From a Vector Computer to an SMP-Cluster Hybrid Parallelization of the CFD Code PANTA

Dieter an Mey

Computing Center
Aachen University of Technology
anmey@rz.rwth-aachen.de
http://www.rz.rwth-aachen.de

Stephan Schmidt

Institute for Jet Propulsion and Turbomachinery
Aachen University of Technology
schmidt@ist.rwth-aachen.de
http://www.ist.rwth-aachen.de

Abstract

Transforming a well vectorized Fortran77 CFD code into a more flexible Fortran90 program using a hybrid parallelization model still is an adventure. Here we report on our experiences with PANTA, a 3D Navier-Stokes solver that is extensively used in the modeling of turbomachinery.
During this ongoing process various software tools are employed: Hewlett Packard's, Silicon Garphics', Portland Group's, and SUN's auto-parallelizing Fortran90 compilers, various serial runtime performance analyzers, Simulog's Foresys Fortran Engineering System in combination with AWK scripts, Etnus' TotalView parallel debugger, Kuck & Associates' KAP/Pro Toolset and Visual KAP for OpenMP preprocessor and Pallas' Vampir performance analysis tools for MPI programs.
A few suggestions for an improved integration of automatic and explicit parallelization are made.

1 Introduction

PANTA is a well vectorized 3D Navier-Stokes solver that is extensively used in the modeling of turbomachinery [2,3]. Fig. 1.1 shows a 2D cut through the two blade rows of a simulated turbine. As the underlying geometry provides a natural decomposition of the computational domain it is well suited for parallelization.

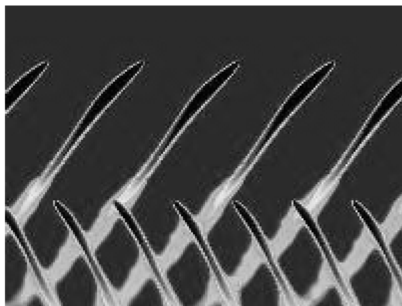

Fig. 1.1: Entropy distribution at midspan ErcC6 turbine

The program structure depicted in Fig. 1.2 contains a couple of nested loops which might be exploited for parallelization. The numbers in parenthesis indicate the typical iteration count of the corresponding loops. In order to use a parallel machine with many processors, multiple levels of parallelization have to be combined. The current version exploits only two levels (see chapters 3 and 4)

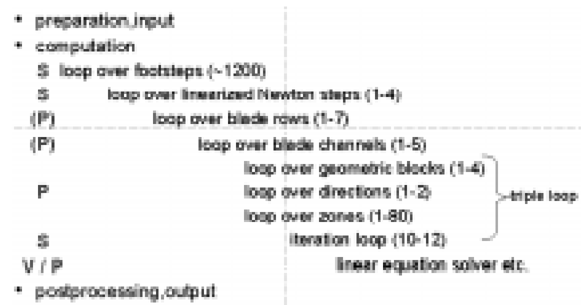

Fig. 1.2: PANTA code structure

In the context of preparing an existing Fortran program for parallelization, it is in our experience advantageous to migrate to Fortran90 in order to be able to make use of its dynamic memory management features. The program transformations have been carried out using the Foresys Fortran Engineering System from Simulog [8] and additional AWK scripts (see chapter 2).
The most time-consuming routines had been well vectorized before, but they also have a high potential to be parallelized at loop level on a shared-memory system. In Chapter 3 the parallelization of a typical loop nest with OpenMP is demonstrated. As the program has many time-consuming routines, it is not possible to parallelize all of them by hand, auto-parallelization is inevitable.
Combining automatic and explicit parallelization with OpenMP is also feasible, but an integration of explicit parallelization with OpenMP directives and auto-parallelization with the support of standardised directives would be very helpful.
The coarse-grained parallelization (see chapter 4) may as well be approached with OpenMP. Using the verification tool Assure by Kuck & Ass. (KAI) [5] pitfalls may easily be detected. But because nested parallelism with OpenMP is not yet supported by the current production (pre-)compilers, finally MPI has

to be used if OpenMP is employed at a lower level. Vampir by Pallas [6] is well suited to display the timing behavior of an MPI program.
If a thread-safe MPI library is available, both parallelization efforts, OpenMP and MPI, can easily be combined, because here the two levels of parallelism do not interfere.
The makefiles are getting ugly because the parameters of compilers and linkers are quite machine-dependant. A higher degree of compatibility with respect to parameters and directives would be highly welcome.
Fortunately it turns out that KAI's GuideView, only intended to show the runtime behavior of OpenMP programs, can be well used for hybrid programs (see chapter 5).

