Nested parallelism: Allocation of threads to tasks and OpenMP implementation

Ragnhild Blikberg* Tor Sørevik†

Published in
Journal of Scientific Programming[1], 9(2,3):185-194, 2001.

Abstract

In this paper we discuss the use of nested parallelism. Our claim is that if the problem naturally possesses multiple levels of parallelism, then applying parallelism to all levels may significantly enhance the scalability of your algorithm. This claim is sustained by numerical experiments.

We also discuss how to implement multi-level parallelism using OpenMP. We find current OpenMP implementation, based on version 1.0, to have severe limitation for implementing nested parallelization. We then show how this can be circumvented by explicitly assign task to threads.

Load balancing issues become more complicated with two (or more) levels of parallelism. To handle this problem, we have designed a distribution algorithm which groups threads into teams, each team being responsible for one course grain outer-level task. This algorithm is proven to produce the optimal load balance, under given assumptions.

Keywords: OpenMP, SMP-parallelism, nested parallelism

*Department of Informatics, University of Bergen, Norway, Tel: +47 55 58 41 66, Fax: +47 55 58 41 99, E-mail: ragnhild@ii.uib.no, URL: `http://www.ii.uib.no/~ragnhild`

†Department of Informatics, University of Bergen, Norway, Tel: +47 55 58 41 72, Fax: +47 55 58 41 99, E-mail: tors@ii.uib.no, URL: `http://www.ii.uib.no/~tors`

[1] Due to copyrights, the publisher objects to include their `pdf`-file in this thesis. Therefore this article has a different layout in this thesis. The text, however, is identical.

1 Introduction

Computational demanding problems, where parallel computing is needed for finding a solution within reasonable time, also tend to be complex problems. When breaking complex problems into smaller independent subproblems for parallel processing, one typically finds several layers. The first level or outer-level consists of few, but large tasks. Next each of these outer-level tasks may split into a number of fine grained tasks, which again may consist of even finer subtasks, and so on.

In this paper we investigate the advantage and problems when implementing multi-level parallelism. The great advantage we claim is enhanced scalability. We see two main problems. The first is how to achieve a good load balance. The second problem is that of increased complexity in implementation. We investigate the latter in the context of OpenMP, the new standard for SMP-programming. The popularity of SMP-programming is mainly due to its ease of programming. Its main drawback has been limited scalability. As long as SMP hardware was bus-connected UMA system with a small number of processors, the limited scalability of the programming model was reflected in the hardware. In the last few years we have, however, witnessed some great successes for scalable cc-NUMA systems with distributed shared memory. These systems support OpenMP program across hundreds of processors. There is no doubt that this trend will be picked up by an increasing number of vendors and that future HPC-hardware will as a rule support SMP-programming across hundreds, and soon thousands, of processors. At this background the limited scalability of any SMP-programming model becomes a serious bottleneck.

The poor scalability of SMP-programs is, to a great extent, due to the fact that most SMP-program is limited to fine grained loop-level parallelism. An important goal for the OpenMP standard is to enhance scalability by encourage more course grain parallelism than possible with loop-level parallelism. In this paper we study the possibilities and limitations of OpenMP to 2-level parallelism.

To set the stage we first would like to give a motivating example. This is meant to illustrate some of the possibilities and problems of 2-level parallelism. Suppose that each outer-level task, i, has a parallel part of sequential computational cost, w_i, and a sequential part of cost, s_i. The sequential part consists of the code which is not easily parallelized by inner-level parallelization, but can also consist of parallel overhead. For N tasks, the total sequential runtime for all the tasks is

$$T_1 = \sum_{i=1}^{N} (w_i + s_i) = W + S. \tag{1}$$

Applying inner-level parallelization the runtime using P threads will be

$$T_P = \sum_{i=1}^{N} \left(\frac{w_i}{P} + s_i \right) = \frac{W}{P} + S. \tag{2}$$

For nested parallelism (NP) we need to distribute the threads among the outer-level tasks. Let p_i be the number of threads associated with task i. Now the runtime for P threads will be

$$T_{NP} = \max_{i=1,..,N} \left(\frac{w_i}{p_i} + s_i \right). \tag{3}$$

If the load balance in the nested version is perfect, $w_i/p_i = W/P$ for the N outer-level tasks. Then the runtime for 2-level parallelism will be

$$T_{NP} \geq \frac{W}{P} + \max_{i=1,..,N} s_i. \tag{4}$$

Optimal 2-level parallelism is achieved when we have $s_1 = s_2 = ... = s_N = S/N$. Then

$$T_{NP} \geq \frac{W}{P} + \frac{S}{N}. \tag{5}$$

It follows that when T_{NP} is close to $\frac{W}{P} + \frac{S}{N}$, then $T_P > T_{NP}$. The level of improvement depends on the actual numbers for P and N as well as the ratio S/W. This dependency is illustrated in Figure 1. As usual we define speed-up on P threads as $S_p = T_1/T_p$, where T_p is the runtime on P threads.

