Automatic MPI Counter Profiling

Rolf Rabenseifner

Rechenzentrum Universität Stuttgart, Germany

Abstract— **This paper presents an automatic counter instrumentation and profiling module added to the MPI library on Cray T3E and SGI Origin2000 systems. A detailed summary of the hardware performance counters and the MPI calls of any MPI production program is gathered during execution and written in MPI_Finalize on a special syslog file. The user can get the same information in a different file. Statistical summaries are computed weekly and monthly and the user specific part is sent by mail to each user. The paper discusses scalability aspects of the new interface: How to obtain the right amount of performance data to the right person in time, and how to draw conclusions for the further optimization process, e.g. with the trace-based profiling tool Vampir. The paper describes two different software designs that allow the integration of the profiling layer into a Unix MPI library and into a dynamic shared object MPI library without consuming the user's PMPI profiling interface. Experiences with this library on the Cray T3E systems at HLRS Stuttgart and TU Dresden and a summary of 6 month are presented in this paper. It is the first time that all MPI applications on such a large system where automatically instrumented and profiled for such a period. The statistics give new insight in how efficiently the MPP system is really used by the MPI applications. Moreover, it gives hints which application and which MPI routine should be optimized. After integrating the hardware performance counters into the MPI counter profiling, first results with these counters are presented. The software is portable to other systems.**

Keywords— **MPI, Counter Profiling, Instrumentation, Hardware Performance Counters, Trace-based Profiling, PerfAPI, PCL, Scalable User Interface.**

Rolf Rabenseifner, High-Performance Computing-Center Stuttgart (HLRS), Rechenzentrum Universität Stuttgart (RUS), University of Stuttgart, Germany, http://www.hlrs.de/people/rabenseifner . Part of the work was done while the author was a visiting research associate at the Zentrum für Hochleistungsrechnen (ZHR), Technische Universität Dresden, Germany (Center for High-Performance Computing, Dresden University of Technology). E-mail: rabenseifner@hlrs.de .

I. Counter-Based Profiling

TODAY, job accounting on MPP hardware platforms does not provide enough information about the computational efficiency or about the efficiency of message passing (MPI) usage either to the users or to the computing centers. There is no information available about bandwidth and latency or integer and floating point operation rates achieved in real application runs. Therefore, users and hotline centers have no reliable information base for technical and political decisions with respect to programming and optimization investment. Existing trace-based profiling tools are too complicated for a first glance at an application and can be used in small test-jobs only, not in long-running production jobs.

To solve this problem, the High-Performance Computing-Center (HLRS) at the University of Stuttgart has combined the method of counter-based profiling with the techniques of writing system log-files. For each MPI routine, the number of calls, the time spent in the routine and the number of transferred bytes are written at the end of each parallel job to a syslog file at the computing center and, optionally, to a user file. The integration of the PCL library[1] allows the automatic instrumentation with the microprocessor's hardware performance counters (e.g. floating point instructions) to receive information about the computational efficiency of each program.[14] With that, the user has a criterion whether tuning the numerical part or the communication part promises greater benefit.

An analysis tool reads the syslog file and, on a weekly basis, sends a summary to each user about his jobs and writes a web-based summary for the computing center. The results of the first half-year on Cray T3E 900-512 at the HLRS are presented.[13] In a survey, our users showed that in the past, the profiling information was used only seldom for tuning the individual applications because the profiling tool was only available after the application development was finished and production was started. But 75 % of those interviewed believe that the profiling can help in the future to improve their applications.[16]

The profiling was implemented, tested and installed as default library on the T3E systems at HLRS Stuttgart and TU Dresden, and it is now ported to the Origin2000 at TU Dresden. The counter-based profiling only has a minimal overhead. The memory requirements on a T3E-900 are 200 kBytes. The counting requires 0.3 - 0.5 μsec per MPI call and writing the syslog file requires about 0.1 sec for each job. The overhead was 0.03 % of the application CPU time in the first half-year average in Stuttgart. Including the hardware counters, the overhead is about 300 kBytes memory, 2 μsec/call and about 0.1 - 0.2 % (expected) in all.

The PCL library was developed by the Forschungszentrum Jülich. For integrating the PCL library, the hardware counters' reading routine of the PCL library on the T3E has been optimized from about $35\,\mu$s to about $0.5 - 1.0\,\mu$s by removing the operating system calls.[1] This allows differentiation between counting hardware events inside and outside of the MPI routines. This is important because otherwise some hardware counters (load, integer instruction, any instructions) could not be used for measuring the user application since the busy wait operations of MPI would inflate their values.

