Early experiences with OpenMP on the Origin 2000

Ragnhild Blikberg[*]
Parallab,
University of Bergen,
NORWAY

Tor Sørevik[†]
Parallab,
University of Bergen,
NORWAY

September 7, 1998

Abstract

OpenMP has been marketed as THE emerging standard for shared memory parallelism (SMP). The first compiler for OpenMP is now available on the Cray Origin 2000. In this paper we report on some early experiences with this compiler on a (quasi-)application code, an implementation of the NAS, BT benchmark.

OpenMP includes, of course, the traditional do-loop parallelization. For programmers familiar with Power Fortran this is simply a question of adapting to a slight modification of syntax. Not surprisingly the performance equals that of Power Fortran. This part is easy and straightforward. The interesting thing about OpenMP is that it goes beyond the simple do-loop parallelization and opens possibilities for a more course grain parallelism by splitting the programming in different independent tasks (Parallel sections) and/or creating parallel regions. Combining this with the more fine grained do-loop parallelism keeps the promise of higher scalability than one usually find in SMP-programming. Allowing for nested parallelism, OpenMP gives us this opportunity. With appearance of SMP-system scaling beyond 128 processors this is very important new features.

In this paper we focus on those features of OpenMP which extend the model for parallel computing already available with Power Fortran. To get a feel for how it works on real application we have applied the techniques to a BT-code for the NAS parallel benchmark and report results on short coming of this first release of the compiler as well as speed-up and scalability results. Comparison with MPI will also be provided.

Keywords: OpenMP, SMP-parallelism, parallel benchmark

1 Introduction

One of the major obstacles to progress of parallel computing for large scale scientific problems has been the lack of standards. Users have for good reasons been reluctant to take the burden of parallelizing their codes when they can't be sure that the investment will pay off when moving to a different parallel computer.

This problem has been overcome for the message passing programming model with the introduction of MPI [GL94] and for data parallel programming with HPF [KLS+94]. For directive based Shared Memory Programming (SMP-programming) no such standard exists. Efforts have been made, most noticeable the ANSI X3H5 standardization effort[Le94]. But their proposal was never formally approved and no successful implementation ever reached the market. It is therefore very welcome in the user community that a vendor driven group now tries to agree on an industry standard for SMP-programming. The new standard is called OpenMP.

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

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 this report we will describe some of the features of OpenMP and some early experiences with it on our Origin 2000 system. In particular we focus on features which are new compared with SGI's old parallelizing Power Fortran.

This paper is organized as follows. First we give a short description of some of the main features of OpenMP and comments on some of the features we miss most. Then we give a brief description of our target test application, the NAS-BT parallel benchmark. In section 4 we report on our experience using OpenMP on this application. Speed-up is measured and compared with MPI.

2 The OpenMP standard

The OpenMP API (Application Programming Interface) is a set of directives to instruct the compiler when to *fork* multiple threads and where to *join* these threads again. An associated runtime library and a set of environmental variables are also defined. Here we only give a very brief overview of the most important directives and their scope.

The key directive is `C$OMP PARALLEL` which defines a parallel region. A parallel region is a block of code which is to be executed by multiple threads. It ends by `C$OMP END PARALLEL`. Different clauses might be added to the `PARALLEL`-directive. In particular one might want to define data as `PRIVATE` or `SHARED`. It's also possible to specify the reduction of `SHARED` data.

The most obvious way of specifying work to different threads is distributing independent iterations of do-loops to the different threads. This can be specified by the `C$OMP DO` and `C$OMP END DO` directives, embedded in `C$OMP PARALLEL` and `C$OMP END PARALLEL`. There is also the shorthand notation of `C$OMP PARALLEL DO` which is handy in cases where the parallel region contains only one do-loop.

Since the dawn of parallelizing compilers, all directive based parallelizing compilers have had the ability of parallelizing do-loops, and it is still, by a large margin, the most used feature. For many applications, however, parallelizing the outer do-loop is not sufficient. When the number of iterations in the outer loop is to small, it only allows for limited parallelism. In other cases parallelizing do-loops only allow for relatively fine grain parallelism, although the problem itself might posses a coarse grain parallelism on a higher level. To assist with the first problem, OpenMP allows for nested parallelism. This can be applied to a set of nested do-loops.

A way of specifying independent tasks, not specified by different iteration of a do-loop, is provided by the `C$OMP SECTION` directive, as in:

```
C$OMP SECTIONS [clause[[,]clause],...
C$OMP SECTION
a block of code
C$OMP SECTION
a block of code
..............
C$OMP END SECTIONS
```

This will cause the different blocks of the code to be executed in parallel. In cases where a block of code contains parallel tasks, nested parallelism might be used in this case too. We would claim that to allow for flexibility in the choice of the number of processors to use, it is necessary to use nested parallelism in combination with `C$OMP SECTION`, as the `C$OMP SECTION` specifies a fixed number of parallel tasks.

To ensure control in multi threaded programming, means of synchronization and exclusive control for one thread, are necessary. A handful of handy directives like `C$OMP BARRIER, C$OMP MASTER, C$OMP CRITICAL` and `C$OMP ATOMIC` are available.