2 Preparing the serial code

The Foresys Fortran Engineering System from Simulog [8] allows automatic restructuring of Fortran codes and conversion from Fortran77 to Fortran90 (see Fig. 2.1). It also checks the syntactic correctness of the program and generates a code which is well structured, and thus allows easy semi-automatic postprocessing.

Fig. 2.1: Conversion from Fortran77 to Fortran90 with Foresys

The program transformations have been carried out in several steps (see Fig. 2.2).

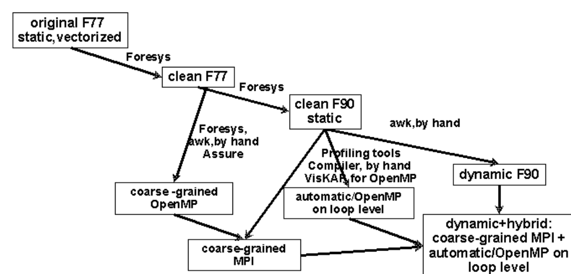

Fig. 2.2: PANTA code versions

The first version generated with Foresys (called "clean F77") contained additional comments indicating the usage of all common block members. Generating the second version with Foresys (called "clean F90") all common blocks have been extracted to separate module files.
Additional AWK scripts have been developed in order to replace the fixed dimensions of all 751 global arrays by ALLOCATABLE declarations, to generate corresponding ALLOCATE Fortran statements, and to replace all local arrays by automatic arrays (see Fig. 2.3).
The ALLOCATE statements are now executed after reading the necessary parameters from an input file. Thus recompilation for each test case is no longer necessary.

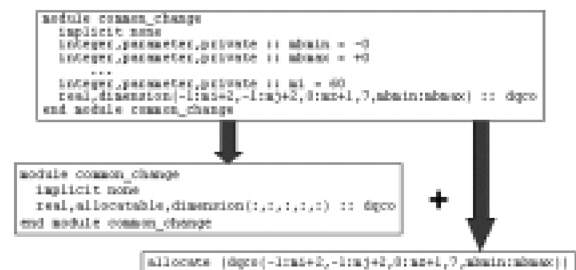

Fig. 2.3: Introducing dynamic memory management with AWK

Heavy use of allocatable arrays reveals a performance bug of the current HP Fortran90 compiler: the program is slowed down by a factor of three, whereas on other machines only a performance loss of up to 20 percent has been measured.

3 Replacing vectorization by fine-grained shared-memory parallelization

A runtime analysis of the serial code shows that one third of the CPU time is consumed in a single loop nest of one routine (see Fig. 3.1 and 3.2). But according to Amdahl's law, it is not sufficient only to concentrate on this routine. As the analysis reveals, quite a few further routines have to be tuned, too, in order to achieve a reasonable parallel speed up. Like vectorization, loop level parallelization is also a step-by-step process.
A closer inspection of the critical code segments and of the vectorization runtime statistics show that there are many loop nests, typically featuring at least one long loop, as shown in Fig. 3.2.
The inner loop has thousands of iterations, so it is chosen for vectorization. The auto-parallelizing compilers would prefer to parallelize the outer loop in order to maximise the work load for the single threads between synchronisation points. But the data dependency analysis as well as a close look at the code reveal that with respect to this loop (over `n`) all elements of the array RHS are used recursively. For

this reason the compilers choose the middle loop for parallelization. But this loop has only seven iterations, so it does not scale well and also induces a load imbalance unless exactly seven threads are employed.