This example illustrates that optimal use of nested parallelism can give significant improvements for a large number of threads. But it is important to stress that a necessary requirement for this is an optimal or near optimal distribution of threads to tasks. Thus finding an optimal distribution of threads to outer-level tasks is very important for good efficiency. This question is addressed in section 2, where we give an algorithm which distributes threads to tasks. We prove that this algorithm gives the optimal solution and give its complexity.

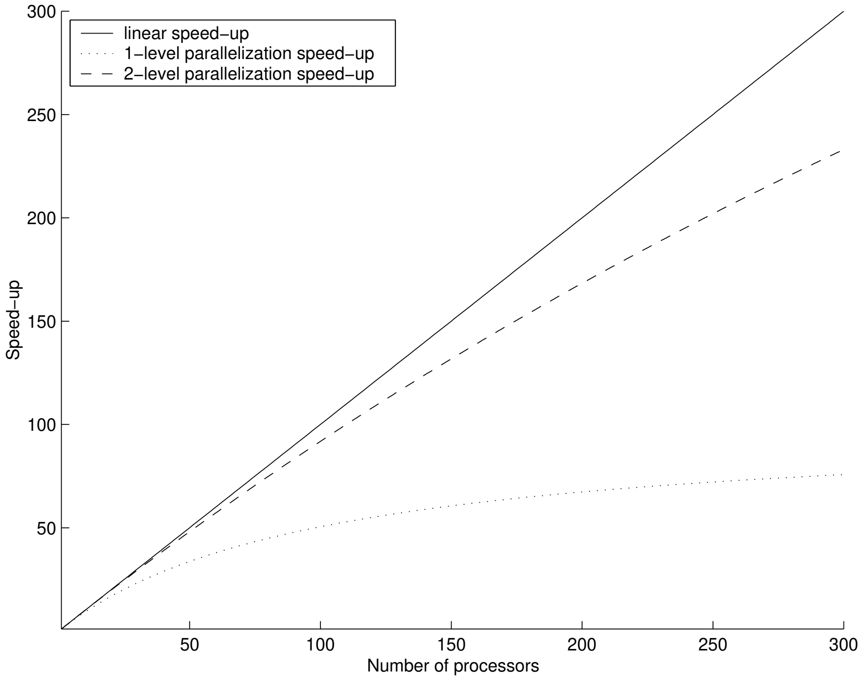

Figure 1: *Speed-up curves for inner-level parallelism as described by (2) and 2-level parallelism obeying the r.h.s. of (5) for the idealized cases above with* $N = 10$ *and* $S = 0.01W$.

In section 3, we discuss how to implement 2-level parallelism in OpenMP. First we look at the possibilities and shortcomings of directive based implementations, and then we show how to implement 2-level parallelism by explicit programming the tasks of the individual threads. In section 4, we test the effect of 2-level parallelism on two test problems, an artificial problem, a matrix multiplication code, and a real life problem, a wavelet based data compression code. In section 5 we will discuss the extensions to OpenMP 1.0 we find necessary to express nesting appropriately. Finally the conclusions will be given.

2 The distribution algorithm

In cases where the number of outer-level tasks are greater than the number of threads, the most coarse grain parallelism is achieved by assigning multiple tasks to each thread and not using any inner-level parallelism. If there is no dependencies between the outer tasks, achieving the optimal load balance reduces to the standard bin-packing problem in this case. We will not consider this case here and therefore assume that 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 highest 'work-to-thread-ratio' and assign an extra thread to this task. Repeat until all threads are assigned to a task.

In the sequel we will give a formal definition of our distribution problem, formalize the above algorithm and prove its optimality. For efficiency we store the tasks in a heap with the task having the highest 'work-to-thread-ratio' as the root node. Then finding the task with highest 'work-to-thread-ratio' is done in $O(1)$, while $O(\log N)$ is needed to update the heap.

Notice that the enumeration of the tasks changes during the distribution in such a way that the first task always has the largest work to threads ratio. The workload, or parallel part of sequential computational cost, of a task is given by its weight, denoted w_i.

Definition 1 {The optimal distribution problem}:

Find an optimal distribution of P threads to N tasks such that $\max_{i=1,...,N}\left(\frac{w_i}{p_i}\right)$ is minimized over all partitions $\{p_1, ..., p_N\}$ given the constraints: $\sum_{i=1}^{N} p_i = P$; $\forall p_i$ positive integers.