The runtime libraries provide functions for, at runtime, finding the total number of threads running and the specific number of one particular thread. You are also allowed to specify whether you want dynamic or static adjustment of thread numbers and to set locks enabling synchronization of threads.

To a large degree OpenMP is only standardizing syntax for features available on most parallelizing compilers, but some features are not universal available on all parallelizing compilers. The two most significant extensions are the `C$OMP SECTIONS` and the possibility of nested parallelism.

2.1 Missing features

The most notable feature we are surprised not to find, is the lack of support for parallel directives applicable to Fortran 90/95 array syntax. Many of the most obvious parallel do-loops translates into the neat array syntax when programmed in F90-style. It seems awkward that an API for Fortran, designed in 1997, does not address the current Fortran-standard, but relates to non-standard Fortran!

We did also miss the opportunity to do reduction on arrays. For reasons unknown to us, the standard explicitly states that reduction variables must be singular. You are however, allowed to have a list of them. Thus the following directive is legal:

```
C$OMP PARALLEL DO REDUCTION(+:  var1, var2,var3,var4,var5)
```

while

```
C$OMP PARALLEL DO REDUCTION(+:  var)
```

where `var` is declared

```
DOUBLE PRECISION, dimension(5)::  var
```

is illegal.

3 The NAS-BT benchmark

The NAS-BT parallel benchmark is a pseudo CFD application [BHW+95]. It simplifies a full application code by simplifying the boundary conditions and I/O. It also applies a regular structured grid, which is not always the case for real application codes.

It sets out to solve a formulation of the 3-d Navier-Stokes equations, using a finite difference scheme in the spatial discretization. Time integration is done by an implicit scheme. This implies the solving of a large system of algebraic equations in each time step. The non-linearity of the partial differential equations is inherited by the discretized system. The nonlinear terms are however, expanded in Taylor series and truncated after the linear terms. Thus effectively we are left with a linear system.

Each grid point holds 5 variables. Their solution is coupled, implying that all over the place we have 5x5 blocks of matrices. In each time step the coefficient matrix (The Jacobian) as well as the right hand side are updated. The same problem are solved for all the 3 NAS-application benchmarks, but three different techniques are applied to solve the linear system.

In the NAS-BT we apply an iterative scheme, the SSOR method. In this case we split the original system matrix in an upper and a lower block triangular part. A new iterate is obtained by solving a lower block triangular system followed by an upper block triangular system.

The algorithm for one time step is then:

1. Compute RHS (right hand side).
2. Form and solve the lower block triangular system.
3. Form and solve the upper block triangular system.
4. Update $U(n+1) = U(n) + \Delta d$.

These steps are sequential in nature, thus we need to search for the parallelism within each step. Luckily there is a lot of it. Step 1 and 4 are embarrassingly parallel and so is building the matrices in step 2 and 3. The solution of the triangular systems turns out to separate nicely into independent block tridiagonal systems.

All data is stored in 5-d arrays. 5x5 blocks of 3-d data. The data access pattern shifts as we go along. Typically one access 3 5x5 blocks which are consecutive along one of the spatial dimensions.

4 Experiments

We have taken the public available sequential implementation of the NAS-BT benchmark, inserted the necessary compiler directives and let the F77-compiler on the Origin 2000 parallelize it. At some places the code needs careful examination in order to disclose which variables to declare as private and which ones should be shared. As usual the key thing is to find the outer do-loops available for parallelization. The code consists of 30 routines with a total of 3400 lines of Fortran. The initial porting to standard do-loop parallelism took only 2 days. Most of the time was spent in examining the body of the do-loop to determine which variables could be `SHARED` and which one had to be `PRIVATE`.

Our experiments focus on testing the reliability and functionality of SGI's OpenMP compiler, as well as its efficiency. We also played around with the `C$OMP SECTIONS` directive and will report on our experience with this. Results reported in this paper are run on a system in full production mode with a relative high load. Resource allocation is managed by the system and occasionally we observe unpredicted slow down, only possible to explain by shortage of resources due to the systems allocation policy. All codes are compiled with `-O3` for optimization.

4.1 Reliability and functionality

To be a first release the compiler is pleasantly reliable. So far we have not encountered any obvious bug in the compiler. We are missing one of essential feature; The possibility for nested parallelism.

4.2 On parallel sections

Updating RHS is done in 6 different routines. Each routine computes its contribution to the new right hand side. The computation of the different contributions are totally independent, and as such a perfect case for the parallel section technique. Ideal, all one should need to do would be inserting the `$OMP PARALLEL SECTIONS` directives with the `REDUCTION` clause as shown below

```
C$OMP PARALLEL SECTIONS REDUCTION (+:  rsd)
C$OMP SECTION
      call rhs_xa (nx, ny, nz, ru, q, rsd, u)
C$OMP SECTION
      call rhs_xb (nx, ny, nz, rsd, u)
C$OMP SECTION
      call rhs_ya (nx, ny, nz, ru, q, rsd, u)
C$OMP SECTION
      call rhs_yb (nx, ny, nz, rsd, u)
C$OMP SECTION
      call rhs_za (nx, ny, nz, ru, q, rsd, u)
C$OMP SECTION
      call rhs_zb (nx, ny, nz, rsd, u)
C$MP END PARALLEL SECTIONS
```

The use of the `REDUCTION` clause is, however, not allowed in this case as `rsd` is an array. Thus we were forced to extend `rsd` by an extra dimension, of size 1:6, and store each local contribution separately. An extra piece of code was added to do the reduction over this dimension after all the parallel sections were done. These changes to the code, violate one of the most attractive feature of directive based SMP-parallelization. That only one code is necessary for running in parallel as well as sequential mode.