Sections II and III describe the automatic MPI counter profiling as part of a scalable user interface. Section IV presents the software design for systems that have implemented MPI as Unix libraries or as dynamic shared objects (DSO). Section V lists which information is counted and Section VI describes how the encountered information is analyzed. Section VII shows the results of the first half-year statistics. Sections VIII and IX present the integration of the hardware performance counters and first statistical results. In Section X, a user survey is summarized. Section XI discusses related work and Section XII give the conclusions.

[1]Meanwhile, this system call is also removed in the new version of PCL

II. Scalable User-Interface

The user interface is designed so that the cost-benefit ratio is scalable for the user. This means that one is able to receive the most important information about an MPI application with an effort that is nearly zero, and if more information is wanted, there are several levels of support:

Level 1: Reading the first lines of an e-mail yields the MPI percentage, the instruction rate of all completed hardware instructions, the floating point instruction rate, and the level 2 cache miss rate of the jobs in the last week, if they have used the MPI default library.

Level 2: Reading the details of the e-mail yields for each MPI routine a summary of the calling count, execution time, and transferred bytes. Additionally, an estimate of the transfer and synchronization parts of the execution time is given. For each hardware performance counter, the total number of events and the rates are printed for the application part and for all MPI routines together.

Levels 1 and 2 could be implemented because the instrumentation is added to the default MPI library and writes the counters via the syslogd daemon to a syslog file.

Level 3: The user has to set a special environment variable in the MPI job before calling mpirun. As a consequence, the user gets all the information mentioned above (except the synchronization part estimate) for each parallel application run that uses the environment variable. For the future it is planned to implement a Web interface for the database created upon the syslog information. Then the user can view each job without setting the environment variable, and for levels 1-3 the user does not need to modify anything in the programs or job commands.

Level 4: The application can call MPI_Pcontrol to print out the state of the counters at any location inside the application. This gives all the information not only at the end of the application run, but also at any time and on each individual process (and not only the average of all processes).

Level 5: Setting one more environment variable, the user can choose other hardware performance counters, e.g. the integer instruction rate, level 1 data cache hits, and misses.

The levels 1 to 5 provide the information base to decide whether other tools should be used. Therefore, the next level is not part of the automatic MPI profiling, but part of the scalability strategy:

Level 6: Using trace based tools, e.g. Vampir, the user gets detailed insight into the computation and communication structure of the application. Using the PCL library directly or using other performance visualization tools can give a detailed insight into the computation efficiency and any problems of the application. PAT and Apprentice can also be used to locate and analyze the computational kernel of the application.

One of the key issues for this design is that on average the instrumentation overhead is in the range of a thousandth of the CPU time consumed by the applications. The automatic MPI profiling is a method to get enough information to decide whether the application is running as expected, one more chance to detect major bottlenecks and a basis to decide whether trace based tools or direct hardware counter instrumentation should be applied.

III. Comparing Counter-Based and Trace-Based Methods

The major differences between the counter and trace-based profiling can be seen by comparing the following characteristics: *Automatic counter-based MPI profiling* analyzes the whole MPI application. The analysis of long-running production jobs is not a problem. The extent of instrumentation must be restricted in time and memory to about a thousandth of the application because the tool is used for all MPI application runs. Only a small amount of data is written for each application run on the system's disk. The computing center and the users can get an overview about all jobs parallelized with MPI.

Trace-based profiling can normally be used to analyze short test jobs only. The instrumentation overhead can vary within the range of 5 to 10 percent because the user decides whether or not the tool is applied. Each trace can produce a large amount of data, written on the user's disk. The computing center does not get any information and long running parallel jobs can not be viewed, although these jobs are more important because they consume most of the computing resources.

IV. Software Design

To use the instrumented MPI library as default library, it is necessary to export the full MPI and PMPI interface for Fortran and C, as described in the MPI standard and implemented in the public and vendor's MPI libraries. This means that it was not possible to consume the PMPI profiling interface for the intended instrumentation, because otherwise the user is no longer able to use the PMPI-interface for privat profiling, e.g. with VAMPIRtrace [10]. Because the standard requires that the MPI routines are implemented by routines and not by macros, it is not possible to modify the MPI calls by adding macros to the mpi.h file. This method also cannot be used with Fortran. Therefore, a method had to be used that allowed two profiling layers. Because we do not have the source code of our vendor's MPI library and as we want to keep the instrumentation in a separate module, we have added this module to the library with two different methods. The Figures 1 and 2 show the different software designs for Unix libraries (archive libmpi.a) and dynamic shared objects (DSO libmpi.so).