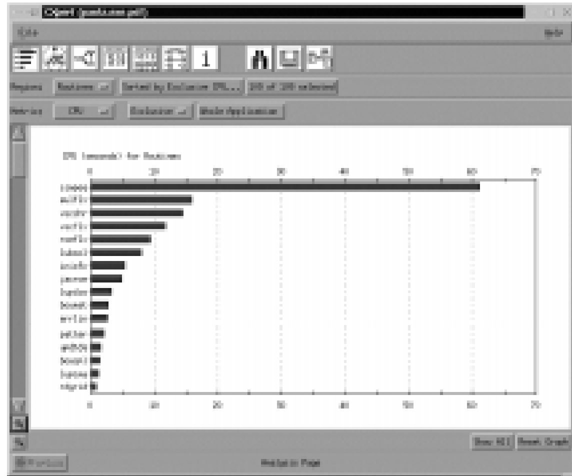

Fig. 3.1: Runtime analysis with CXperf (HP)

```
!  when parallelizing this loop, recursion for RHS(l,m)
do n = 1,7
   !  this loop is chosen by auto-parallelizing compilers,
   !  it only has 7 iterations, load imbalance
   do m = 1,7
      !  many iterations, used for vectorization
      do l = LSS(itsub),LEE(itsub)
         i = IG(l)
         j = JG(l)
         k = KG(l)
         lijk = L2IJK(l)
         RHS(l,m) = RHS(l,m)- &
            FJAC(lijk,lm00,m,n) &
             *DQCO(i-1,j,k,n,NB)*FM00(l) - &
            FJAC(lijk,lp00,m,n) &
             *DQCO(i+1,j,k,n,NB)*FP00(l) - &
            FJAC(lijk,l0m0,m,n) &
             *DQCO(i,j-1,k,n,NB)*F0M0(l) - &
            FJAC(lijk,l0p0,m,n) &
             *DQCO(i,j+1,k,n,NB)*F0P0(l) - &
            FJAC(lijk,l00m,m,n) &
             *DQCO(i,j,k-1,n,NB)*F00M(l) - &
            FJAC(lijk,l00p,m,n) &
             *DQCO(i,j,k+1,n,NB)*F00P(l)
      end do
   end do
end do
```

Fig. 3.2: Code fragment with the most time consuming loop nest

A better approach is taken by KAI's preprocessor Visual KAP for OpenMP (see Fig. 3.3). The long inner loop is splitted and stripmined with block size 64. This method gives more flexiblity with respect to the number of threads used while preserving a reasonable granularity.
With these experiences gathered from using various compilers and tools, another solution was implemented manually (see Fig. 3.4). With OpenMP it is possible to put all three nested loops inside the parallel region, to execute the outer loops redundantly and only share the work in the long inner loop. Because no synchronisation is necessary at the end of the inner loop, maximal flexibility and the optimum granularity and balance of the workload is maintained.

```
!$OMP PARALLEL SHARED(ITSUB,LSS,II15,RHS,&
!$OMP& II14,L2IJK,FJAC,NB,KG,JG,IG,DQCO,&
!$OMP& FM00,II13,FP00,II12,F0M0,II11,F0P0,&
!$OMP& II10,F00M,II9,F00P,LM00,LP00,L0M0,&
!$OMP& L0P0,L00M,L00P,LEE,LFJ) &
!$OMP& PRIVATE(II2,II1,N1,M,L,II16,II4,RR1,&
!$OMP& II3,I,J,K,LIJK,II17,II18,II20,II21,&
!$OMP& II22,II23,II24,II25,N,NM,II19,II32,&
!$OMP& I26,II27,II28,II29,II30,II31,II6,II5)
!$OMP DO
DO II1=LSS(ITSUB),II15,64
  II2 = MIN (II15,II1+63)
  DO N1=1,7
    DO M=1,7
      DO L=II1,II2,1
        RHS(L,M) = RHS(L,M) - &
        & FJAC(L2IJK(L),II14,M,N1) &
        & * DQCO(IG(L)-1,JG(L),KG(L),N1,NB)&
        & * FM00(L) - &
        & FJAC(L2IJK(L),II13,M,N1) &
        & * DQCO(IG(L)+1,JG(L),KG(L),N1,NB)&
        & * FP00(L) - &
        & FJAC(L2IJK(L),II12,M,N1) &
        & * DQCO(IG(L),JG(L)-1,KG(L),N1,NB)&
        & * F0M0(L) - &
        & FJAC(L2IJK(L),II11,M,N1) &
        &* DQCO(IG(L),JG(L)+1,KG(L),N1,NB) &
        & * F0P0(L) - &
        & FJAC(L2IJK(L),II10,M,N1) &
        & * DQCO(IG(L),JG(L),KG(L)-1,N1,NB)&
        & * F00M(L) - &
        & FJAC(L2IJK(L),II9,M,N1) &
        & * DQCO(IG(L),JG(L),KG(L)+1,N1,NB)&
        & * F00P(L)
      END DO
    END DO
  END DO
END DO
!$OMP END DO NOWAIT
!$OMP BARRIER
```

