N]}\left(\frac{\mathbf{w}_i}{\mathbf{p}_i}\right)$;
    $\mathbf{p}_k = \mathbf{p}_k + 1$;
end for
```

This very simple algorithm produces not only a good partition, but the optimal one as proved in [9].

2.2. 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 any task will have more than one thread. In some sense this is the dual of the previous problem. Here tasks are assigned to threads.

This subproblem corresponds to a bin-packing problem, and we can distinguish between two versions of it. We can either assume that 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 fix the maximum bin size and try to minimize the number of bins needed to pack all tasks. In either case the problem is known to be NP-complete.

Good approximate solutions can be found by the *best fit decreasing* method [2]. This algorithm is given in Algorithm 2a for the fixed-number-of-bins problem.

Algorithm 2a. Distribution of tasks to threads. Fixed number of bins

```
/* This algorithm finds a distribution of N tasks to P threads, by best fit decreasing
method for bin packing in fixed number of bins. The total amount of work distributed
to thread i is given by pw_i. and the thread which task j is distributed to by thread_
alloc_j. */

[thread_alloc] = Distribute2a($N, P, \mathbf{w}$)

Sort the tasks such that $\mathbf{w}_i \geqslant \mathbf{w}_j \; \forall i < j$ and $i,j \in [1, N]$;
for $j = 1, P$
   $\mathbf{pw}_j = \mathbf{w}_j$;          $\mathbf{thread\_alloc}_j = j$;
end for
for $j = P + 1, N$
   Find $k \in [1 : P]$ such that $\mathbf{pw}_k = \min_{i \in [1,P]} (\mathbf{pw}_i)$;
   $\mathbf{pw}_k = \mathbf{pw}_k + \mathbf{w}_j$;     $\mathbf{thread\_alloc}_j = k$;
end for
```

For the maximum-bin-size problem, the best fit decreasing method attempts to put the task to allocate in the fullest bin it fits. This algorithm is given in Algorithm 2b. Notice that P is not an input parameter anymore, but an output parameter, while $\bar{w}$, which represents the maximum bin size now is needed as an input parameter.

Algorithm 2b. Distribution of tasks to threads. Fixed max size of bins

```
/* This algorithm finds a distribution of N tasks and P, the minimum of threads
needed under the constrain that no thread should have a workload of more than $\bar{w}$.
The method used is best fit decreasing for bin packing in fixed max-sized bins.
The total amount of work distributed to thread i is given by pw_i, and the thread which
task j is distributed to by thread_alloc_j. */

[$P$, thread_alloc] = Distribute2b($N, \bar{w}, \mathbf{w}$)

Sort the tasks such that $\mathbf{w}_i \geqslant \mathbf{w}_j \; \forall i < j$ and $i,j \in [1, N]$;
$P = 1$; $\mathbf{pw}_P = 0$
```

```
for $j = 1, N$
    Find a $k \in [1, P]$such that $k \in \mathrm{argmax}_{i\in[1,P]}\{\mathbf{pw}_i + \mathbf{w}_j : \mathbf{pw}_i + \mathbf{w}_j \leqslant \bar{w}\}$
    if $k$ not exists
        $P = P + 1$;
        $\mathbf{pw}_P = \mathbf{w}_j$;                $\mathbf{thread\_alloc}_j = P$;
    else
        $\mathbf{pw}_k = \mathbf{pw}_k + \mathbf{w}_j$;        $\mathbf{thread\_alloc}_j = k$;
    end if
end for
```

2.3. The combined algorithm

The combined algorithm works by simply sorting the tasks in two sets; one set 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 one thread.

Algorithm 3. Combined Algorithm

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

```
$[\mathscr{L}, \mathscr{S}, \mathbf{p}, \mathbf{thread\_alloc}] = \mathbf{Distribute\_All}(N, P, \mathbf{w})$