A. Additional Profiling Layer with MPI Unix Library

For Unix libraries – see Fig. 1 –, the instrumentation is implemented as one wrapper routine to each MPI routine and added to the original MPI

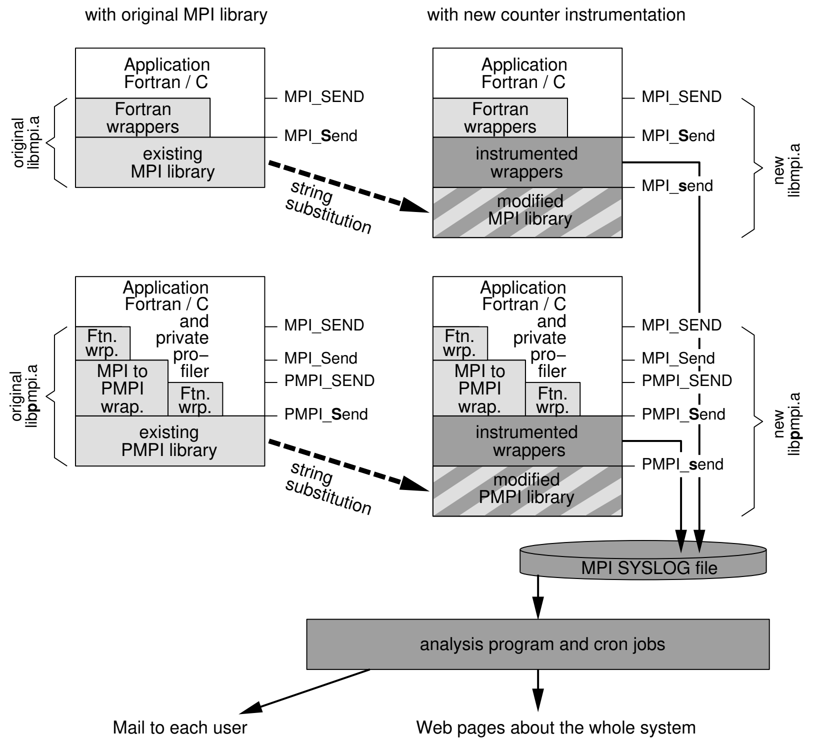

Fig. 1. The software design for MPI libraries (libmpi.a)

library, in which the original routines' names are modified with a binary-file editor.

The existing binary MPI libraries (libmpi.a and libpmpi.a) were modified by substituting the string of each MPI routine name by the same name, except that the last upper case letter was converted to lower case, e.g. MPI_Send was modified into MPI_send. The instrumentation is done in small wrapper routines named with the official MPI routine name (e.g. MPI_Send). They handle the counters for each routine and call the original MPI library routine, now named e.g. MPI_send. Fig. 1 shows this method. The method was developed for the MPI-GLUE project [12], in which it was necessary to combine three different MPI implementations (the metacomputing *glue*, the vendors MPI library and a global MPI library) inside of one executable.

In lib**p**mpi.a the same was done with the PMPI_... interface, e.g. PMPI_Send was substituted by PMPI_send. The lib**p**mpi.a consists of two parts, a) the real PMPI library routines, and b) the wrapper routines, e.g. the wrapper MPI_Send calls PMPI_Send. Prior to the substitution the wrapper routines must be extracted and afterwards added again.

If the Fortran interface is also implemented as wrapper routines, then these wrappers must be handled in the same way as the PMPI wrapper routines. In our case only two Fortran routines (MPI_SEND and MPI_RECV) were not implemented as wrappers. Therefore, they had to be implemented additionally for the profiling interface.

In the instrumented wrapper to MPI_Finalize the counters are written to a special syslog file by the syslog daemon. We use the instrumented MPI libraries as the default libraries. Therefore, all MPI applications are profiled only after they have been relinked since the installation of the instrumented MPI libraries.

Additionally, there exists a user interface. By setting an environment variable the user can write a copy of the information written by MPI_Finalize to a file or the standard output. With MPI_Pcontrol he can write subtotals of one MPI process or collectively of all MPI processes.