Fig. 3.3: Code fragment auto-parallelized with the Visual KAP for OpenMP preprocessor

Fig. 3.5 shows that the latter solution is slightly better than the one obtained with Visual KAP for OpenMP and much better than the compiler-generated version.
Unfortunately it is not sufficient only to care for the most time consuming loop nests, because too many leftover routines dominate the runtime behaviour of the total program. Therefore the most time consuming of the remaining routines have been parallelized automatically with proprietary compilers and alternatively with the preprocessor Visual KAP for OpenMP from KAI, which augments existing Fortran code with OpenMP directives. It is available for NT and Tru64 Unix

only, but the modified code can be ported to other platforms.

```
!$omp parallel private(n,m,l,i,j,k,lijk)
do n = 1,7
  do m = 1,7
    !$omp do
    do l = LSS(itsub),LEE(itsub)
      i = IG(l)
      j = JG(l)
      k = KG(l)
      lijk = L2IJK(l)
      RHS(l,m) = RHS(l,m)- &
        FJAC(lijk,lm00,m,n) &
        & * DQCO(i-1,j,k,n,NB)*FM00(l) - &
        & FJAC(lijk,lp00,m,n) &
        & * DQCO(i+1,j,k,n,NB)*FP00(l) - &
        & FJAC(lijk,l0m0,m,n) &
        & * DQCO(i,j-1,k,n,NB)*F0M0(l) - &
        & FJAC(lijk,l0p0,m,n) &
        & * DQCO(i,j+1,k,n,NB)*F0P0(l) - &
        & FJAC(lijk,l00m,m,n) &
        & * DQCO(i,j,k-1,n,NB)*F00M(l) - &
        & FJAC(lijk,l00p,m,n) &
        & * DQCO(i,j,k+1,n,NB)*F00P(l)
    end do
    ! no synchronisation needed
    !$omp do nowait
  end do
end do
!omp end parallel
```

Fig. 3.4: Manual parallelization of the most time consuming loop nest using OpenMP directives

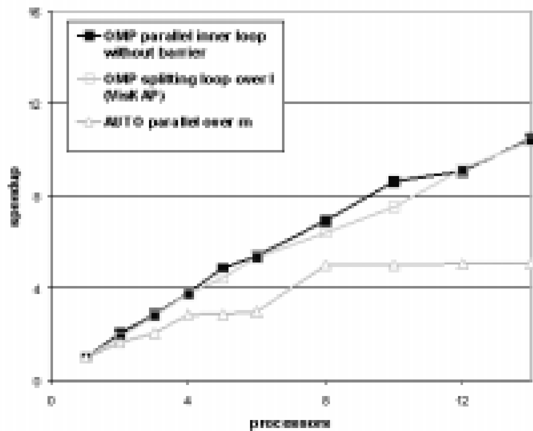

Fig. 3.5: Speedup of three different parallel versions of the most time consuming loop nest on a 16-processor HP V-class

Fig. 3.6 compares the timings for some routines for

- the original vector version on the Fujitsu VPP300,
- the serial version on a SUN Enterprise 450 (296 MHz, 4 CPUs, SUN WorkShop 6 Fortran90 compiler)
- the auto-parallelized version (SUN E450)
- the explicitly parallelized version with OpenMP directives (SUN E450)
- the explicitly parallelized version with OpenMP directives using the guide preprocessor from KAI (SUN E450)
- the auto-parallelized version by Visual KAP for OpenMP ported to the SUN E450
- the best version for each routine

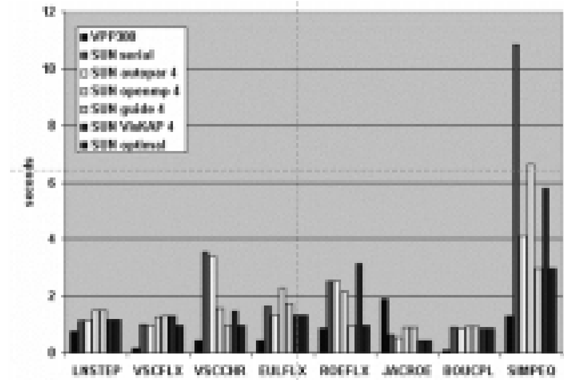

Fig. 3.6: Different versions of some of the most time consuming routines on a SUN E450 with 4 CPUs versus the vector version on the Fujitsu VPP300

The measurements show that sometimes the version parallelized by the compiler is best, sometimes the hand-tuned version and sometimes the version preprocessed by Visual KAP. Therefore a combined version delivers the best results, although these effects are not really related to the whole subroutines but to the loop nests. But it is troublesome to analyse and control a mix of explicit and automatic parallelization within one subroutine. In chapter 6 a few suggestions for an extension of the OpenMP definition are made.