Algorithm 1: Distribution of threads to tasks

> **for** $i = 1, N$
> $p_i = 1;$
> **end for**
> **for** $j = N+1, P$
>   update the heap such that $\frac{w_1}{p_1} \geq \max_{i=2,...,N}(\frac{w_i}{p_i})$;
> $p_1 = p_1 + 1;$
> **end for**

This extremely simple algorithm produces not only a good partition, but the optimal one. The reminder of this section is devoted to prove the optimality of Algorithm 1. First we need the following lemma.

Lemma 1: After the update of the heap in any iteration, j, of Algorithm 1, $\frac{w_i}{p_i-1} \geq \frac{w_1}{p_1} \ \forall i$.

For simplicity we assume $\frac{w_i}{0}$ to be a well defined number larger than $\max_{i=1,\cdots,N} w_i$. The proof holds without this dubious assumption, but without it we need to treat this as a special case making the proof rather messy.

Proof: The lemma is proved by induction on j:
A) The lemma is true after the initialization $j = N$ since by definition $\frac{w_i}{0} > \frac{w_1}{1} \ \forall i$.
B) Assume the lemma is true for fixed j. Let $\{w_i, p_i\}$ be the indexing after the heap reordering in step j, and define $rmax(j) = \max_{i=1,\ldots,N}(\frac{w_i}{p_i}) = \frac{w_1}{p_1}$. Then for $j+1$: $rmax(j+1) = \max(\frac{w_1}{p_1+1}, \max_{i=2,\ldots,N}(\frac{w_i}{p_i})) \leq \frac{w_1}{p_1} = rmax(j)$. Then since $\frac{w_i}{p_i-1} \geq rmax(j) \geq rmax(j+1)$ for $i = 2, \ldots, N$ by the induction assumption, and $\frac{w_1}{p_1} = rmax(j) \geq rmax(j+1)$, the hypothesis is true for $j+1$ as well. □

Theorem 1: Algorithm 1 gives the optimal distribution to the distribution problem, with optimality defined as in Definition 1.

Proof: Theorem 1 can be proved by self-contradiction using Lemma 1: Suppose that there is another distribution $q_1, \ldots, q_N$ which gives $\frac{w_1}{q_1} < \frac{w_1}{p_1}$ that implies $q_1 > p_1$. Since $P = \sum_{i=1}^N q_i = \sum_{i=1}^N p_i$ it therefore must exist a k such that $q_k < p_k$ for a $1 < k \leq N$. But then $\frac{w_1}{q_1} \geq \frac{w_k}{q_k} \geq \frac{w_k}{p_k+1} \geq \frac{w_1}{p_1}$, the last inequality coming from Lemma 1. This obviously contradicts the assumption that $q_1, \ldots, q_N$ is a better distribution. Thus the assumption must be wrong and the theorem proved ad absurdum. □

The main loop is repeated $P - N$ times. Inside the loop we extract the top of the heap, update that element and reorganize the heap. The two first operations are done in constant time, the second is at worst $O(\log N)$. This gives an overall complexity of $O((P-N)\log N)$.

3 Implementation of nested parallelism in OpenMP

Nested parallelism is possible to implement using message passing parallelization. In MPI [7], 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 grain parallelism, while the coarse grain 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 intervention. Therefor directive based SMP programming is usually the recommended approach. Below we discuss the possibilities and limitations of the two approaches for multilevel parallelism.

3.1 Directives for nested parallelism

Explicit construct for expressing multilevel parallelism is not usually found in directive based, multi-threaded programming for SMP. OpenMP, the new industry-standard for SMP-programming, does however allow some form of nested parallelism. The OpenMP group released its Fortran standard version 1.0 in October 1997 [1], [2], and the first commercial implementations of this were available in the spring of 1998. It is supported by all major vendors of SMP-systems and has now become the de facto standard for SMP-programming.

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, as shown in Example 1. To enable the nesting one has to set the environment variable `OMP_NESTED` to `TRUE` or call the subroutine `OMP_SET_NESTED`.

Example 1:

```
!$OMP PARALLEL DO PRIVATE(i)
      do i = 1, N
!$OMP PARALLEL DO PRIVATE(j), SHARED(i)
         do j = 1, w(i)
```

```
            < WORK(i,j) >
         end do
!$OMP END PARALLEL DO
      end do
!$OMP END PARALLEL DO
```

If nothing else is specified, all variables used in a parallel region gets `SHARED` by default if they are not put in lists of other clauses. It is not well explained in the 1.0 spec. how the variables should be declared in nested regions. Nevertheless, we find it naturally to declare the `i` index as `PRIVATE` in the outer loop, and as `SHARED` in the inner loop, since it should be private to each team and shared among the threads within the same team.