    $W = \sum_{i=1}^{N}\mathbf{w}_i$;        $\bar{w} = W/P$;
    $\mathscr{L} = \{\forall i; \mathbf{w}_i > \bar{w}\}$;   $\mathscr{S} = \{\forall i; \mathbf{w}_i \leqslant \bar{w}\}$;
    $N_{\mathscr{L}} = |\mathscr{L}|$;             $N_{\mathscr{S}} = |\mathscr{S}|$;
    $\mathbf{w}_{\mathscr{L}} = \mathbf{w}_i \in \mathscr{L}$;         $\mathbf{w}_{\mathscr{S}} = \mathbf{w}_{i\in\mathscr{S}}$;
    $W_{\mathscr{L}} = \sum_{i\in\mathscr{L}}\mathbf{w}_i$;       $W_{\mathscr{S}} = \sum_{i\in\mathscr{S}}\mathbf{w}_i$;
    /* For tasks $i \in \mathscr{S}$ apply Algorithm 2: */
    if Algorithm2a
        $P_{\mathscr{S}} = \mathrm{int}(W_{\mathscr{S}}z/\bar{w})$;
        $[\mathbf{thread\_alloc}] = \mathbf{Distribute2a}(N_{\mathscr{S}}, P_{\mathscr{S}}, \mathbf{w}_{\mathscr{S}})$
    else if Algorithm2b
        $[P_{\mathscr{S}}, \mathbf{thread\_alloc}] = \mathbf{Distribute2b}(N_{\mathscr{S}}, \bar{w}, \mathbf{w}_{\mathscr{S}})$
    end if

    $P_{\mathscr{L}} = P - P_{\mathscr{S}}$;
    /* For tasks $i \in \mathscr{L}$ apply Algorithm 1: */
    $[\mathbf{p}] = \mathbf{Distribute1}(N_{\mathscr{L}}, P_{\mathscr{L}}, \mathbf{w}_{\mathscr{L}})$
```

Note that if $\mathscr{L}$ is empty after the initial splitting into two sets, the work distribution algorithm degenerates to a 1-level, outer-level parallel strategy. This is fine, as this is precisely the situation where outer-level parallelism is expected to be good:

many tasks, few threads. In this case only Algorithm 2a can be used as Algorithm 2b most likely will produce a value of P_S larger than the number of available threads. In general, if Algorithm 2b is used, the combined algorithm fails to find a feasible answer whenever a P_S is returned which does not satisfy $N_L \leqslant P - P_S$.

This is of course a very convincing argument for always choosing Algorithm 2a. However, also Algorithm 2a has its shortcomings. There exist problems where the combined algorithm produces poorer results than if one puts all tasks in $\mathscr{L}$ and use only Algorithm 1. An example of this is our first numerical test case, in Section 4.1, where the combined Algorithm 3 ($= 1 + 2\text{a}$) produces poorer results than Algorithm 1 for $P \in \{14, 21, 22, 23, 24\}$. This is never the case when 2b is used in the combined algorithm. In that case it can be proved that the combined algorithm will always give a work distribution which is at least as well balanced as the one we get when Algorithm 1 is used alone.

If we in Algorithm 1 store the nodes in a heap order by $\mathbf{w}_i/\mathbf{p}_i$, finding k is done in only one operation. However, the heap needs to be updated in each step, requiring $\mathrm{O}(\log P)$ operations, making the complexity $\mathrm{O}((N - P)\log N)$. The complexity of the sorting step in Algorithm 2 is $\mathrm{O}(N\log N)$ provided an optimal sorting routine is used. If, as for Algorithm 1, the $\mathbf{pw}_k$ is kept in a heap for efficient "best-fit", the complexity of the assignment part of Algorithm 2 becomes $\mathrm{O}(N\log N)$, and for the overall algorithm $\mathrm{O}(N_{\mathscr{S}} \log N_{\mathscr{S}} + N_{\mathscr{L}} \log N_{\mathscr{L}}) = \mathrm{O}(N \log N)$.

For tasks in the set $\mathscr{S}$ the above algorithm gives a complete description of the work distribution, while for a team of threads assigned to a task in $\mathscr{L}$, we also need to divide the work as equal as possible within each team. For problems where the subtasks are many, small and of equal size, this is straightforward, and the subtasks can just be divided in $\mathbf{p}_i$ pieces. The parallelism of a subtask in $\mathscr{L}$ may itself be constituted of independent tasks of unequal size, but this is just the problem already described and should be solved using the above algorithm. Repeating this process recursively until all tasks are in a set $\mathscr{S}$ produces a multi-level work distribution. The number of parallel levels is one more than the number of recursions.

3. Implementation of nested parallelism in OpenMP

Nested parallelism is possible to implement using message passing parallelization. In MPI [11], creating communicators will make it possible to form teams of threads,[1] where the members of the same team can communicate among themselves (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 with its job. A more explicit approach, where the programmer explicitly allocates tasks to threads, is also possible. The explicit approach gives the programmer full control, but does require a much higher level of programmer

[1] In the message passing jargon this is usually called "*groups of processes*".

intervention. Therefore, 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 constructs for expressing multi-level parallelism is not always 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 90 starts by the directive `!$OMP PARALLEL` and ends by `!$OMP END PARALLEL`. The standard allows these to be nested.[2] A simple example is displayed in Example 1, where we have two nested do-loops, both parallelizable. This is made explicit to the compiler by the `!$OMP DO` directive.[3] This example corresponds to having all tasks in the set $\mathscr{L}$; That is each task needs at least one thread to its own disposal. The `NUM_THREADS` clause, is needed to control the number of threads in a team.

Example 1