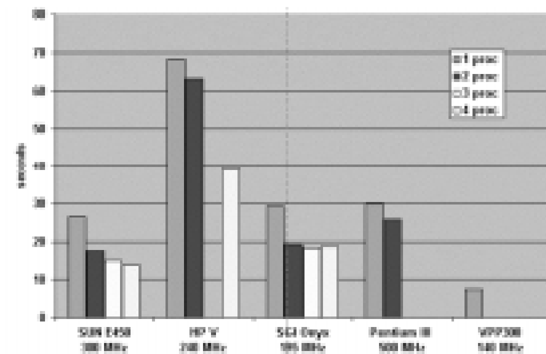

Fig. 3.7: Timings of tests runs combining auto-parallelization and manual parallelization

Fig. 3.7 shows the timings of test runs with a combined version on different machines.
Although the best parallel version of the most time consuming loop scales well (see Fig. 3.5) no more

than about four processors should be used on this parallelization level.
The speedup of the test case is not really satisfying, but production runs will certainly show a better performance.
The comparison of these measurements with the output of a performance analysis on the Fujitsu vector machine (see Fig. 3.8) shows that for some routines (SIMPEQ, VSCCHR, ROEFLX), where the difference between the vector version on the VPP300 and the serial version on the SUN E450 is particularly large, also the parallelization effect is high, whereas other routines which also run with a high vectorization rate are not yet parallelized sufficiently. These codes contain very complicated loops, which can be vectorized but not (yet) parallelized automatically. Also the explicit parallelization would cost enormous efforts with all the dangers of introducing bugs.

Count	Percent	VL	V_Hit(%)	Name
21420	38.6	444	99.6	SIMPEQ_
9245	16.6	51	97.4	EULFLX_
4943	8.9	60	83.1	ROEFLX_
3781	6.8	20	53.5	JACROE_
3511	6.3	60	92.0	VSCCHR_
2718	4.9	267	76.7	PATTER_
2287	4.1	60	77.9	VSCFLX_
2082	3.7	60	0.5	TSTEP_
671	1.2	640	93.9	ERRLIN_
430	0.8	633	98.4	UPDTDQ_
413	0.7	32	45.2	BOUVAR_
388	0.7	70	86.2	YPSPLU_
367	0.7	638	81.2	LUPIBS_
353	0.6	67	29.2	BOUCPL_
55540		277		TOTAL

Fig. 3.8: Sampling analysis of a test run on the Fujitsu vector machine VPP300 (extract)

4 Coarse-grained parallelization

Introducing coarse-grained parallelism in a large code can be a tedious job. Again Foresys may help a lot by introducing comments with information on the usage of the common block data. With AWK-scripts this information has been extracted to generate a global common block cross-reference list, which is very helpful for specifying the communication structure of the coarse-grained parallelization. It is also very comfortable to let Foresys write local variables using small letters and global variables using capital letters.

In a first approach we used OpenMP directives. At this point we observed that the usage of Fortran90 modules and their privatization with the `threadprivate` OpenMP directive is not part of the first version of the OpenMP Fortran API, but fortunately it is already supported by KAI's Guide preprocessor.
Exploiting the global common block cross-reference list we generated the necessary `threadprivate` OpenMP directives.
The original code segment chosen for coarse-grained parallelization contains three nested loops (see Fig. 4.1).

```
n12d = MOD((NFOOT+NT+NL)/2+1,2)
do ib = 1,ICLEAR+1
  do n12 = 1+n12d,2-n12d,1-2*n12d
    do nzo = 1,NZONE(n12,LSF,ib,NR)
      is = ISZ(nzo,n12,LSF,ib,NR)
      ie = IEZ(nzo,n12,LSF,ib,NR)
      js = JSZ(nzo,n12,LSF,ib,NR)
      je = JEZ(nzo,n12,LSF,ib,NR)
      ks = KSZ(nzo,n12,LSF,ib,NR)
      ke = KEZ(nzo,n12,LSF,ib,NR)
      call LNZONE(is,ie,js,je,ks,ke,ib)
      call POSTZO
    end do
  end do
end do
```