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 far as we know, none of the vendors supporting OpenMP Fortran version 1.0 have implemented nesting.

SGI's MIPSpro compiler has a restricted form for nested parallelism. This does however not follow the OpenMP 1.0 standard. It applies only to do-loops in Fortran and demands the loops to be perfectly nested (Loops are *perfectly nested* in Fortran if there is no code between the `DO` statements and between the `END DO` statements.). The following example shows how the SGI nesting is supposed to work. In this case, 2 threads will be created in the outer-loop. These will act as masters for two teams of threads working together on the inner-loop.

Example 2:

```
!$OMP PARALLEL DO
!$SGI+NEST(i,j) ONTO(2,*)
      do i = 1, N
         do j = 1, w(i)
            < WORK(i,j) >
         end do
      end do
!$OMP END PARALLEL DO
```

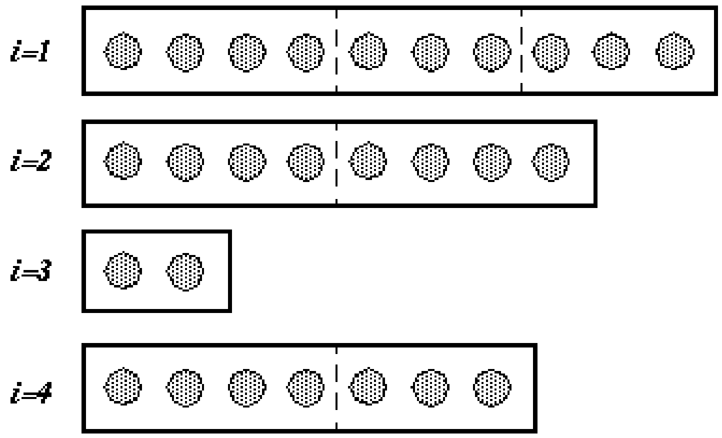

Figure 2: *Problem which consist of four tasks of work, each with a different amount of work.*

3.2 Explicit thread programming

In this approach the user has to manually change the code to distribute tasks to threads. Here we show a simple example of how this can be done, and then parallelize the code in Example 1 in two levels.

Suppose a problem consists of $N = 4$ outer-level tasks, each of them with a different amount of work. The work in each task is given as weights w_i. The problem can be illustrated as in Figure 2, where the balls symbol fine grain tasks of unit weight. Having $P = 8$ available threads and the weights $\mathbf{w}^T = (10, 8, 2, 7)$, using Algorithm 1 will give the distribution $\mathbf{p}^T = (3, 2, 1, 2)$ of threads to outer-level tasks. In the next step the threads within each team have to divide the work within each task among each other. If one weight unit equals the work associated with one inner loop iterations, the threads will divide the iterations as shown in the Table 1. `thread` will then work on task `mytask`, and do the iterations from `jbegin` to `jend`. The data in Table 1 describe exactly the mapping of threads to outer-level tasks and the portion of inner-level task to be carried out of each thread. With this information in hand, nested parallelism in Example 1 can now be implemented as:

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

Table 1: *'thread' will work on 'mytask' and do the iterations from 'jbegin' to 'jend'.*

Example 3:

```
      P = OMP_GET_MAX_THREADS()
!$OMP PARALLEL DO PRIVATE(thread,i,j)
      do thread = 0, P-1
         i = mytask(thread)
         do j = jbegin(thread), jend(thread)
            < WORK(i,j) >
         end do
      end do
!$OMP END PARALLEL DO
```

The values of the arrays `mytask`, `jbegin` and `jend` have to be carefully decided by the user in advance in order to make sure that the exact same computations are done in parallel as in sequential. Since the values of these arrays decide the distribution of work to threads, they dictates the load balancing. In the example above and the test cases presented in the next section, each item of `WORK(i,j)` require 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`. In our test cases, `mytask` is computed using Algorithm 1.

The kind of programmer interventions needed for these changes are to some degree similar to the work needed when parallelization using MPI [7], except for the fact that no explicit communication is needed in OpenMP. In both cases the programmer has to split the work and data and allocate it to specific threads by carefully rewriting the program. The correctness of the program is her full responsibility.

4 Experiments

In this section we report on two experiments on 2-level parallelism. The load balancing is done using the work allocation algorithm presented in section 2. For implementation we have used the explicit thread programming technique outlined in section 3.2. The first experiment is done on a synthetic test example, a matrix multiplication test code. Our second example is a real application, a wavelet based data compression routine.

4.1 Matrix-multiplication

To test our ideas on the importance of utilizing multilevel parallelism, we made an artificial test code, using a simple matrix multiply as the computational kernel.