The number of arithmetic operations is exactly the same in the 3 routines `rhs_*a`, but different from the numbers of arithmetic operations in the routines `rhs_*b`. The data is, however, accessed in different order in the x, y and z cases. This makes the execution time different for the 6 sections. Timing for a typical iteration is shown in Figure 1, and as shown here a static sectioning leads in this case to an inevitable load imbalance.

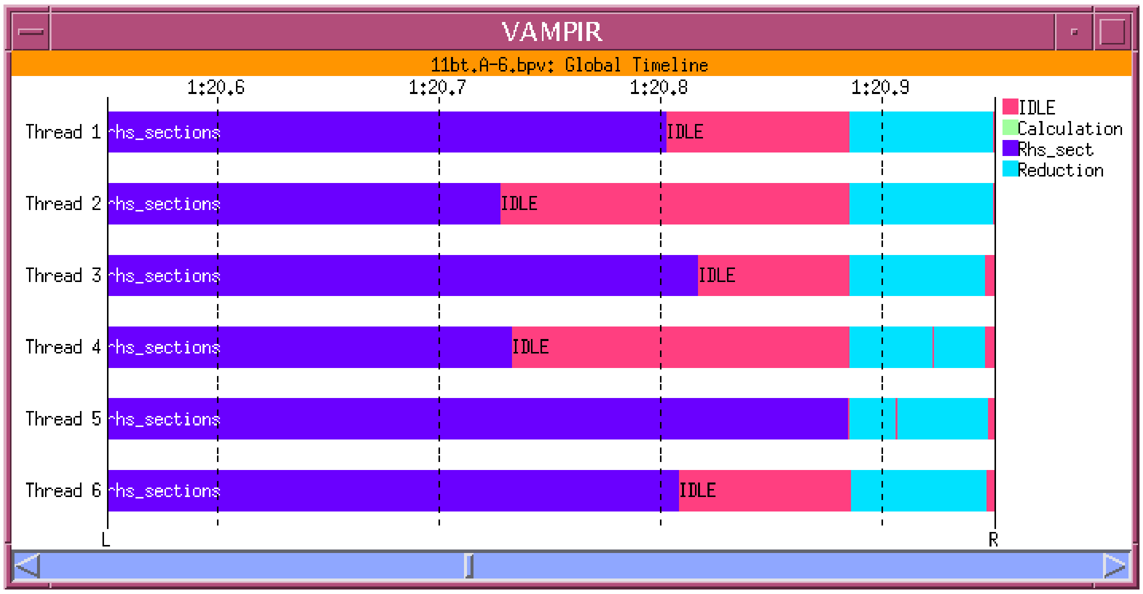

Figure 1: Execution time as recorded by VAMPIR for parallelizing the code above using `C$OMP PARALLEL SECTIONS`.

As noted above, we were forced to add an extra dimension to our arrays to apply the `PARALLEL SECTION` directives, due to the missing functionality of the `REDUCTION` clause. On a Distributed Shared Memory (DSM) system, like the Origin performance obviously is influenced by the distribution of the data. Done correctly we are able to force the system to allocate the data at the processor who are responsible for calculating the corresponding contribution to the right hand side. However, this is the worst possible distribution for the reduction step as the data now is distributed across all the participating processors. As seen in Figure 1 the time for doing the reduction is about 20% of the time calculating the different contribution. A simple operation count does however indicate that it only amounts to 5 % of the arithmetic work. We claim that this situation is not unique to our example, but is quite likely to happen in situations like this, where different processors computes independent contribution to overall results, and a global reduction is needed to accumulate the 'grand total'.

Another non-desirable feature is also illustrated in this example. Dividing into a fixed number of sections allows for little flexibility in the number of processors to parallelize over. In this case the only reasonable choices are 2, 3 or 6 respectively. Since each of these tasks are rich in fine grained loop level parallelism, this form of parallelism could be applied as well. For OpenMP we could then apply an outer course grain parallelism using `C$OMP SECTION` and an inner fine grain do-loop parallelism. By combining the two level of parallelism we should be able to obtain the best possible scalability combined with good load balancing. This strategy is however, not yet available to us as the current implementation of SGI's OpenMP does not support nested parallelism.

4.3 Efficiency of do-loop parallelization

In this section we report on our experience using `C$OMP PARALLEL DO` directives on the do-loops. Standard SGI profiling, *ssrun* and *prof* (pc-sampling) [Gra], shows that 99.7% of the code is being parallelized.

