

A comparison of OpenMP and MPI for the parallel CFD test case

Michael Resch, Björn Sander and Isabel Loebich
High Performance Computing Center Stuttgart (HLRS)
Allmandring 30, D-70550 Stuttgart
Germany
resch@hls.de

Abstract— This paper presents a comparison of OpenMP and MPI based on the parallel CFD test case defined by Prof. Akin Ecer from Indiana University Purdue University Indianapolis (IUPUI) [1]. The focus of our study is on performance of the benchmark comparing OpenMP and MPI. However, in the frame of the project we also wanted to find out about the complexity of the two different parallel programming models. Furthermore we decided to use a modified version of this test case for teaching in parallel programming courses. Experience with teaching MPI and OpenMP in such courses is reflected in this paper.

In the following we first will briefly describe the test case. The second chapter is about how we implemented the test case. The third chapter presents first results measured on an SGI Onyx 2. We compare results for MPI and OpenMP for a three-dimensional problem. To learn more about the behavior of OpenMP we additionally did some tests with a two-dimensional problem investigating on the effect of different scheduling algorithms for the PARALLEL DO construct.

Keywords— OpenMP, MPI, Performance, Parallel CFD Test Case

I. THE PARALLEL CFD TEST CASE

The Parallel CFD test case was defined in 1996 by Prof. Akin Ecer from Indiana University Purdue University Indianapolis [1] to study the development of software tools and computer systems. Since benchmarks like LINPACK are too academic for real applications, Prof. Ecer chose to define a "best least-common denominator" for CFD applications.

Problem to be solved:

- Solve the following PDE on a parallel computer:
 $\frac{df}{dt} = \Delta f$ (unsteady, heat conduction)
- Use an explicit scheme: forward-time, centered-space
- Solve the equation over a cube of unit dimensions
- Initial conditions: $f = 0$ everywhere inside the cube
- Boundary conditions: $f = x$ on all edges
- Calculate the steady-state solution

Programming requirements:

- Language: Fortran
- Message Passing Library: PVM shall be used (we decided to use MPI since it has replaced PVM as the standard)
- Load Balancing: The grid shall be as evenly divided as possible such that each processor has about the same amount of work to do. All tests shall be performed with no additional loading.

In order to investigate on the behavior of different parallel architectures the test case may be varied in different ways:

- Vary number of grid points in each direction to be able to simulate problems of different size.
- Vary memory requirements (store each grid point 1, 10 or 100 times).
- Solve each grid point 1, 10 or 100 times and thus vary the computational costs.
- Send messages 1, 10 or 100 times and thus vary the communication costs.

By modifying these parameters one may get an insight into the behavior of parallel architectures and learn about their computational costs and benefits as well as about their communication performance. First results with a C version of the code based on MPI gave interesting insights [2].

A. Modifications to the test

The test case as described was chosen with having the message passing paradigm in mind. Using other parallel programming models (like HPF or OpenMP) it is getting difficult to do the parameter variations as asked for. The grid size and the memory requirements can be varied easily for each programming model. But since neither HPF nor OpenMP allow the user to explicitly send messages, a variation in number of messages sent can not easily be achieved keeping all the other parameters constant. The same is true for the variation of computational work. An MPI program may simply increase the number of times that the calculation loop is executed before messages are exchanged. For HPF and OpenMP this can not be done easily. Communication and computation are interwoven to an extent that makes it difficult for the programmer to separate them in a reasonable way.

Given these restrictions we decided to skip that part of the test and focus on the comparison of MPI and OpenMP for this study.

II. IMPLEMENTATION

Our intention was not only to run one single implementation of the test case on different machines to learn about hardware, but also to implement the test case using different parallel programming models. So, we started with a

sequential version of the code written in Fortran. In the next step we parallelized that code using MPI [3]. Only then we went back to the sequential code and parallelized it using OpenMP [4], [5]. This way we were able to learn more about how easy or difficult it is to parallelize a code using the two different models.