```
      call distributel(N, P, w, p)/* Distributing threads to
      tasks, Alg. 1*/
!$OMP PARALLEL DO PRIVATE(i) NUM_THREADS(N)
      do i = 1, N
!$OMP PARALLEL DO PRIVATE(j) SHARED(i) NUM_THREADS(p(i))
        do j = 1, w(i)
          <WORK(j, i) >
        end do
      end do
```

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, as it should be private to each team and shared among the threads within the same team.

If `NUM_THREADS` is not set, it is implementation dependent how many threads will work in each (nested) parallel region. Assuming that all tasks are in set $\mathscr{L}$, and Algorithm 1 is used to distribute threads to tasks. Then the natural choice is to establish `N` threads on the outer loop, and fork `p(i)` threads for each of the inner loops.

We are aware that perfectly nested loops,[4] like the simple case of Example 1, can be re-written as one single loop by writing `dok = 1, Nw(N)`, where $Nw(i) = \sum_{j=1}^{i} w(j)$; $i = 1, \ldots, N$. The original indexes i,j can be retrieved as: find i such that

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

[3] The `!$OMP PARALLEL DO` directive provides a shortcut form for specifying a parallel region that contains a single `!$OMP DO` directive.

[4] Nested loops are perfectly nested if assignment statements only occur in the innermost loop of the loop nest.

$Nw(i-1) < k \leqslant Nw(i)$, $j = k - Nw(i-1)$. For perfectly nested loops this rewriting removes the nesting, and straightforward 1-level parallelism can be applied. However, in addition to requiring restructuring of the code, this strategy does not apply to all possible multi-level nesting when the nesting is non-perfect. To keep the example simple we demonstrate the nesting techniques by the simple code-fragment of Example 1. Of course, where nesting is really needed is in more involved cases.

The more complicated situation, where the tasks split into the two sets $\mathscr{L}$ and $\mathscr{S}$ should also be possible to implement by directives. An example of how this situation can be implemented using nested directives, and a few changes in the code, is given in Example 2. Here we assume that we have a routine, based on Algorithm 3, which separates the tasks in the sets $\mathscr{L}$ and $\mathscr{S}$ and find the distribution of threads to tasks (for $\mathscr{L}$) and tasks to threads (for $\mathscr{S}$). This splitting is maintained in the code and will in most cases require some restructuring of the code. The `!$OMP SECTION` directive is used to separate between the two cases. Note that this gives us yet another layer of parallelism.

Example 2

```
      call compute_weights_of_tasks(w)
      call distribute_all(N, P, w, p, thread_alloc)
!$OMP PARALLEL SECTIONS NUM_THREADS(2)
!$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 */

