Nested parallelism: Allocation of processors to tasks and OpenMP implementation

Ragnhild Blikberg[*]
Dept. of Informatics,
University of Bergen,
NORWAY

Tor Sørevik[†]
Dept. of Informatics,
University of Bergen,
NORWAY

August 4, 2000

Abstract

In this paper an algorithm which distributes processors to teams is presented. The number of processors belonging to each team, depends on the amount of work in the associated task. The algorithm is proved to be optimal, and the complexity is given.

This algorithm can be used in 2-level nested parallelism where each thread in the outer parallelization level becomes the master of a team of threads at the inner parallelization level. Due to the distribution algorithm, the load balance approaches the perfect load balance when the number of threads increases. Since nesting in OpenMP is not implemented on the Origin 2000, nested parallelism has to be done by explicit thread programming. In this paper we will report on the results of parallelizing two different cases, a matrix-multiplication test code and a more real code doing data-compression, using a 2-level shared memory nested parallelism.

Keywords: OpenMP, SMP-parallelism, nested parallelism

1 Introduction

Many computational problems encountered in practice have an outer-level of coarse grained parallelism, where the number of tasks are few, but where each task contains a large amount of work. Each such outer-level task might itself be a parallel task of more fine-grained parallelism. These kind of problems invite to the use of multilevel parallelism, or nested parallelism. While the concept is simple, the implementation of nested parallelism is not, and in most implementations only one of the levels is used. In many cases this gives a far from optimal parallelization. Using only the outer-level parallelism our parallel efficiency may be hampered by too few tasks or poor load balance, as one task may dominate the computation. Using only inner-level parallelism speed-up may be prohibited by the attempt to use to many processors in a fine-grained parallelism.

Obviously the limitation of using only 1 level of parallelism is more apparent as more processors are used, Thus we believe efficient use of multilevel parallelism to be of great importance for scaling of many algorithms.

To set the stage we first would like to give a motivating example. This is meant to illustrate some of the problems and possibilities 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

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

Applying inner-level parallelization the runtime on P processors will be

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

For nested parallelism (NP) we need to distribute the processors to tasks. Let p_i be the number of

[*] http://www.parallab.uib.no/~ragnhild
[†] http://www.parallab.uib.no/~tors

processors associated with task i. Now the runtime for P processors will be

$$T_{NP} = \max(\frac{w_i}{p_i} + s_i). \quad (3)$$

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

$$T_{NP} \geq \frac{W}{P} + \max s_i. \quad (4)$$

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

$$T_{NP} \geq \frac{W}{P} + \frac{S}{N}. \quad (5)$$

and implies that $T_P > T_{NP}$. 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 CPUs as, $S_p = T_1/T_p$, where T_p is the running time on P processors.

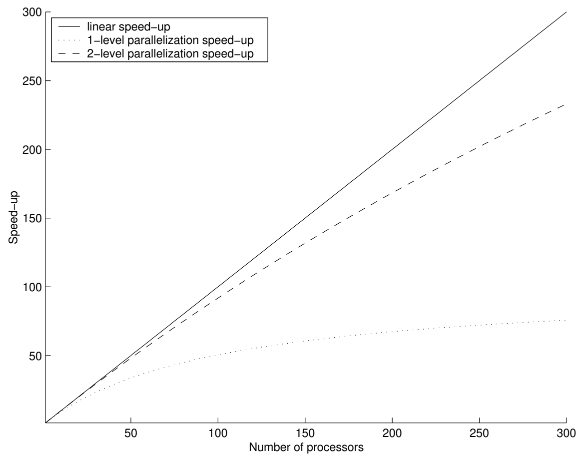

Figure 1: *Speed-up curves for the idealized cases above with $N = 10$ and $S = 0.01W$. The dashed line is for 2-level parallelism obeying the r.h.s. of (5), while the dotted line is for inner-level parallelism as described by (2)*

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

In section 3, we discuss how to implement 2-level parallelism in OpenMP. First we look at the possibilities and short comings 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, matrix multiplication, and a real life problem, a wavelet based data compression code. In section 5 we will discuss the extension we find necessary concerning nesting in OpenMP. 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 processors, the most coarse grain parallelism is achieved by assigning multiple tasks to each processor and not using any inner-level parallelism. In this case achieving the optimal load balance reduces to the standard bin-packing problem, assuming there is no dependencies between the different tasks. We will not consider this case here and therefor assume that the number of available processors is larger than the number of outer-level tasks.

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

In the sequel we will give a formal definition of our distribution problem, formalize the above algorithm and prove it optimality. For efficiency we store the tasks in a heap. Then finding the task with highest 'work-to-processor-ratio' is done in $O(1)$, while we need $O(\log N)$ 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 processors ratio. The workload of a task is given by its weight, denoted w_i.