Suppose we have N tasks, where a task, $i = 1, \cdots, N$ is a multiplication of the matrices $A_{m \times w_i}$ and $B_{w_i \times m}$. Each task has an amount of work proportional to the weight w_i. Again we assume that the number of threads is larger than N, the number of tasks. 1-level parallelization of the matrix-multiplication where we parallelize each of the N matrix-multiplications can be implemented like in Example 4, assuming $N < P$.

Example 4:

```
      C = 0
      do i = 1, N
!$OMP PARALLEL DO PRIVATE(j,k,l)
         do j = 1, w(i)
            do k = 1, m
               do l = 1, m
                  C(l,j,i) = C(l,j,i) + A(l,k,i) * B(k,j,i)
               end do
            end do
         end do
!$OMP END PARALLEL DO
      end do
```

Parallelizing Example 4 by explicit thread programming, as explained in section 3.2, we get:

Example 5:

```
      C = 0
!$OMP PARALLEL DO PRIVATE(thread,i,j,k,l)
      do thread = 0, P-1
         i = mytask(thread)
         do j = jbegin(thread),jend(thread)
            do k = 1, m
               do l = 1, m
                  C(l,j,i) = C(l,j,i) + A(l,k,i) * B(k,j,i)
               end do
            end do
         end do
      end do
!$OMP END PARALLEL DO
```

These two cases have been tested for $m = 700$ and for $N = 1, 10$. w_i is chosen uniformly random from the interval $700 \leq w_i \leq 7000$.

In Figure 3 we display the linear speed-up, together with the speed-up achieved for 2-level nested parallelization and 1-level parallelization as a function of threads. For this particular case the number of outer-level tasks is $N = 4$. The runs are on a dedicated Origin 2000 using MIPSpro Fortran Compilers, Version 7.3.1.1m. For the other values of N the details are different, but the overall picture is the same as for $N = 4$.

For up to about 20 threads the 1-level parallelization shows super linear speed-up, probably due to cache effects. However, as the number of threads increases, some unavoidable overhead starts to creep in for the 1-level parallelism. The 2-level parallelization speed-up is for up to 50 threads lower than the 1-level speedup. This is naturally since the load balance among the tasks in 2-level parallelism is bad for a low number of threads. However, adding CPU/threads beyond 50, the 2-level parallelism still increases the speed-up, while the 1-level speed-up starts to fall below. The 2-level parallelization speedup increases until 100 threads, where its speedup is about 50.

The reason for sub-linear speed-up in this case is not sequential execution of part of our application program. But most likely the extra cost of forking and joining threads in OpenMP and the increased number of synchronization

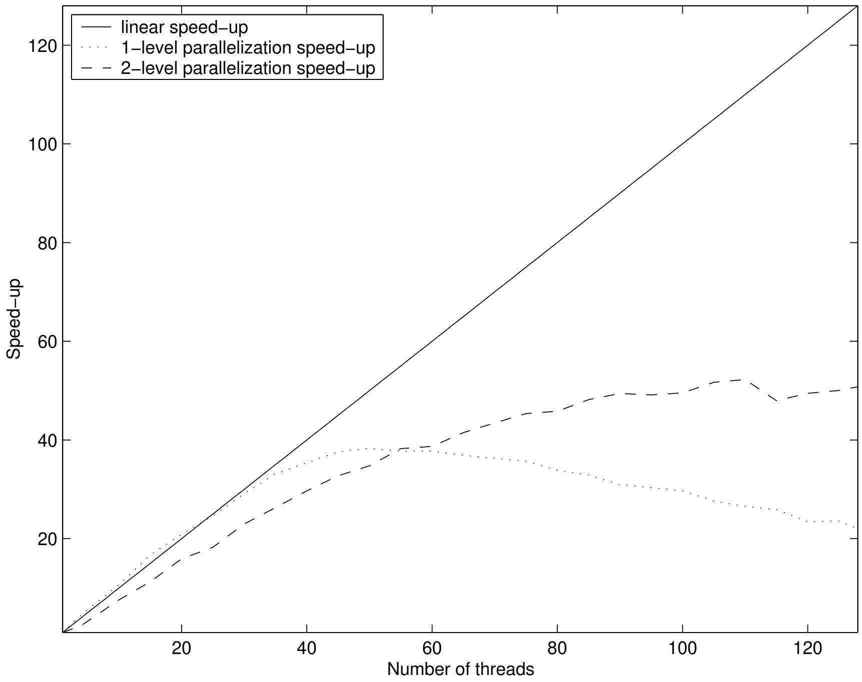

Figure 3: *Linear speed-up, speed-up for 1-level inner-loop parallelization and speed-up for 2-level parallelization for $N = 4$ outer-level tasks of matrix multiplication.*

points. The difference in the two implementations than is that in the inner-level parallelism P threads are forked N times, while they in the 2-level case only is forked once. The cost for this saving is some extra index juggling and a slightly lower bound for theoretical speed-up. The later having a measurable effect on small number of threads. See [10] for documentation on the cost of fork-join and synchronization in OpenMP.