Our analysis of the runtime behavior of the program is based on the use of the VAMPIR (Visualization and Analysis of MPI Resources) performance visualization tool [NAW+96] [Gmb], which has been rebuild by Zentrum für Hochleistungrechnen (ZHR) at the Technical University of Dresden to be able to analyse OpenMP programs as well as MPI programs.

The outer loop of the code is representing time stepping. As always, time is sequential, thus parallelism is achieved by distributing the update of the different grid points to processes within a time step. By using VAMPIR it is easy to confirm that the behavior of a parallelized loop is similar from one time step to another within the same run. Therefore, wishing to investigate a parallelized loop, it is sufficient to pick one arbitrary time step.

For all of the parallelized loops in this code the operation count is exactly the same in all iterations. Thus provided the number of threads divides the number of loop-iterations, the workload should be perfectly balanced. Observing the execution time of a sequence of parallelized loops, we do, however, see that the execution time is far from evenly distributed across the threads. Typically the fastest threads for one loop will be the fastest threads for the other loops as well. A typical scenario is illustrated in Figure 2.

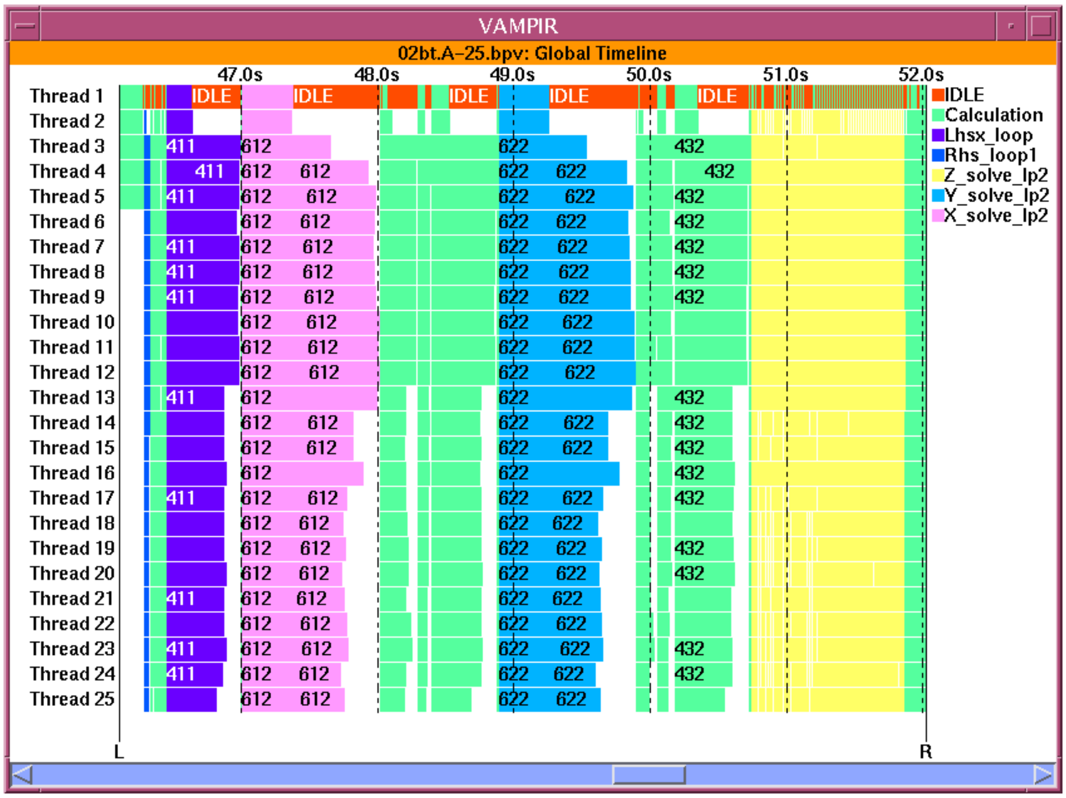

Figure 2: Work performed by 25 threads in one time step as displayed by VAMPIR.

Here we display the work done for each thread during one time step on a run for 25 threads. The time line shows the wall clock time in seconds. The different loops are showed with different colors. The numbers showing on some of the loops are the loop identity numbers, and one number represents one loop iteration. These are showing if there is room for them. If there is even more room, the name of the loop is showing instead of the loop identity number. White areas represents no act and the red area on thread 1 is the time spent waiting for the rest of the threads to finish.

By inspection we can see that the pattern is repeating itself. The fastest threads in one time step, continue to be the fastest in the next and so forth.