!$OMP PARALLEL PRIVATE(thread,i,j) NUM_THREADS(P_S)
      thread = OMP_GET_THREAD_NUM()
      do i = N_L+1, N
        if (thread /= thread_alloc(i)) cycle
        do j = 1, w(i)
          < WORK(j,i) >
        end do
      end do
!$OMP END PARALLEL
!$OMP END PARALLEL SECTIONS
```

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. Unfortunately many of the vendors have chosen to do so.

3.2. Explicit thread programming

OpenMP also allows a more low level work distribution, where the programmer explicitly assigns work to threads. Here we illustrate how this technique can be used to obtain 2-level parallelism with the following simple example.

Suppose a problem consists of $N = 4$ independent outer-level tasks, where the work of each task is of different size. Let the sizes be given by the weight vector $\mathbf{w} = \{10, 8, 2, 7\}$. Feeding this into Algorithm 1 with $P = 8$ will give the distribution $\mathbf{p} = \{3, 2, 1, 2\}$ for the number of threads allocated to each of the four outer-level tasks.

We want the threads within a team to divide the work of their task equally among each other. If one weight unit equals the work associated with one inner loop iteration, the threads will divide the iterations as shown in Table 1. `thread` will then work on `task(thread)`, and do the j-iterations from `jbegin` to `jend`.

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`). The nested parallelism of Example 1 can now be implemented as

Example 3