This example modify slightly the assumption in the introduction. It does not have a perfect load balance in the 2-level case. But even with this (realistic!) modification it confirms our fundamental hypothesis: 2-level parallelism scales better and for high thread-numbers it shows better speed-up than 1-level parallelism!

4.2 Data compression

To try our ideas on a more realistic test case, we moved on to a data compression routine which are used in an out-of-core earthquake simulator.

The underlying idea of the compression algorithm is to first transform the data into wavelet-space using a 2d-wavelet transform and then storing only the non-zero wavelet coefficients [9]. To increase the compression rate, two more techniques are used.

Thresholding: What we have is approximate values to inaccurate data. Thus all data less than a certain value should be regarded as noise and could be represented by zero without loss of significance.

Quantization: In essence thresholding says that only the M first binary digits are significant. Thus without any further loss of accuracy the wavelet coefficients can be represented by only M bits, giving us an extra saving factor of M/64.

After the wavelet coefficients have been massaged by thresholding and quantization they are encoded. In the parallel version this is done by dividing the data in P separate streams, making this an embarrassing parallel operation.

The wavelet routine only works for arrays $m \times n$ where m and n are integers power of 2. Thus we first chop up the array 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 transform to each row in the block matrix, and next on the columns.

The overall compression algorithm is displayed in Algorithm 2.

Algorithm 2: Compression

> **input:** `UU, N, M`
> */*The wavelet transform for all* `UU` *blocks*/*
> **for** `i = 1, N`
>   `wavelet2d(block no.  i of UU);`
> **end for**
> `Umax` = $\max_{\forall(i,j)} \mid UU(i,j) \mid$;
> **where**($\mid UU \mid < Umax \times 2^{-M}$) `UU = 0 ;`
>   */* Thresholding */*
> **end where**
> `UUI = UU` */* Quantization */*
> `encode (UUI);`

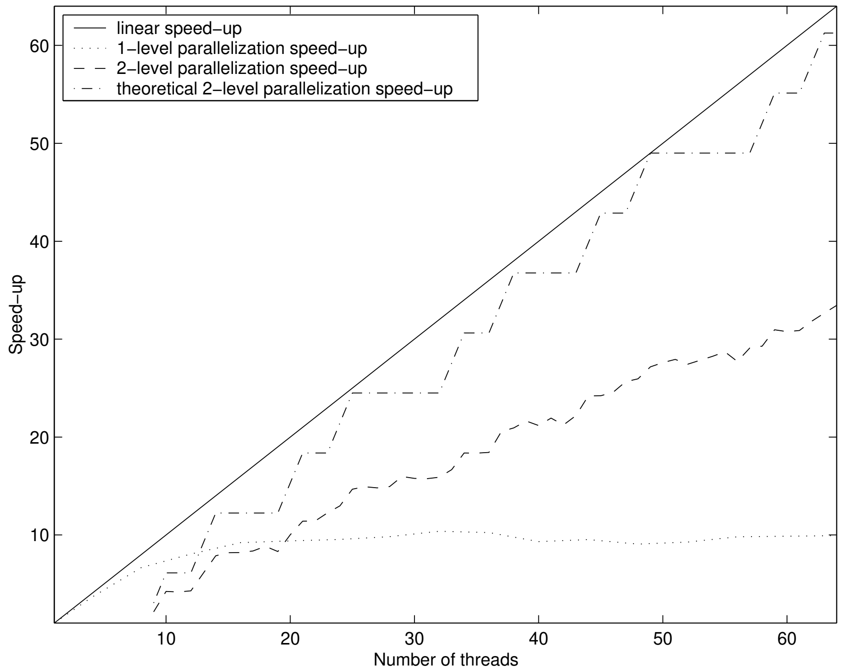

Figure 4: *Linear speed-up, theoretical bound for 2-level speed-up, speed-up for 1-level inner-loop parallelization and 2-level parallelization for a 1792×1792 data compression problem.*

The time consuming part is the wavelet transform (typical 60-70%). This is also the only part having 2-level parallelism.

As our test case we have chosen a 2d array of size 1792×1792. For the wavelet transforms this is divided into 9 pieces of unequal sizes. We are using a fast wavelet transform which is known to have linear complexity. Thus the work is proportional to the size of the corresponding array.