Figure 3 zooms in on one loop. For a high number of threads, thread 1 and 2 are really working fast (3ms each iteration), while the rest of the threads are working much slower (19ms for their first iteration). Since differences in execution time can not be explained by differences in arithmetic work, and we know that each thread had the exclusive attention of one CPU, our prime suspect for this huge differences in execution time is differences in data access. The suspicion is under pinned by an inspection of the code

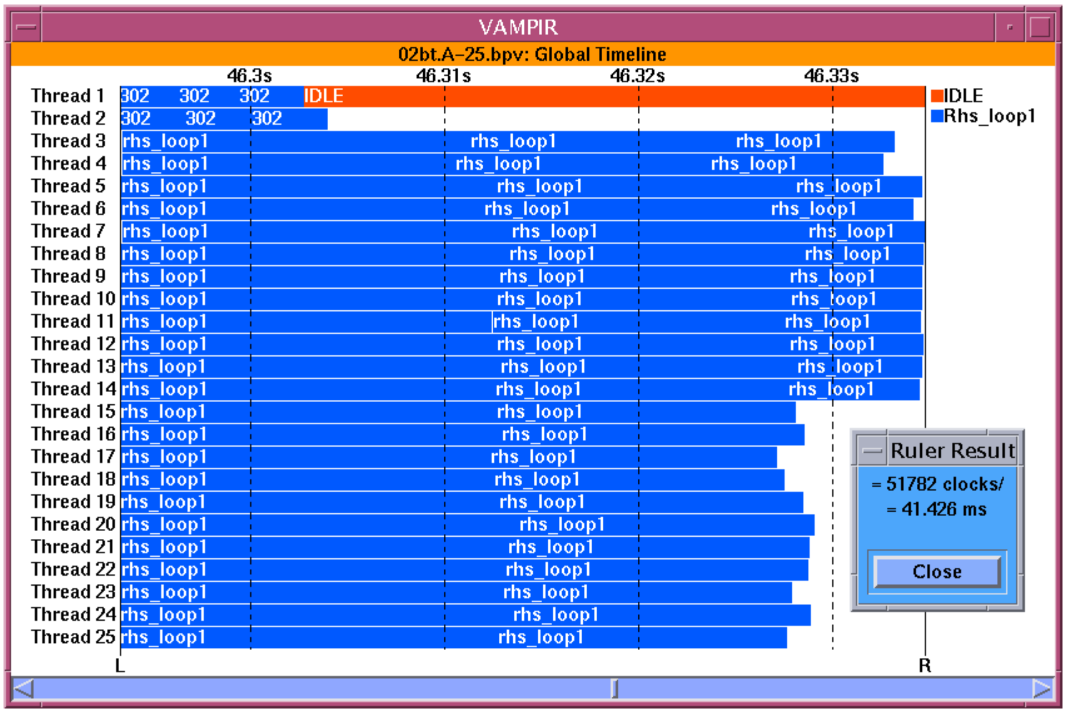

Figure 3: Execution time of 25 threads for one parallelized loop. 14 threads do 3 iterations of the loop while 11 do only 2.

which shows that to do 12 flops a CPU needs 6 double precision numbers, and no reuse of data is possible. Thus this piece of code is very data intensive.

The CRAY Origin 2000 memory is physically distributed across nodes, with two processors located at the same node, having equal access to this memory. Remote CPUs also have direct access to this memory, but have a longer way to go. Moreover, if many processors compete for access to one node's memory simultaneously, the single access channel to this memory is doomed to be a bottleneck.

The fast work of thread 1 and 2 indicates that these two are located at the same node as the data needed for the computation. While the slow working threads do not have the data located at their node. Therefore we can conclude that the speed of the threads depends strongly on the location of memory. References to location in the remote memory of another processor takes longer to complete than references to locations in local memory. This can severely affect the efficiency of a program that suffer from a large number of cache misses.

When most of the threads are finished, the threads still working starts to work faster (6ms each iteration), showing how the speed depends on the number of threads accessing data from the same memory location. Note that the time from start to end for this loop is 41ms.

We also found that the forking and joining of the loops (parallel overhead) is not time-consuming, and can not be used as an explanation for the poor speed-up seen in this example.

4.4 Data distribution

To get good performance in programs which is bandwidth limited, it is important to distribute data across the memory modules to increase the aggregated bandwidth, and then try feeding the caches of individual processors from local rather than remote memory. On a Distributed Shared Memory system like the Origin 2000 it therefore makes lot of sense to distribute not only work, but data as well. This is acknowledged by SGI in their Power Fortran were they provide directives controlling data distribution, as well as the `AFFINITY` clause to the `DOACROSS` directive, which enable us to glue the thread working on a piece of data to a processor sitting on the same node as the memory keeping the data.

The data distribution directives is not a part of OpenMP, but is available on the Origin Fortran compiler. To apply the affinity clause, one need however to replace `C$OMP PARALLEL DO` with Power Fortran's `C$DOACROSS`, so this was not used.

```
C$OMP PARALLEL DO PRIVATE(i,j)
      do j = 1,ny
         do i = 1,nx
            A(i,j) = i*j + A(i-1,j)
         end do
      end do
```

If A is declared as `A(nx,ny)`, this loop suggests a distribution

```
C$DISTRIBUTE A(*,block)
```

This distribution directive divides the data in blocks of size $nb = \lceil \frac{ny}{P} \rceil$ and tells the compiler to allocate memory for column i on the same node as processor $\lceil \frac{i}{nb} \rceil$. Note that in case $ny \bmod P \neq 0$, some of the processors will get fewer or no columns. For the example displayed in Figure 4, $ny = 64$ and $P = 25$, we notice that only the 22 first processors get any work, and the last only perform one iteration over the columns. Except for this feature we now get a much better distribution of work and for this particular case the time from start to end for this loop is only 13ms compared to 41 ms for the previous one!