```
      call distribute1(N, P, w, p) /* Distributing thread to
      tasks, Alg. 1 */
      call find_jindexes(w, p, task, jbegin, jend) /* Con-
      struct 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
```

Since the values of these arrays decide the distribution of work to threads, they dictate the load balancing. In the example above and the test cases presented in the next section, each item of `WORK(j, i)` requires the same amount of work. This means that good load balance is achieved if `jend(thread)—jbegin(thread)` is approximately the same for all `threads`.

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 combined problem of Example 2 can also be implemented by the use of explicit thread programming. The same technique as in Example 3 is used for the first section. The second section is done by explicitly assigning a loop iteration to the dedicated thread. The `!$OMP SECTION`—construction disappears by explicitly deciding which threads are doing the set $\mathscr{L}$ tasks and which are doing the set $\mathscr{S}$ tasks.

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 [11]. The programmer needs to do a lot of error prone index juggling to get a specific thread to work on the appropriate data. However, no explicit communication is needed as a (virtual-)shared memory is assumed.

4. Experiments

In this section we report on experiences on 2-level parallelism in work we have done on real applications. The load balancing is done using the work distribution algorithms presented in Section 2. For implementation we have used the explicit thread programming technique outlined in Section 3.2.

The first example is a wavelet based data compression routine. In this case we have a predefined and fixed number of tasks. Our second example is an adaptive mesh refinement code. Here the number of tasks, as well as the sizes of the tasks are dynamically changing. The dynamic structure makes this a much more challenging problem both to implement as well as to obtain significant parallel speedup.

All runs are done on a dedicated Origin 2000 using MIPSpro Fortran Compilers, Version 7.3.1.3m.

4.1. Data compression experiments

The wavelet-based data compression routine is used in an out-of-core earthquake simulator to minimize memory usage as well as disk-traffic [14]. In our experiments the compression routine was run as a stand alone routine. The compression routine first transforms the data into wavelet-space using a 2d-wavelet transform, and then stores only the non-zero wavelet coefficients [13].

The wavelet-transform routine only works for arrays $m \times n$ where m and n are integers power of 2. Thus, the array was first divided in N blocks of (different) power of 2 sizes. For each of these blocks a 2d-wavelet transform is carried out. A 2d-wavelet transform is done by applying multiple, independent 1d-wavelet transforms to each row in the block matrix, and next to the columns. As our test case we have chosen a 2d array of size 1792×1792. For the wavelet transforms this is divided into nine pieces of unequal sizes, as shown in Fig. 1. We are using a fast wavelet transform which is known to have linear complexity. Thus the workload of each piece is proportional to its size. If we take the 256×256 piece as our "*unit-of-work*", the size of the nine tasks becomes $\mathbf{w} = \{16,8,8,4,4,4,2,2,1\}$. Giving this problem to Algorithm 3 with $P = 8$, using Algorithm 2a, gives: $\bar{w} = 6.125, \mathscr{L} = \{16, 8, 8\}, \mathscr{S} = \{4, 4, 4, 2, 2, 1\}, P_{\mathscr{S}} = 3, P_{\mathscr{L}} = 5$.

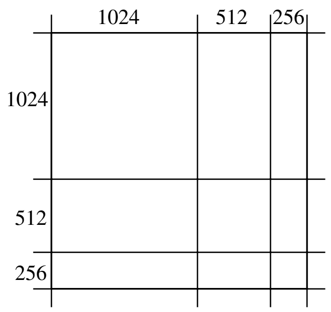

Fig. 1. 1792×1792 grid divided into nine pieces.

In this example 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, while still assuming 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

$$\hat{S}_P = T_1/\hat{T}_P \tag{1}$$

where $T_1 = \sum_{i=1}^{N} \mathbf{w}_i$ and

$$\hat{T}_P = \max\left(\max_{i \in N_{\mathscr{S}}} \frac{\mathbf{w}_i}{\mathbf{p}_i}, \max_{i \in P_{\mathscr{S}}} \mathbf{p}\mathbf{w}_i\right) \tag{2}$$

For the 1792×1792 compression case, with $P = 8$, we will have $\hat{T}_8 = \max(\frac{8}{1}, 6) = 8$, implying $\hat{S}_8 = \frac{49}{8} = 6.125$ as compared to a perfect linear speedup of 8.