In this example we can not expect perfect load balance due to the integer restriction on the number of threads. If we assume no extra parallel overhead and perfect load balancing within an outer-level task, but not necessarily between outer level tasks, we obtain a sharper bound on the 2-level speed-up. We may define $\hat{T}_p = \max_i T_i/p_i$ as the theoretical bound on the run-time of 2-level parallelism. The theoretical bound on the 2-level speed-up is then given as $\hat{S}_p = T_1/\hat{T}_p$. (In the example displayed in Table 1 and Figure 2, $\hat{T}_8 = 4$ and $\hat{S}_8 = \frac{27}{4} = 6.75$.)

In Figure 4 we display the linear speed-up and theoretical upper bound on the 2-level speed-up, together with the speed-up achieved for 2-level and 1-level parallelization. The runs are done on a dedicated Origin 2000 using MIPSpro Fortran Compilers, Version 7.3.1.1m. 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 speed-up at about 20 threads, while the 2-level parallelized code increases its speed-up up to at least 64 threads, where the speed-up is 33. In particular notice that for less than about 40 threads, the speed-up curve of the 2-level parallelized code mimics the shape of the theoretical upper bound.

We find that these results to be very encouraging, in particular the fact that the 2-level parallelism shows the same improvements in scaling as indicated for the theoretical case displayed in Figure 1.

5 Shortcomings of the OpenMP directives

In section 3 we showed how to implement multilevel parallelism in OpenMP using explicit thread programming. But as we argued in section 3, for simplicity we would prefer programming this with directives and/or function calls only. In this section we will discuss briefly some of the current shortcomings of OpenMP 1.0 and the needed extensions to enable directive based multilevel parallelism.

The work described in this paper was done in the spring of year 2000. At that time only version 1.0 of the OpenMP Fortran compiler was available to us. Nov 3, 2000 the OpenMP ARB released version 2.0 [3] of the Fortran specification. As of writing, still no working version of 2.0 is available. Thus our experiences is with version 1.0, but we'll try to comment on whether or not version 2.0 will cure the problems we describe.

Setting the number of threads by call to the `OMP_SET_NUM_THREADS` routine in OpenMP is only legal outside a parallel region. If nesting is implemented, and nested directives are applied in 2 levels, the total number of threads created in the inner-level becomes the same as the number set outside the outer-level, say P. But as a `PARALLEL DO` directive at the outer-level will apply to all the threads set, no additional threads is available at the inner-level, and nested parallelism is not obtained.

In the recent released OpenMP 2.0 there is a new clause, `NUM_THREADS`(*scalar_integer_expression*), to the parallel regions directives.

This clause requests that a specific number of threads are used in the region. This also works for nested regions, and as far as we can see, it solves the problem described above. The code in Example 1, 2 and 3 can now be written as:

Example 6:

```
!$OMP PARALLEL DO PRIVATE(i), NUM_THREADS(N)
      do i = 1, N
!$OMP PARALLEL DO PRIVATE(j), SHARED(i), NUM_THREADS(p(i))
         do j = 1, w(i)
            < WORK(i,j) >
         end do
!$OMP END PARALLEL DO
      end do
!$OMP END PARALLEL DO
```

When parallelizing the data compression code in 2 levels, it was necessary to have two inner-level loops with a synchronization barrier in between, like this:

Example 7:

```
      P = OMP_GET_MAX_THREADS()
!$OMP PARALLEL DO PRIVATE(thread,i,j)
      do thread = 0, P-1
         i = mytask(thread)
         do j = jbegin(thread), jend(thread)
            < WORK1(i,j) >
         end do
         < ...  >
         < BARRIER >
         < ...  >
         do j = jbegin(thread), jend(ihread)
            < WORK2(i,j) >
         end do
      end do
!$OMP END PARALLEL DO
```

When a `!$OMP PARALLEL DO` directive is used on a loop in 1-level parallelism the loop iterations must be data independent, and consequently synchronization inside the region makes no sence. Since version 1.0 of OpenMP seems to be targeting 1-level parallelism only, the `!$OMP BARRIER` is not, allowed inside of a `!$OMP PARALLEL DO`-directive.

In explicit thread programming the `!$OMP PARALLEL DO` directive is used in combination with changes in the code to create nesting, and in the data compression code a barrier is needed between the inner-levels, as indicated in example 7. We therefore run into problems since a barrier is not allowed inside of a `PARALLEL DO` region. Calling the SGI's global barrier routine `mp_barrier` partly solved our problem, but what we really needed for this construct, was a team barrier synchronizing only threads within the same team.

If OpenMP 2.0 was available and nesting was implemented, the code in Example 7 could have been written as in Example 8. Notice that computing `jbegin` and `jend` will not be needed. Only the number of threads working in each team `p(:)` has to be computed by Algorithm 1 to get the optimal load balance. But best of all, no changes in the code will be needed!