Fig. 4.1: Converting a triple loop to a single loop for parallelization - original code

```
integer,dimension(maxtsk)::A_ib,A_n12,A_nzo
  n12d = MOD((NFOOT+NT+NL)/2+1,2)
  ic = 0
  do ib = 1,ICLEAR+1
    do n12 = 1+n12d,2-n12d,1-2*n12d
      do nzo = 1,NZONE(n12,LSF,ib,NR)
        ic = ic + 1
        A_ib(ic) = ib
        A_n12(ic) = n12
        A_nzo(ic) = nzo
      end do
    end do
  end do

  !$omp parallel private(ib,n12,nzo,is, &
  !$omp& ie,js,je,ks,ke)
  !$omp do
  do iloop=1, ic
        ib=A_ib(iloop)
        n12=A_n12(iloop)
        nzo=A_nzo(iloop)
        is = ISZ(nzo,n12,LSF,ib,NR)
        is = ISZ(nzo,n12,LSF,ib,NR)
        ie = IEZ(nzo,n12,LSF,ib,NR)
        js = JSZ(nzo,n12,LSF,ib,NR)
        je = JEZ(nzo,n12,LSF,ib,NR)
        ks = KSZ(nzo,n12,LSF,ib,NR)
        ke = KEZ(nzo,n12,LSF,ib,NR)
        call LNZONE(is,ie,js,je,ks,ke,ib)
       !$omp critical
        call POSTZO
       !$omp end critical
  end do
  !$omp end do nowait
  !$omp end parallel
```

Fig. 4.2: Converting a triple loop to a single loop for parallelization - modified code

After recording the loop indices in three arrays, they can be replayed in a single loop thus collapsing the

loop nest and facilitating the parallelization (see Fig. 4.2).
A verification run of this OpenMP code with KAI's unique verification tool Assure gives valuable information about possible data access conflicts (see Fig. 4.3).

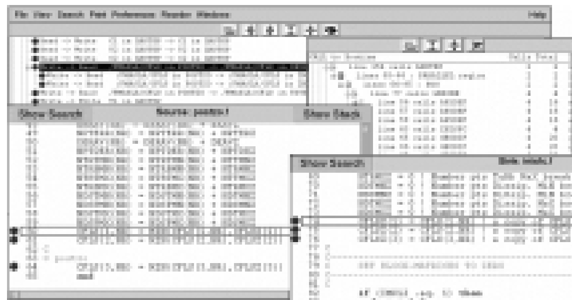

Fig. 4.3: Screen shot of the verification tool Assure detecting a possible data access conflict

Because we want to combine shared-memory parallelism on a lower level with coarse-grained parallelism, and as nested parallelism with OpenMP is not yet supported by the current production compilers, finally MPI has to be used. Nevertheless the knowledge gained with the Assure analysis may as well be used for the MPI parallelization. Also the global common block cross-reference list can be exploited again to generate MPI library calls.
A state-of-the-art tool for displaying the timing behavior of an MPI program is Vampir, available from Pallas [6]. After linking the additional VampirTrace library to the program a tracefile is written during program execution. This tracefile can then be analyzed very thoroughly with Vampir. Fig 4.4 shows the timeline display of a 4-processor run. The bright (originally green) parts of the four thick bars represent the calculation phases of each of the processors and the dark (originally red) parts show the communication phases.

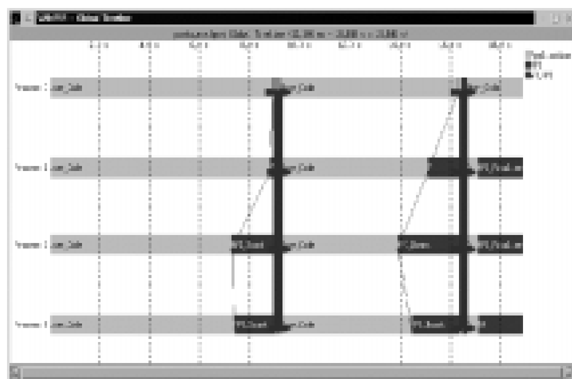

Fig. 4.4: Screen shot of the Vampir performance analysis tool

It can be seen that the load is not perfectly balanced as processors 2 and 3 are waiting for the first broadcast operation to terminate after each of the two iteration steps. During the communication phases many messages are exchanged, depicted by corresponding lines between the bars. After zooming in, Vampir allows to look at single events and also to display precise information about every message sent.