A. Implemented variations of the test case

We implemented two variations of the test case. In preparation of our workshops about parallel programming, MPI and OpenMP we realized that users need to work with real applications to understand the meaning of a parallel programming model. A three-dimensional example seemed to be too complex to be used in a two-day course. However, the parallel CFD test case was simple enough to be understood by engineers. So, we decided to implement a two-dimensional version of it. First experience shows that both for MPI and for OpenMP this implementation can help engineers and students to get a better understanding of parallel programming while keeping the overhead for understanding of technical details and the complexity of the program small enough.

On the other hand - to learn more about MPI and OpenMP - we implemented the three-dimensional test case. This was done in a student project.

B. First lessons learned from the implementation

As a first result of our implementation and teaching efforts we found:

- Initially MPI programming is much more difficult to learn and understand than OpenMP. This seems to be the case because in MPI the user has to be aware of parallel processes. Once the user is familiar with the concept of parallel processes as expressed in the MPI communicator concept it seems to be quite natural to express the exchange of messages.
- With MPI it is more easy to naturally express the parallelism of the program. Once the user has understood the concept behind MPI it is easy to make use of the extensive functionality the standard provides. So, even complex functionality can easily be implemented.
- Initially OpenMP seems to be easier to understand when you first look at it. Just inserting directives allows to implement a parallel program without even having to care about basic concepts of OpenMP. But our experience shows that what the user has in mind very often is not what the compiler does. Typically this is the case if the user has not tried to understand the basic assumptions of OpenMP.
- Programming MPI the user will always know what the program is going to do. Working with OpenMP the user will have to check carefully whether his understanding of programming is the same as that of OpenMP.

These findings are the summary of initial testing and of some teaching of students and engineers from a research environment. But they were also found in the frame of this project.

We concede that our long lasting experience with MPI programming may have biased these results. It is most easy for us to explain concepts and handling of MPI to our customers. With OpenMP this is more difficult. However, most of the work in this project was done by a student who did not know anything about parallel programming before. So, we assume that our findings can be generalized at least to some extent.

Generally we can say: Usage of MPI requires a good understanding of the concept of independent processes and the explicit exchange of messages to write even a very simple program. Usage of OpenMP requires not a lot of understanding to write a parallel program. However, good results can only be achieved if those concepts are well understood.

III. RESULTS

In the following we summarize some results of our investigations. The focus is on two aspects:

- An investigation of the impact of different parallelization methods in OpenMP.
- A comparison of MPI and OpenMP performance for the parallel CFD test case.

The hardware and software tested was:

- SGI Onyx 2:
 - Number of processors: 14
 - Operating system: Irix64 6.4
 - Compiler: MIPSpro Compiler Version 7.2.1.2m, KAI compiler V3.6
 - MPI: SGI MPI optimized for IRIX 6.2 and IRIX 6.4
- HP V2250
 - Number of processors: 8
 - Operating system: HP-UX B.11.00 A 9000/800
 - Compiler: HP F90 v2.0 (no OpenMP), KAI V3.7

Although we did testing with the KAI compilers we only show results for the SGI native compiler here. Typically KAI performs similar to the SGI compiler. The reason we do not show all KAI results is that we had found some minor problems with that compiler which we reported to KAI. We assume that these are fixed by the day that paper is published. Where we have found similar qualitative behavior of native and KAI compilers we explicitly mention this.