4.5 Scheduling

Another possibility for evening out the load imbalance due to differences in speed of the different threads, is applying a different scheduling of tasks to threads by using one of OpenMP's environment routines. `SCHEDULE(DYNAMIC,`*chunk*`)` specifies another scheduling; the loop iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the loop space, it dynamically obtains the next set of loop iterations. When no *chunk* is specified, it defaults to 1. We added `SCHEDULE(DYNAMIC)` to the `C$OMP PARALLEL DO` directive for the same loop as referred to earlier. Now, when each thread finishes a loop iteration, it dynamically obtains a new loop iteration. The result is seen in Figure 5. Now the different threads were kept working almost equally long, but almost none reduction in the overall running time was observed. This comes as no surprise as the scheduling is not any cure for fixing the fundamental problem, the memory bandwidth bottleneck.

We also tried combining data distribution with dynamic scheduling. This kept the processors busy, but the overall execution time went up compared to only using data distribution. We strongly believe the reason for this is that the dynamic scheduling breaks the affinity of threads to local data, and thus force the processors to non-local data access.

4.6 Comparing OpenMP with MPI

We wanted to compare the OpenMP version with the public available MPI version of the same NAS-BT as used above. One drawback of this version is that it is only possible to run when the number of processors is a quadratic number (1, 4, 9, 16, 25, ...).

Comparing MPI with OpenMP, the most obvious result is the much better speed-up for MPI than for OpenMP. This is seen in Figure 6.

OpenMP and MPI both achieve their parallelizm by dividing the computation of the spacial gridpoints to the processors. Besides applying different programming model, the parallelism differs in how to divide data and the parallel granularity. To minimize the communication the MPI code applies a 2 dimensional partition of data and thus achieve a better 'surface-to-volume' ratio than the OpenMP, which due to the

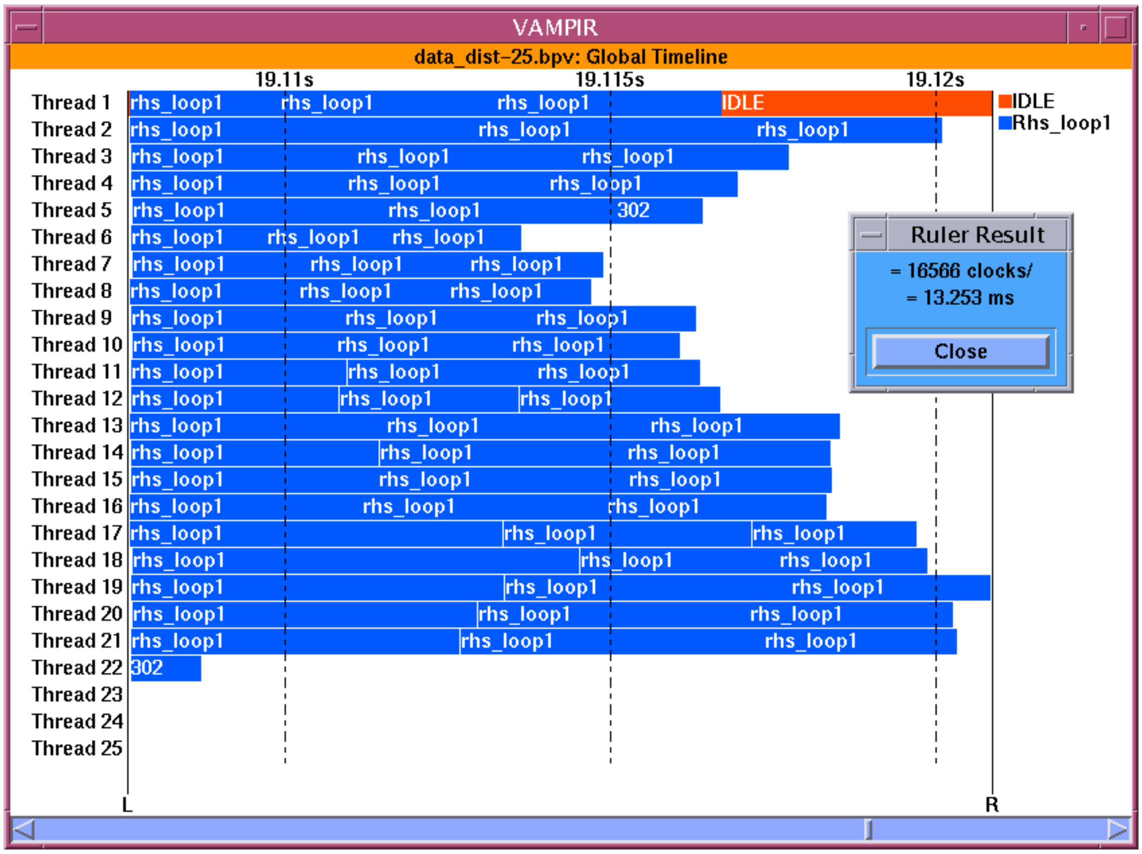

Figure 4: Visualization of the same loop as in Figure 3 run on 25 threads where data distribution is used. We see that the loop iterations are differently distributed among the threads.

lack of nested loop-parallelism has no choice, but slicing the data in 1 dimension only. A second difference is the parallel granularity. In the MPI code there are far less syncronization points within a time step than for the OpenMP code. Large chunks of data is exchanged between processors up front, giving each of them a juicy piece of work to chew on before further syncronization is needed.