5 Combining MPI and OpenMP

If both parallelization efforts, OpenMP and auto-parallelization on a low level and MPI on a high level, are combined the makefiles are getting more complicated. Fig. 5.1 shows some simple (!) commands to compile, link and start a hybrid application and also to display the runtime profile with KAI's GuideView on an HP system. Take care to link the thread safe version of HP's MPI library!

```
# Compile
guidef90 -I/opt/mpi/include \
   +DA2.0 +DS2.0 +O3 +Odataprefetch \
   +Oparallel +Oautopar +Onodynsel *.f90
# Link
guidef90 -WGstats +DA2.0 -o panta.exe \
   +O3 +Oparallel *.o \
   -L/opt/mpi/lib/pa1.1 -lmtmpi
# Go
   # remove old guide statistic files
   rm guide.*.gvs
   # gang scheduling
   export MP_GANG=ON
   # guide statistic filenames with pid
   export KMP_STATSFILE=guide.%i
   # if combining autopar and OpenMP
   export KMP_BLOCKTIME=10
   # 4 OpenMP threads
   export OMP_NUM_THREADS=4
   # 4 fold autoparallel
   export MP_NUMBER_OF_THREADS=4
   # 2 MPI tasks
   mpirun -np 2 panta.exe
# Visualize the OpenMP performance
   # remove stats file of MPI master
   rm $(ls guide.*.gvs | head -1 ))
   guideview guide_stats.*.gvs
```

Fig. 5.1: Compile, link, run and visualise a hybrid program on an HP system using KAP Pro/Toolset

Currently there is no specific performance analysis tool for hybrid programs available. But it turns out that KAI's GuideView can also be used for the hybrid programming model. GuideView is able to display the statistics files of multiple OpenMP program runs side by side, in order to compare the parallelization success. Here this feature can be used to display the statistics files written be each of the MPI tasks together. On an HP system it is convenient to remove the statistics file of the MPI

master process, which does not contribute to the computation (see Fig 5.1).

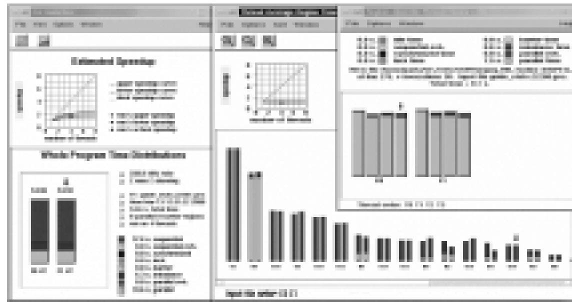

Fig. 5.2: Screen shot of GuideView displaying a hybrid program run with 2 MPI tasks and 4 OpenMP threads

Fig. 5.2 shows a screen shot of GuideView displaying a hybrid program run with 2 MPI tasks and 4 OpenMP threads each. The left window shows one bar for each MPI task. Only a small lower part of the two bars has a bright (originally green) color depicting a parallel computation phase, because only a short program run has been measured with all the pre- and postprocessing phases and only two iterations. The window in the middle shows all the parallel and serial regions of the two MPI tasks always side by side. Only one pair has a bright color, because during this phase of the program development only one routine (see chapter 3) has been thoroughly parallelized using OpenMP directives. The right window shows that all 4 threads of the two MPI tasks are quite well balanced during this parallel region. This preliminary result indicates that the program may be well parallelized, if the other time consuming routines are parallelized with a similar success.
The hybrid approach also is attractive if an MPI application with only a limited number of MPI tasks has to be accelerated on a relatively cheap hardware like a PC cluster. Fig. 5.3 shows the speedup of a PANTA test run on a Siemens hpcLine PC-cluster with double-processor nodes connected with a fast SCI network and the thread-safe MPI library ScaMPI from the Norwegian company Scali [9]. The Fortran90 compiler from Portland Group Inc. [7] is used for autoparallelization. The compiler also supports OpenMP directives. Running two MPI tasks on two double processor nodes with two OpenMP threads each, a speedup of about 2.8 has been measured.