In Fig. 2 we display the linear speedup and 2-level work-load corrected speedup with dashed lines, together with the speedup achieved for 2-level and 1-level fine-grained parallelization (solid lines).

We have applied two versions of our distribution algorithm; Algorithm 3 with the choice of 2a, and a simplified version where Algorithm 1 is applied to all tasks regardless of whether they belong to the set $\mathscr{L}$ or $\mathscr{S}$. When Algorithm 1 is used, it is not possible to run the nested version on less than nine threads, which is why the graph 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. Up until 25 threads almost all sub-optimal speedups can be explained by the work imbalance. We find these results to be very encouraging.

It should be mentioned that profiling shows that the time spent in the distribution routine is neglectable.

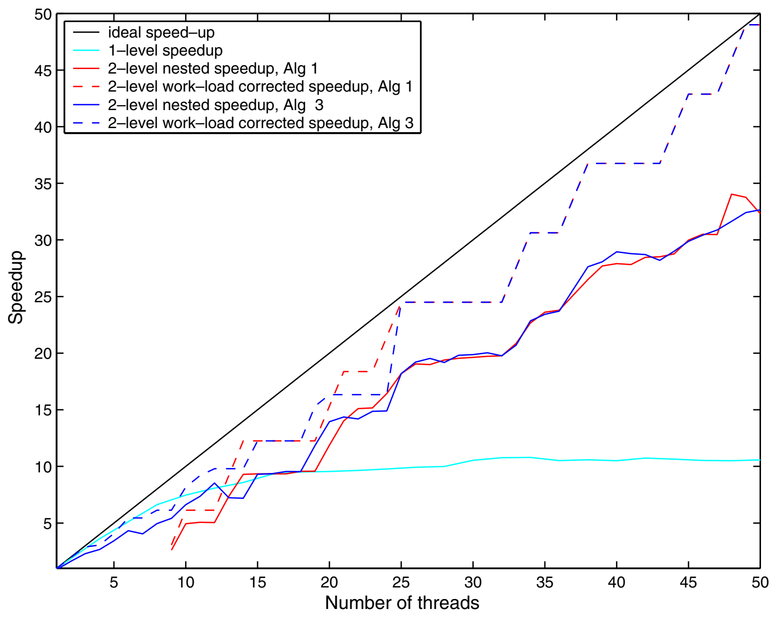

Fig. 2. Linear speedup, 2-level nested work-load corrected speedup, speedup for 1-level inner-loop parallelization and 2-level parallelization for a 1792×1792 data compression problem.

4.2. Adaptive mesh refinement for the shallow water equations

For complex systems modeled by partial differential equations, the difficulty is not usually evenly distributed across the computational domain. Thus, the cell size necessary to capture the difficulties with the appropriate accuracy may differ from region to region and sometimes also in time. Classical cases are problems where complicated structure only appears within a limited area exemplified by non-linear CFD computation, producing shocks. Using the resolution needed to adequately resolve these features on a uniform grid could be extremely expensive. The obvious answer is to refine the grid only where necessary.

If it is not known in advance where a high resolution is needed, or if the features requiring high resolution are moving, adaptivity is called for. Adaptive mesh refinement (AMR) [6] identifies cells which need refinement and organizes them into patches (rectangular blocks of cells). The patches are dynamically created and removed as the need for fine resolution changes. At each level the cells are uniform. If one level of refinement is not sufficient for satisfying the accuracy requirement, the process is repeated recursively.

All grids on a particular level are regenerated every Kth time step, where K is a user-defined input parameter.

AMRCLAW [5] is a freely available software package based on M. Berger's AMR algorithms [4,7] and AMR software [3] combined with R. LeVeque's software Conservation LAW PACKage, CLAWPACK [12].

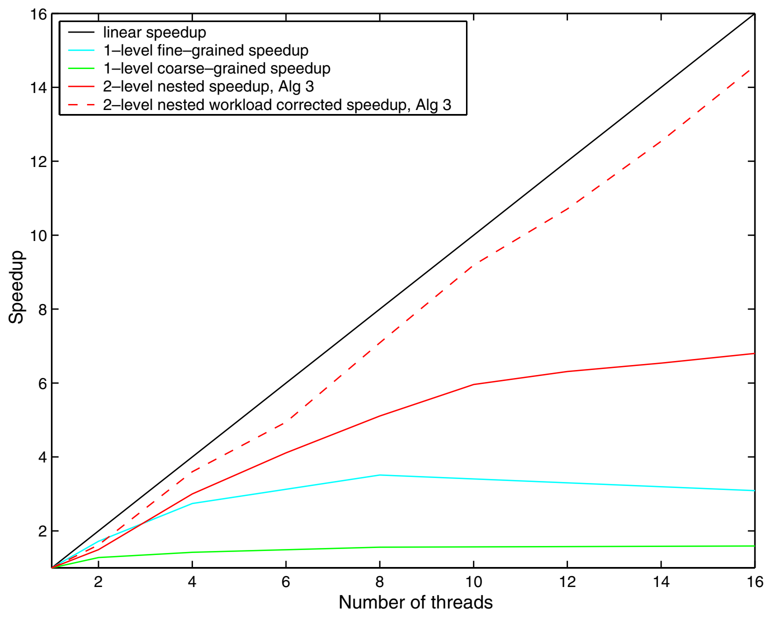

Fig. 3. Speedup obtained for 2-level nested parallelization of AMRCLAW, together with results obtained for 1-level fine-grained and coarse-grained approaches. Linear speedup and 2-level work-load corrected speedup is included.

AMRCLAW has a built-in multi-level structure, where each patch on a given level constitutes an outer-level task, and where all the tasks can be processed in parallel.[5]

For AMRCLAW the obvious strategy is to apply coarse-grained parallelism at patch level, and fine-grained to all grid point updates within a patch. Both time integration and regridding may be parallelized in two levels.

Work-estimates have been done based on a previous study of the costs and savings of AMR in [10]. These have been used to estimate workloads used as input to Algorithm 3.

Fig. 3 shows the speedup obtained for nested parallelization of AMRCLAW applied to a dam-break, shallow water equation test case. For a detailed description of the test case, see [8]. The 2-level "work-load corrected" speedup is also displayed in the figure, to provide a more realistic view of maximum theoretical obtainable speedup. Speedup results for the two 1-level parallel approaches (fine-grained and a coarse-grained) are also shown in the figure. One can clearly see that the nested version has much better speedup than the two 1-level versions. The speedup increases steadily until 10 threads, before it starts to level off.

[5] A minor dependency in AMRCLAW had to be removed at the finest level of grids. In [8], we report how this can be done without altering the mathematics, and at the expense of some very minor additional computation.