A. Impact of different methods in OpenMP

The impact of different parallelization methods of OpenMP was tested using the two-dimensional parallel CFD test case. The core of the two-dimensional test case are two loops that are parallelized using PARALLEL DO

```
do it=1,itmax
  dphimax=0.
  do k=1,kmax-1
    do i=1,imax-1
      dphi=(phi(i+1,k)+phi(i-1,k)-2.*phi(i,k))*dy2i
      f      +(phi(i,k+1)+phi(i,k-1)-2.*phi(i,k))*dx2i
      dphi=dphi*dt
    dphimax=max(dphimax,dphi)
  enddo
enddo
```

```

    phin(i,k)=phi(i,k)+dphi
  enddo
enddo
c save values
do k=1,kmax-1
  do i=1,imax-1
    phi(i,k)=phin(i,k)
  enddo
enddo
if(dphimax.lt.eps) goto 10
enddo
10    continue

```

The dimensions *imax* and *kmax* were set to 80. This ensures to achieve a reasonable runtime for testing purposes using default code optimization which is in the range of 1 minute. Overhead for thread creation should therefore not be visible in the results. When compiling with `-O3` the time is reduced to about 10% which is too short for testing purposes. To do the convergence check in the first loop correctly there are two options:

- Use a critical section

```

do it=1,itmax
  dphimax=0.
  !$OMP PARALLEL PRIVATE(i,k,dphi),
  !$OMP& SHARED(phi,phin,dx2i,dy2i,dt,dphimax)
  !$OMP DO
    do k=1,kmax-1
      do i=1,imax-1
        dphi=(phi(i+1,k)+phi(i-1,k)-2.*phi(i,k))*dy2i
        f      +(phi(i,k+1)+phi(i,k-1)-2.*phi(i,k))*dx2i
        dphi=dphi*dt
      !$OMP CRITICAL
        dphimax=max(dphimax,dphi)
      !$OMP END CRITICAL
        phin(i,k)=phi(i,k)+dphi
      enddo
    enddo
  !$OMP END DO
  ...
  !$OMP END PARALLEL

```

- Use a reduction clause

```

do it=1,itmax
  dphimax=0.
  !$OMP PARALLEL PRIVATE(i,k,dphi),
  !$OMP& SHARED(phi,phin,dx2i,dy2i,dt,dphimax)
  !$OMP DO REDUCTION(max:dphimax)
    do k=1,kmax-1
      do i=1,imax-1
        dphi=(phi(i+1,k)+phi(i-1,k)-2.*phi(i,k))*dy2i
        f      +(phi(i,k+1)+phi(i,k-1)-2.*phi(i,k))*dx2i
        dphi=dphi*dt
        dphimax=max(dphimax,dphi)
        phin(i,k)=phi(i,k)+dphi
      enddo
    enddo
  !$OMP END DO
  ...
  !$OMP END PARALLEL

```

A comparison of the two methods quickly reveals that a critical section is much more time consuming on both the SGI and the HP than a reduction operation. This is true for the native SGI compiler and for the KAI compilers on both machines. Timings for the critical section are several times higher than those for the reduction operation. And while the reduction algorithm shows good speedup, the critical

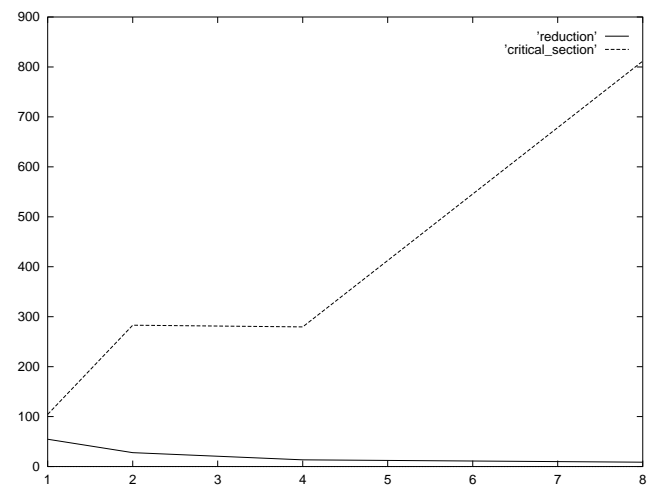


Fig. 1. Times for parallelization of the two-dimensional test case for reduction and critical section (SGI MIPSpro compiler version 7.2.1.2m).

section increases the time for increasing number of processors. It is obvious that a critical section is so expensive that it should only be used when there is no work around. At least with this implementation of OpenMP. First tests on the HP using the KAI compiler indicate that this is also the case for other platforms. A critical section should therefore at least not be used inside a loop.