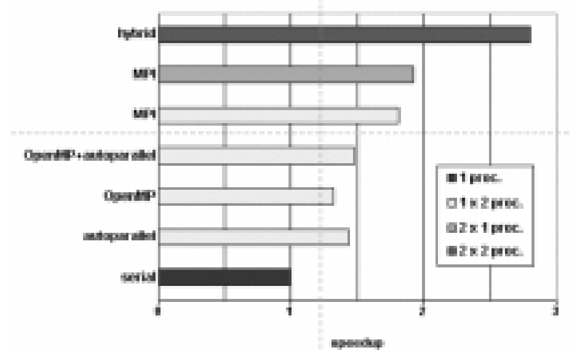

Fig. 5.3: Speedup of the hybrid version of PANTA on up to four processors of a Siemens hpcLine PC-cluster

6 Conclusion

It is our experience, that such a combination of the various tools for tuning and parallelization of Fortran90 codes frequently exposes problems and deficiencies. Nevertheless, the first results obtained with the code modifications so far are quite promising. The conversion of a static vector code to a dynamic hybrid parallel code seems to be feasible with reasonable manpower and with the help of suitable software tools.
But some additional features of OpenMP in order to integrate explicit and automatic parallelization would have made life much easier (see Fig. 6.1, 6.2, and 6.3).
For long loops a good compiler should be able to perform the parallelization analysis and the variable privatization more reliably than a programmer. (In order to successfully parallelize PANTA's routine VSCCHR the maximum number of continuation lines of an OpenMP directive has become critical because of the large number of variables which had to be privatized.)

```
!$omp serial
! do not parallelize
...
!$omp end serial
```

Fig. 6.1: Proposal: do not auto-parallelize this region

```
!$omp parallel auto
! try to autoparallelize
...
!$omp end parallel auto
```

Fig. 6.2: Proposal: try to auto-parallelize this region

```
!$omp parallel
DO m ...
  !$omp do
..DO n ...
    DO k ...
      ...
    END DO
  END DO
  !$omp end do nowait
END DO
...
!$omp auto [prefer_loop(n)]
! reuse threads, try to parallelize second loop
DO m ...
  DO n ...
    DO k ...
      ...
    END DO
  END DO
END DO
!$omp end auto
!$omp end parallel
```

Fig. 6.3: Proposal: try to auto-parallelize [a certain] loop and reuse the OpenMP threads

Having the long lists of directives for vector-compilers in mind (unfortunately they have never been standardized) it may be wise to think about the standardization of even more clauses to such an OpenMP `auto` directive. With such clauses the programmer would have the possibility to provide additional information to an auto-parallelizing compiler (see Fig. 6.4).

```
!$omp parallel
!$omp auto disjoint(a)
! try to autoparallelize
DO i = 1, n
  a(index(i)) = a(index(i)) + b(i)
END DO
!$omp end auto
!$omp end parallel
```

Fig. 6.4: Proposal: try to auto-parallelize this region, on the assumption that different elements of the array `a` are accessed

Porting a code from one machine to another is still an adventure, if the auto-parallelization features of the proprietary compilers are used. On the other hand, it is very attractive to autoparallelize an existing Fortran program using a preprocessor, like KAI's Visual KAP for OpenMP, augmenting the original code with OpenMP directives. If the generated code remains readable it can be used for further development. Unfortunately Visual KAP for OpenMP is only available on NT and Tru64 Unix.
With the rapidly growing number of large SMP systems there will be a need for the support of nested parallelism by the compilers including auto-parallelization and parallel libraries (e.g. BLAS). This would have been an interesting alternative for the PANTA code.
Last but not least compatible options for compilation and linking hybrid programs would be very nice:

```
f90 -mpi -openmp -autopar ...
```

References

1. Computing Center, Aachen University of Technology (RWTH) http://www.rz.rwth-aachen.de
2. Institute for Jet Propulsion and Turbomachinery, Aachen University of Technology (RWTH) http://www.ist.rwth-aachen.de [/en/forschung/instationaere_li.html]
3. Volmar, T., Brouillet, B., Gallus, H.E., Benetschik, H.: Time Accurate 3D Navier-Stokes Analysis of a 1½ Stage Axial Flow Turbine, AIAA 98-3247, 1998.
4. Etnus http://www.etnus.com
5. Kuck & Associates, Inc. http://www.kai.com
6. Pallas http://www.pallas.com
7. Portland Group, Inc. http://www.pgroup.com
8. Simulog http://www.simulog.fr
9. Scali http://www.scali.com