Profiling shows that the time spent in the distribution routine is neglectable, and can not be hold responsible for slowing down the performance. We would like to stress that this is a hard problem to parallelize efficiently. The dynamic structure of the problem implies non-uniform data access as grid patches are created and/or deleted. Moreover the dynamic assignment of threads to tasks will most likely create a very irregular data-access pattern, and put severe stress to the memory-bottleneck of all cache based HPC-systems; The internal moving of data. Whether the cost of this data movement could be reduced by explicit message passing is an unanswered question. We do however know that the effort in code rewriting would have been excessive.

Keeping all these difficulties in mind, we find the speedup quite satisfactory. Not at least because both the standard 1-level strategies failed so badly. We therefore conclude that for these cases, nested parallelism offers much better scaling opportunities than the 1-level parallel strategy.

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 two levels of parallelism turned out to be imperative for good parallel performance on our test cases. Utilizing nested parallelism on a real life problem, a wavelet compression application, a speedup of 33 was achieved for 50 threads. We find this result very satisfactory.

As always good load balancing is essential in achieving good scalability. This becomes more difficult when applying multi-level parallelism. In Section 2 we give an algorithm which distributes threads to tasks. The algorithm is simple and efficient, computing near optimal solution to an NP-complete problem. We also show how 2-level parallelism can be implemented in OpenMP, using directive based or explicit thread programming.

The work of parallelizing AMRCLAW in two levels of parallelization in OpenMP was time consuming, mainly because directives appropriate for the team concept were missing. For instance, a team-aware directive like `!$OMP TEAMPRIVATE`, corresponding to `!$OMP THREADPRIVATE`, would have saved us for extra changes in the code. This directive is not included in the OpenMP 2.0 standard, and consequently not available even on compilers which implement the full standard with proper nesting.

The proper implementation of tasks in set $\mathscr{S}$ could have been made simpler and more intuitive with an `ON_THREAD` clause to the `!$OMP PARALLEL DO` directive. Such a clause could give the programmer admission to dictate which thread should do which loop iteration. For debugging purpose, it would have been convenient being allowed to use the `!$OMP BARRIER` within parallel do-loops. This is not permitted by the standard.

With the steady increasing size of the SMP-systems being used, 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 specification.

References

[1] OpenMP. Available from: <http://www.openmp.org/>.

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

[3] M.J. Berger. Adaptive mesh refinement for hyperbolic conservation laws. Available from: <http://cs.nyu.edu/cs/faculty/berger/amrsoftware.html>, 1995.

[4] M.J. Berger, P. Colella, Local adaptive mesh refinement for shock hydrodynamics, Journal of Computational Physics 53 (3) (1984) 561–568.

[5] M.J. Berger, R.J. LeVeque. AMRCLAW, Adaptive mesh refinement + CLAWPACK. Available from: <http://www.amath.washington.edu/~rjl/amrclaw>, 1997.

[6] M.J. Berger, R.J. LeVeque, Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems, SIAM Journal on Numerical Analysis 35 (6) (1998) 2298–2316.

[7] M.J. Berger, J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, Journal of Computational Physics 53 (1984) 484–512.

[8] R. Blikberg, Nested parallelism applied to AMRCLAW, in: Proceedings of EWOMP'02, Fourth European Workshop on OpenMP, 2002.

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

[10] R. Blikberg, T. Sørevik, Cost and savings of adaptive mesh refinement, Technical Report, January 2002.

[11] W. Gropp, R. Lusk, Using MPI, Portable Parallel Programming With The Message Passing Interface, The MIT Press, Cambridge, MA, USA, 1994.

[12] R.J. LeVeque, J.O. Langseth, CLAWPACK 3.0, a Software package for conservation laws and hyperbolic systems. Available from: <http://www.amath.washington.edu/~rjl/clawpack>, 1997.

[13] M. Lucká, T. Sørevik, Parallel wavelet based compression of two-dimensional data, in: Proceedings of Algoritmy 2000, 2000.

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