Definition 1{The optimal distribution prob-

lem}:

Find an optimal distribution of the processors to N tasks such that $\max_i \left(\frac{w_i}{p_i}\right)$ is minimized given the constraints: $\sum_{i=1}^{N} p_i = P$ and $p_i;\ \ i = 1, ..., N$ positive integers.

Algorithm 1: Distribution of processors 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: For any iteration, j, of the above algorithm $\frac{w_i}{p_i - 1} \geq \frac{w_1}{p_1}$ for $1 \leq i \leq N$.

For simplicity we we assume $\frac{w_i}{0}$ to be a well define number larger than any other positive integer. The proof holds without this dubious assumption, but without it we need to treat this as a special case and the proof becomes rather messy.

Proof: The lemma is proved by induction on j:
A) The lemma is true for $j = 1$ since $\frac{w_i}{0} > \frac{w_1}{1}$.
B) Assume the lemma is true for fixed j.
Let $rmax(j) = \max_i(\frac{w_i}{p_i})$.
Then for $j + 1$: $rmax(j + 1) = \max(\frac{w_1}{p_1+1}, \max_i = 1, ..., N \frac{w_i}{p_i}) \leq \frac{w_1}{p_1} = rmax(j)$.
Then since $\frac{w_i}{p_i - 1} \geq rmax(j) \geq rmax(j + 1)$ for $i = 1, ..., N$ 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{Optimality}: 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 $q_1, ..., q_N$ which gives $\frac{w_1}{q_1} < \frac{w_1}{p_1}$ and consequently $q_1 > p_1$. Since $P = \sum_{i=1}^{N} q_i = \sum_{i=1}^{N} p_i$ it therefor must exist a k such that $q_k < p_k$ for a $1 < k \leq N$. But then $\frac{w_k}{q_k} \geq \frac{w_k}{p_k+1}$ and by Lemma 1 $\frac{w_k}{q_k} > \frac{w_1}{p_1}$ which contradicts our assumption that $q_1, ..., q_N$ 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, creating communicators will make it possible to form groups of processes 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 directly program the specific tasks of each thread is also possible. The explicit approach gives the programmer full control, but do require a much higher level of programmer intervention. Therefor directive based SMP programming is usually the recommended approach. Below we discuss the possibilities and limitation 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 standard for SMP-programming, does however allow some form of nested parallelism. The OpenMP group released its Fortran standard in October 1997 [whp97], [omp97], and the first commercial available implementation of this was available on SGI's Origin 2000 in the spring of 1998.

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 use the subroutine `OMP_SET_NESTED`.

Example 1:

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

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 have implemented nesting.

SGI's MIPSpro compiler has a restricted form for nested parallelism. This does however not follow the OpenMP standard. It applies only to do-loops in Fortran and demands that the loops should be perfectly nested (Loops are *perfectly nested* in Fortran if there is no code between the two `DO` statements and between the two `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
```

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 for the do-loops in the example above in order to achieve nested parallelism:

Example 3:

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

The values of the arrays `mytask`, `jbegin` and `jend` have to be carefully decided by the user in order to assure data-independency and to make sure that the exact same computations are done in parallel as in sequential. Since the values of these vectors decide the distribution of work to threads, they dictates the load balancing. In the examples presented in the next section, each item of `WORK(i,j)` require the same amount of work. However, the range of the second index is a function of the first. This means that good load balance is achieved if `jend(i,proc)-jbegin(i,proc)` is approximately the same for all values of `i` and `proc`. In our examples `mytask` is computed using the algorithm given in chapter 2.

Suppose that nested parallelism was implemented, and let us compare the OpenMP nested loops in Example 1 with the hand-coded nested loops in Example 3. The hand-coded case will probably produce less parallel overhead than the OpenMP case, since the threads are only created once for the hand-coded case, and this case only needs one synchronization. In the first case, when nesting is done by two directives in OpenMP, new threads are created twice and its also necessary to synchronize twice.

The kind of programmer intervention needed for these changes are to some degree similar to the work needed when parallelization using MPI [GL94]. In both cases the programmer has to split the work and allocate it to specific threads/processes. The correctness of the program is her full responsibilities. The advantage is that, assuming a (virtual) shared memory, no explicit communication is needed for OpenMP, and addresses are still globally unique.

4 Experiments

In this chapter we report on two experiments on 2-level parallelism. The load balancing is done using the work allocation algorithm presented in chapter 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 of the importance of utilizing multilevel parallelism, we made an artificial testcode, using a simple matrix multiply as the computational kernel.

Suppose we have N tasks, where a task, i, is a multiplication of the matrixes $A_{m\times w_i}$ and $B_{w_i\times m}$, where $i = 1, \cdots, N$. Each task has a different amount of work proportional to the weight w_i. Again we assume that the number of processors is larger than N, the number of tasks. 1-level parallelization of the matrix-multiplication can be implemented like this, assuming $N < P < \min_i w_i$.

Example 4:

```
      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
```

To facilitate load balance between tasks, the loop above are rewritten as

Example 5:

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

These two cases have been tested for $m = 500$ and for $N = 1, 10$. w_i is chosen uniformly random from the interval $500 \leq w_i \leq 5000$.

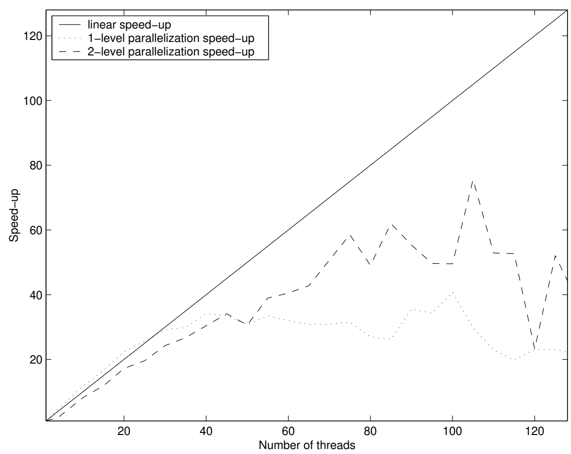

Figure 2: *Linear speed-up, speed-up for 1-level inner-loop parallelization and speed-up for 2-level parallelization for $N = 4$ outer-level tasks of matrix multiplication.*

In Figure 2 we display the linear speed-up, together with the speed-up achieved for 2-level nested parallelization and 1-level parallelisation as a function of threads. For this particular case the number of outer-level tasks is $N = 4$. The runs are done on a susped Origin 2000 using MIPSpro Fortran Compilers, Version 7.3.1.1m.

For up to about 25 threads the inner-level parallelization shows superlinear speed-up, probable due to cache effects. However, as the number of threads increases some unavoidable overhead starts to creep in for the 1-level parallelism and the achieved speed-up starts to fall behind the linear speed-up. In the 2-level case each outer-level task applies the same 1-level parallelism, but with fewer threads. Thus we expect the 2-level parallelism to stay closer to the linear speed-up longer as the number of threads increases. This is exactly what we see in this figure. The jagged shape of is an artifact of the unpredictable behavior of the O-2000's system software and the fact that the test was not run on a dedicated system.

In our case the crossover point where 2-level parallelism ourperform 1-level is at approximately 45 threads.

This example modify slightly the assumption in the introduction. It does not assume 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 Datacompression

To try our ideas on a more real test case, we moved on to a datacompression 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 [LS00]. 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 N first binary digits are significant. Thus without any further loss of accuracy the wavelet coefficients can be represented by only N bits, giving us an extra saving factor of N/64.

Encoding/Decoding: After the wavelet coefficients have been massaged by thresholding and quantization they are encoded. In the parallel version this is done by deviding the data in P separate streams.

The 2d-wavelet transform: The wavelet routine only works for arrays $m \times n$ where m and n are integer power of 2. Thus we first chop up the array in `Nfrag` 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, Nfrag, N`

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

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 1792x1792. 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.

In this example we can not expect perfect load balance due to the integer restriction on the number of processors. 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 sharper bound on the 2-level speed-up. We may define $\hat{T}_p = \max_i T_i/p_i$ as the theoretical minimum run-time of 2-level parallelism. The theoretical maximimum 2-level speed-up is than given as $\hat{S}_p = T_1/\hat{T}_p$.

In Figure 3 we display the linear speed-up and theoretical maximal 2-level speed-up, together with the speed-up achieved for 2-level and 1-level parallelisation. The runs are done on a dedicated Origin 2000 using MIPSpro Fortran Compilers, Version 7.3.1.1. 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 speed-up top at about 20 threads, while the 2-level parallelized code increases its speedup at least until 64 threads, where the speedup is 33. Especially notice that for less than about 40 threads, the speed-up of the 2-level parallelized code has the same shape as the theoretical 2-level speed-up. The result of this example proves that there can be a lot to gain parallelizing a real test code in 2 levels.

5 Needed extensions of OpenMP

In the previous section we showed how to implement multilevel parallelism in OpenMP using explicit thread programming. But as we argued in

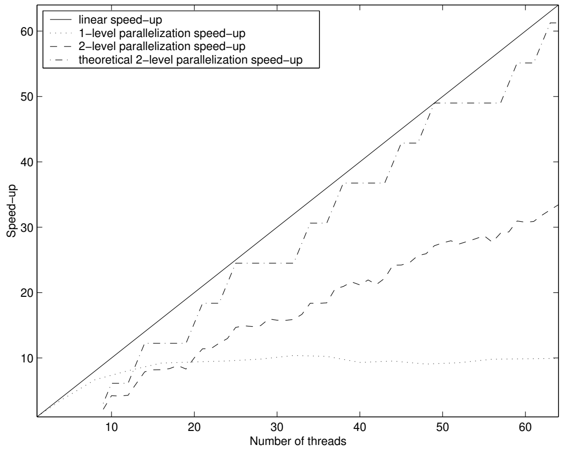

Figure 3: *Linear speed-up, theoretical 2-level speed-up, speed-up for 1-level inner-loop parallelization and 2-level parallelization for a 1792x1792 datacompression problem.*

section 3 for the each of programming we would prefer doing this with directives and/or function calls only. In this section we will discuss briefly the current shortcomings of OpenMP and what the needed extensions to enable directive based multilevel parallelism.