Appearently the same coarse grain parallelism should be possible to obtain in OpenMP by letting `C$OMP PARALLEL` encapsulate multiple do-loops and allowing a thread to continue on the next loop as soon as the first is finished. This is accomplished using a `NOWAIT` clause to the `C$OMP END DO` like this:

```
C$OMP DO
        do-loop
C$OMP END DO NOWAIT
```

Inserting `NOWAIT` clauses turned out to be an error prone process. To do it correctly the programmer needs to do the same analysis of dependency between different loops, as needed to determine when and where to communicate when programming with MPI. Finding bugs introduced when adding `NOWAIT` was a painful job. The reward was also small in terms of speed-up in our case, as the memory bottleneck still remains, and the thread running slowest for one particular loop turned out to be the slowest one in the next loop as well.

Probably the greatest advantage performance wise of using MPI is that the correct distribution of data to processors for optimizing cache performance, comes as a free bonus with the MPI implementation. We find this, for the programmer, hard to control in OpenMP.

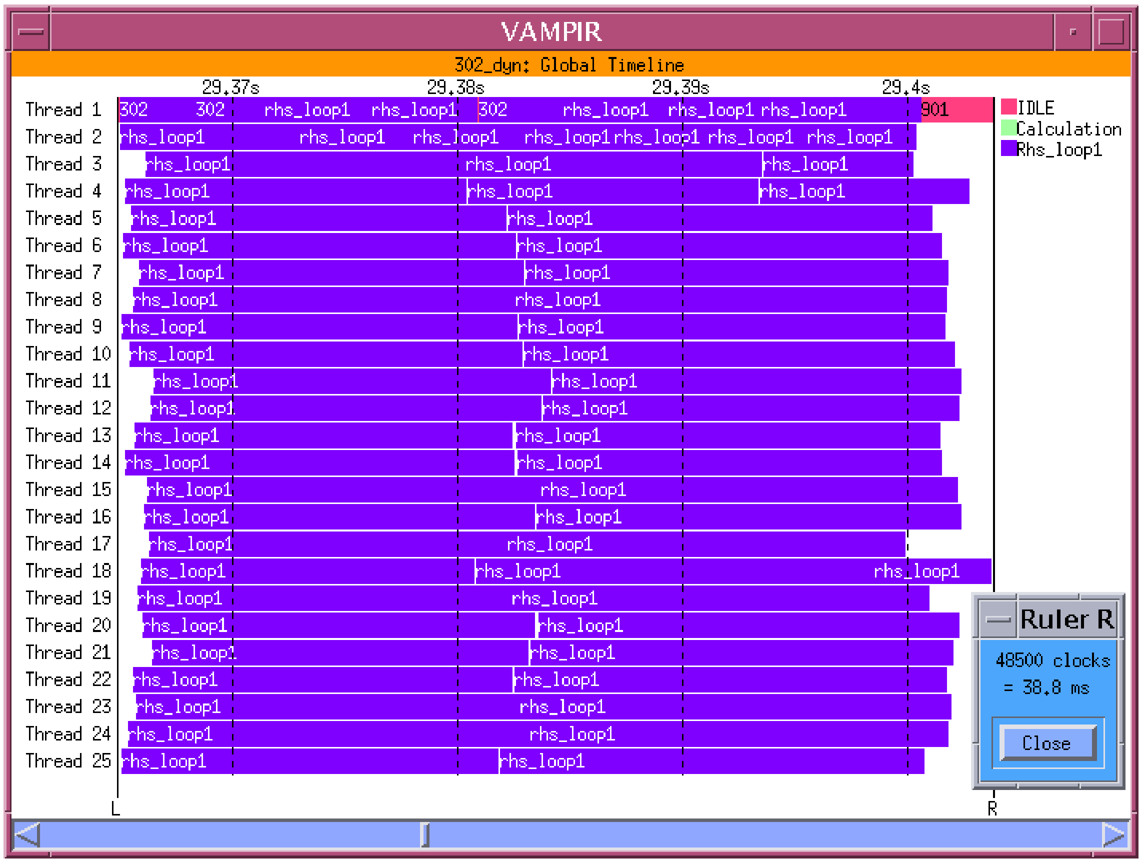

Figure 5: Visualization of the same loop as in Figure 5 when dynamic scheduling is used

5 Conclusion

OpenMP is a promising step towards a standard for SMP-programming. To be successful it is important that reliable and efficient solutions become available on most vendors platform soon. A similar OpenMP API for C, should also be made available soon.