Example 8:

```
!$OMP PARALLEL DO PRIVATE(i), NUM_THREADS(N)
      do i = 1, N
!$OMP PARALLEL PRIVATE(j), SHARED(i), NUM_THREADS(p(i))
!$OMP DO
         do j = 1, w(i)
            < WORK1(i,j) >
         end do
         < ...  >
!$OMP BARRIER
         < ...  >
!$OMP DO
         do j = 1, w(i)
            < WORK2(i,j) >
         end do
!$OMP END PARALLEL
      end do
!$OMP END PARALLEL DO
```

In OpenMP 2.0 the `!$OMP BARRIER`-directive binds to the closest enclosing `PARALLEL` directive. This implementation will therefore create barriers for teams of threads.

Unfortunately, also in 2.0 of the Fortran version nested parallelism still is implementation dependent. We are afraid this will imply that many (most?) vendors still will choose to serialize nested parallelism. But at least one vendor promise to have this feature available in near future [8]. The effect of making true 2-level parallelism implementation dependent is that programs which relies on nesting for scalability, will not be 'performance portable' when coded in OpenMP, and since performance is at the very heart of parallel programming, while portability and ease of programming is the selling argument of OpenMP, we are afraid the lack of performance portability will be held as a strong argument against OpenMP!

The OpenMP Nanos Compiler [4] is a source-to-source parallelizing compiler implemented around a hierarchical internal program representation that captures the parallelism expressed by the user through OpenMP directives and extensions, and the parallelism automatically discovered by the compiler. One of the main features of this compiler is the ability to exploit multiple levels of parallelism. In [4] two sets of extensions to OpenMP is described. One of them is oriented towards the definition of threads groups. These proposed extensions allow 1) the definition of the groups (how many groups, and how many threads in each group); and 2) the assignment of work to the groups (user controlled). The `GROUP` clause can be applied to any parallel construct. We find this an interesting concept which should work well on our problem. It also shows that when nested parallelism is possible to implement for a small research group, with limited resources, there should be no excuses for large vendors not to implement it.

6 Conclusions and future work

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

As always good load balancing is essential in achieving good scalability. This becomes more difficult when applying multilevel parallelism. In section 2 we give an algorithm which, under some well-defined assumptions compute the optimal allocation of threads to tasks. We also show how 2-level parallelism can be implemented in OpenMP, using explicit thread programming, and discuss some of the shortcoming of the current OpenMP directives for implementing the same algorithm using directives.

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

Our ultimate target application for nested parallelism is the numerical simulation of PDE's, using adaptive mesh refinement (AMR) [5, 6]. In AMR grid points are clustered adaptively in regions where they are most needed. Refined grids are created or existing ones removed based upon estimates of the truncation error. Finer grids consists of independent patches, and the work associated with each patch can be done by a team of threads. This problem has the kind of multilevel parallelism discussed in this paper. It also lends itself naturally to SMP-programming as refined patches are created and dismissed as the computation proceeds in an unpredictable way. This makes distributing data evenly hard and expensive, and a (virtually) shared memory programming much more attractive.

Acknowledgments

This project has been supported by a grant from the Norwegian Supercomputing Program with computing time on Parallab's Origin 2000. We also like to thank the Parallab staff for making it possible to run full scale test on their heavily loaded system.

It is with great pleasure we acknowledge the stimulating email discussions with Prof. Eduard Ayguadé of the Polytechnic University of Catalunya on this topic, and we do believe the Nanos compiler developed by his group is an important work in the right direction.

References

[1] Openmp: A proposed industry standard api for shared memory programming. Technical report, `http://www.openmp.org/`, October 1997.

[2] Openmp fortran application program interface, ver 1.0. Technical report, `http://www.openmp.org/`, October 1997.

[3] Openmp fortran application program interface, ver 2.0. Technical report, `http://www.openmp.org/`, November 2000.

[4] Eduard Ayguade, Marc Gonzales, Jesus Labarta, Xavier Martorell, Nacho Navarro, and Jose Oliver. *Nanoscompiler: A research platform for openmp extensions.* 1999.

[5] Marsha Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53, 1984.

[6] Marsha J. Berger and Randall J. LeVeque. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. *SIAM J. Numer. Anal.*, 35(6):2298–2316, 1998.

[7] William Gropp and Rusty Lusk. *Using MPI.* The MIT Press, 1994.

[8] Jonathan Harris. *Private communication.* 2000.

[9] Maria Lucka and Tor Sørevik. Parallel wavelet based compresiion of two-diemsional data. In *Proceedings of Algoritmy 2000*, 2000.

[10] J. M. Bull, Measuring Synchronisation and Scheduling Overhead in OpenMP. In *Proceedings of First European Workshop on OpenMP*, Lund, Sweeden, Sept. 99, pp. 99-105