Setting the number of threads by call to the `OMP_SET_NUM_THREADS` routine in OpenMP is only legal outside of 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, there will be no additional threads to available at the inner-level, and nested parallelism is not obtained.

In draft 9 of OpenMP 2.0 it is proposed a `NUM_THREADS`(*scalar_integer_expression*) clause 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 tell it solves the problem described above of spawning additional threads in multilevel parallelism Using this clause there will be no need for placing `OMP_SET_NUM_THREADS` inside a parallel region. Moreover, the sequential code need not differ from the parallel code.

When parallelizing the data compression code in 2 levels, it was necessary to have two inner-level loops with a synchronization barrier in between. Unfortunately the barriers of OpenMP 1.0 `!$OMP_BARRIER` is not allowed inside a parallel region, in this case the outer parallel loop, and we had to call SGI's routine `mp_barrier`. This is however a global barrier, but what we really needed at for this construct, was a team barrier synchronizing only threads within the same team. In the next version of OpenMP a team barrier should be included in the standard, in addition to other synchronization constructs which should only apply to a team of threads. This is however not proposed in the draft of OpenMP 2.0, but we think that these extensions should be included in connection with the other improvements concerning nesting in OpenMP.

Unfortunately, it is proposed in the draft of OpenMP 2.0 that nested parallelism should still be implementation dependent. No doubt this will imply that many (most?) vendors still will choose to serialize nested parallelism. The effect of this is that programs which relies on nesting for scalability, will not have be 'performance portable' when coded in OpenMP, and since performance is at the very hart 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!

A proof a feasibility of these extensions to OpenMP is given by the Nanos project. The OpenMP Nanos Compiler [AGL+99] 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 [AGL+99] two sets of extension to OpenMP is described. One of them is oriented towards the definition of processors groups. These proposed extension 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.

6 Conclusions and future work

The main contribution of this paper is that we show by theoretical and practical examples the need of utilizing multilevel parallelism when it is available in the problem at hand. In particular how it improves on the scalability of the problem.

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 given assumptions compute the optimal allocation of processors 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 OpenMP directives for implementing the same algorithm using directives.

As we see larger SMP-systems becomes more and more common, the scalability of OpenMP becomes more important. Utilizing multilevel parallelism will become an important issue in this context. The suggested extension for OpenMP 2.0 points in the right direction, but still important features are missing. In particular the possibility of setting team barrier. We are also unhappy with with the fact 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) [BO84, BL98]. In AMR grid points are clustered adaptively in regions where it is most need of close grid points. Refined grids are created or existing ones removed based upon estimates of the truncation error. Each grid makes a patch, and the work associated with each patch can be done by a team of processors. This problem has the kind of multilevel parallelism as we discuss here. It also lends it self 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.

Acknowledgment

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 Eduard Ayguadé of the Polytecnic University of Cataluna i on this topic, and we do believe the Nanos compiler developed by his group is an important work in the right direction.

References

[AGL+99] Eduard Ayguade, Marc Gonzales, Jesus Labarta, Xavier Martorell, Nacho Navarro, and Jose Oliver. Nanoscompiler: A research platform for openmp extensions. 1999.

[BL98] 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.

[BO84] Marsha Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53, 1984.

[GL94] William Gropp and Rusty Lusk. *Using MPI.* The MIT Press, 1994.

[LS00] Maria Lucka and Tor Sørevik. Parallel wavelet based compresiion of twodiemsional data. In *Proceedings of Algoritmy 2000*, 2000.

[omp97] Openmp fortran application program interface, ver 1.0. Technical report, `http://www.openmp.org/`, October 1997.

[whp97] Openmp: A proposed industry standard api for shared memory programming. Technical report, `http://www.openmp.org/`, October 1997.