SGI should work hard on implementing the entire OpenMP standard on their compiler. Integration with the automatic parallelizer should also be done. On a physically distributed memory system like the Origin 2000, the data distribution directives of Power Fortran have proved valuable. We would like to see these as a part of the OpenMP compiler.

Acknowledgment

We greatfully acknowledge the assistance of Prof. Wolfgang Nagel, Stephan Seidl and the rest of the staff at ZHR at TU-Dresden in this project. The use of their VAMPIR prototype for OpenMP has been of great help to us, and so has the free use of computer time on their Origin system been. The first author also wants to thank for their kind hospitality and financial support at her visit in Dresden.

References

[BHW+95] David Bailey, Tim Harris, Saphir William, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel bench-

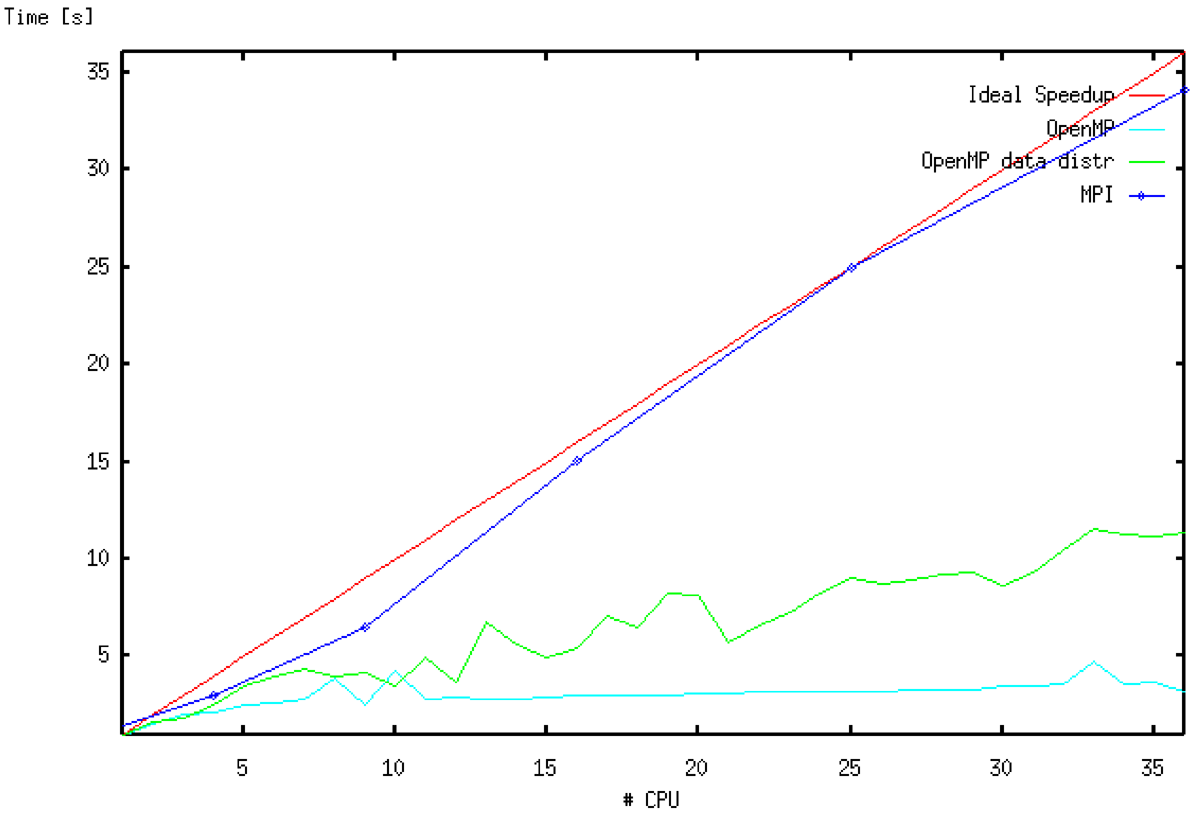

Figure 6: Speed-up for the NAS-BT benchmark class A, for the MPI-program and the two versions of OpenMP.

marks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, `http://science.nas.nasa.gov/Software/NPB/Specs/npb2_0/npb2_0.html`, December 1995.

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

[Gmb] Pallas GmbH. Vampir 2.0, users manual. Technical report, `http://www.pallas.de/`.

[Gra] Silicon Graphics. Speedshop user's guide. `http://www.hpcc.nectec.or.th/PCH/man/ssrun.html` and `http://www.hpcc.nectec.or.th/PCH/man/prof.html`.

[KLS+94] Chuck Koelbel, David Loveman, Rob Schreiber, Guy Steele, and Mary Zosel. *The High Performance FORTRAN Handbook.* The MIT Press, 1994.

[Le94] B Leasure and ed. Parallel processing model for high level programming languages. Technical report, Draft Proposed American National Standard for Information Processing Systems, 1994.

[NAW+96] Wolfgang E. Nagel, A Arnold, M. Weber, H. C. Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of mpi resouces. *Supercomputer*, 12(1), 1996.

[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.