In the following only the reduction algorithm is used for testing. To learn about the behavior of parallel loops we tried different versions of scheduling. OpenMP provides basically three different ways of scheduling the iterations of a loop:

- **STATIC**: Pieces of work are statically assigned to threads in the team in a round-robin fashion
- **DYNAMIC**: Iterations are broken into pieces of a specified chunk. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations.
- **GUIDED**: The chunk size is reduced in an exponentially decreasing manner.

The scheduling may be changed every time the program is started. To do that the scheduling mode has to be set to `RUNTIME` in the code. The decision about scheduling is then deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. We did that and investigated on the behavior of the program for the following different scheduling modes.

- `OMP_SCHEDULE` not set. We expected that some default value would be used.
- `OMP_SCHEDULE` set to `STATIC` with chunk size 4
- `OMP_SCHEDULE` set to `STATIC` with chunk size 20
- `OMP_SCHEDULE` set to `DYNAMIC` with chunk size 10

The results show the following:

- The default results are always the best. The compiler obviously is able to automatically find a good strategy for

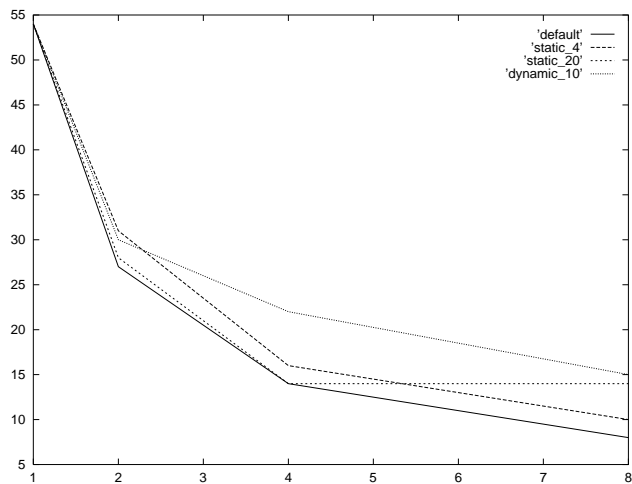


Fig. 2. Comparison of different scheduling strategies in OpenMP for a parallel loop (SGI MIPSpro Compiler Version 7.2.1.2m).

scheduling. This indicates that even a not very experienced user can benefit substantially from the automatic parallelization of OpenMP in this case.

- The dynamic scheduling shows rather bad results in all cases. Using a chunk size of 10 one would expect that the load can be perfectly balanced. However, the overhead for 8 processors is already 100% compared to the default strategy. This behavior may change for bigger loops and bigger chunk sizes. But it clearly indicates that the user has to pay a penalty for the more flexible scheduling.
- Static scheduling performs rather well. Using a chunk size of 4 is obviously not the best solution (we are computing a loop of size 80) but the results are close to the best results. For a bigger chunk size the results are better for small numbers of processors. Obviously the reduced overhead improves the performance. The bad result for 8 processors is due to the fact that with a chunk size of 20 and a loop size of 80 there is only work for 4 processors. And that is why the results for 4 processors and for 8 processors are the same.

Summarizing these results we can say that the user should go with the default scheduling strategy first. Further improvement could be reached by carefully studying the behavior of the program with respect to variations of chunk size. But only very experienced programmers will be able to set this size correctly in any case. These findings hold in principal also for the KAI compiler.

B. A comparison of MPI and OpenMP

The impact of different parallelization methods of MPI and OpenMP was tested using the three-dimensional parallel CFD test case.

First results in this case are not very promising. These results were achieved using a mesh of size 30x30x30. Given that we only have 8 processors that should be big enough to show some reasonable results. However, our first tests indicate that the test case still is too small to give us all the information that we need. Using the optimization of

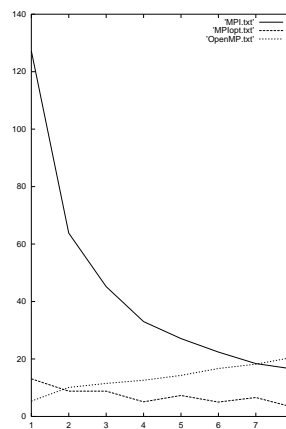


Fig. 3. Comparison of timings for MPI, optimized MPI (-O3) and OpenMP for the three-dimensional test case.

the SGI compiler already the sequential code is 7 times faster than the version we did not optimize. Even though the non-optimized code shows good speedup a first result is only that one should carefully optimize the sequential code before going parallel.

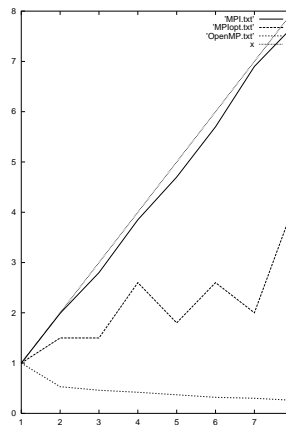


Fig. 4. Comparison of speedup for MPI, optimized MPI (-O3) and OpenMP for the three-dimensional test case.

Having a look at the speedup we see the following:

- The speedup is excellent for the non-optimized MPI code.
- The speedup of the optimized code indicates that the example is probably too small for our tests.
- OpenMP faces a slow down.

Next steps are therefore to run bigger examples on bigger machines and to investigate carefully on the behavior of the OpenMP code in the three-dimensional case.

IV. SUMMARY

We did a first study of the behavior of OpenMP on two machines and combined this with a study on different parallel programming models (OpenMP and MPI). The results that we found are not yet satisfactory. Some initial information about the behavior of different scheduling strategies could be found. It seems that using default values is good

for both the native SGI compiler and the KAI compiler on an SGI. Results for the KAI compiler on the HP are similar but so far we have not been able to verify them on a dedicated machine.

Another finding is that critical sections are too expensive to be just used like another feature. To some extent they compare well to the `MPLBarrier` that is a nice feature but should be avoided wherever possible.

With respect to scheduling strategies we have found that for our rather simple loops the default settings of all compilers seem to be very reasonable. This may be due to the fact that parallelization of loops is one of the topics that has been investigated very intensively in the past. Here the results of computer science show a very positive impact for the programmer.

Our findings for the three-dimensional problem are a little bit disappointing at the moment. However, we admit that the reason for this is mainly our poor understanding of OpenMP. In the end we have not even yet found out why a sequential run of an OpenMP program is much slower than the fastest sequential version. Even though OpenMP adds some overhead we were surprised by times that were significantly higher than what we expected.

Finally we would like to express a positive feeling about OpenMP with respect to future usage in our center. Although it may take some time to understand what is hidden away from the user - and unfortunately influences the performance of the code - we find that it's much easier to "reuse" user experience than it is for MPI. Furthermore, it's much easier to come up with a first parallel program in OpenMP. Therefore, we expect OpenMP to become more important for our center and for the user community.

REFERENCES

- [1] <http://www.hlr.de/people/resch/PROJECTS/PARACFD.html>
- [2] Dirk Sihling, Michael Resch, Alfred Geiger *Performance-Tests paralleler Computersysteme am Beispiel eines standardisierten Anwendungs-Benchmarks*, RUS-36 internal report, Januar 1997, ISSN 0941-4665.
- [3] Message Passing Interface Forum *MPI A message-passing interface standard*. International Journal for Supercomputing Applications, 8(3/4), 1994.
- [4] *OpenMP Fortran Application Program Interface Ver 1.0*, October 1997.
- [5] *OpenMP C and C++ Application Program Interface Version 1.0*, October 1998.