This interface was developed for a Cray T3E

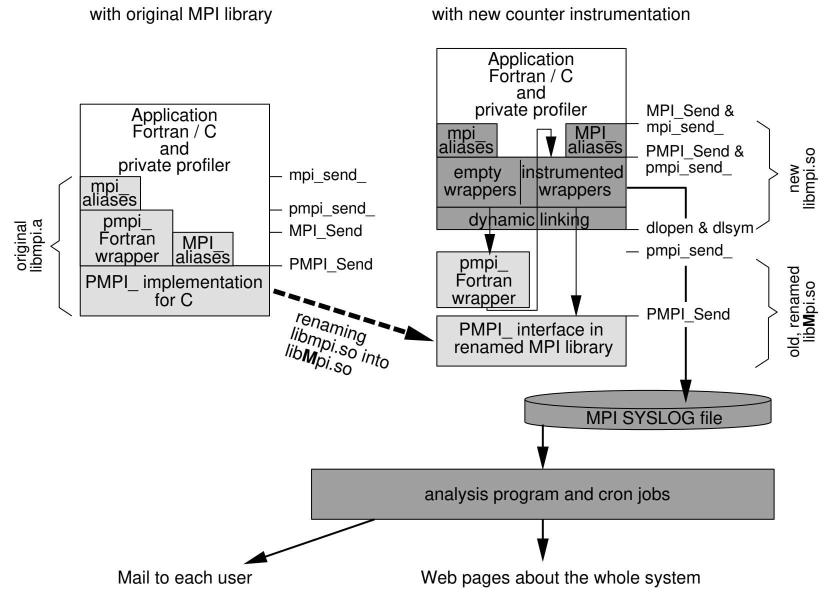

Fig. 2. The software design for MPI dynamic shared objects (libmpi.so)

system and is described in [13]. On the T3E, the profiling interface adds an overhead of 200 kbyte to the executable (i.e. 0.15 % of the available memory) , 0.385 μsec to each call to an MPI routine, and additional 0.190 μsec if a message length information is counted, and about 0.1 sec once for writing the statistics in MPI_Finalize. Compared with the minimal MPI message latency on our system (16 μsec) and an average runtime of more than 1 hour, the profiling overhead is marginal. Overall in the first 6 months the overhead was about 0.03 % of the CPU time of the profiled applications. This overhead was mainly produced by the instrumented wrappers and only about 10 % of the overhead was produced by writing the syslog file.

B. Dynamic Shared objects (DSO)

For DSOs – see Fig. 2 –, the new MPI-DSO libmpi.so contains only the instrumented wrappers and an initialization routine that binds the (up to this time) unresolved MPI references of the wrapper routines with dlopen and dlsym to the original MPI-DSO at start-up time. For this, the original MPI-DSO library was renamed libMpi.so. The dynamic linking at start-up time is necessary because static linking is impossible since the entry-point names of the instrumented wrapper routines and of the original MPI library routines are identical. This design requires that the original MPI routines never internally call other MPI routines. This is the case for the SGI MPI library. For applications written in C, this design adds only one additional subroutine call and the instrumentation.

For applications written in Fortran, the design depends on the implementation method of the Fortran MPI language binding. If a Fortran MPI routine is implemented as a Fortran-to-C wrapper routine that calls its C counterpart, then an empty wrapper for this Fortran interface must be added to the new libmpi.so and the dynamic linking establishes the following calling stack: application → new empty Fortran wrapper → original Fortran-to-C wrapper → new instrumented C wrapper → original MPI C interface. Compared with C, this case costs one additional subroutine call. If a Fortran MPI routine is implemented directly, then the Fortran wrapper in the new libmpi.so must be instrumented like the C wrapper, and there are no additional costs. This case may be necessary for routines with arguments that are externals or for optimized MPI routines. Additional wrappers must be included for the three special argument values MPI_NULL_COPY_FN, MPI_DUP_FN and MPI_NULLL_DELETE_FN in Fortran and C. This design was developed for SGI IRIX 6.5 and

```
Nov 28 22:09:57 hwwt3e syslog: B <56770>
size = n =   4  uid=843   mpt.1.2.1.0 / avg. on each PE:
argv[0] = ./mpi_bench2
send      1344 calls 2.94e+00 sec cnt= 9.4e+08 lng= 9.40e+08
recv      1344 calls 8.73e+00 sec cnt= 9.4e+08 lng= 9.40e+08
barrier    149 calls 1.29e+01 sec
bcast       40 calls 5.36e+01 sec cnt= 4.3e+01 lng= 3.44e+02
reduce    1682 calls 1.49e+01 sec cnt= 7.7e+07 lng= 6.15e+08
comm_size    2 calls 6.21e-06 sec
comm_rank    2 calls 3.39e-06 sec
attr_get     1 calls 2.99e-06 sec
wtime      430 calls 3.11e-04 sec
init         1 calls 5.96e-03 sec
finalize     1 calls 2.07e-05 sec
time     [sec] mpi=  9.32e+01 / all=  1.13e+02 = 8.3e-01
time * size    mpi=  3.73e+02 / all=  4.51e+02
profiling overhead=  1.54e-01 sec = time_all* 1.4e-03
```

Fig. 3. Example of the output of one MPI job

described in [16]. The library is portable to any system with fast subroutine calls and a fast local clock routine.

V. What is Counted

For each MPI job, i.e., for each application partition that uses MPI, the profiling counts:

- the number of processes in MPI_COMM_WORLD,
- the wall clock time of the application [sec],
- the wall clock time spent in all MPI calls in [sec] and as percentage of the application time,
- for each of the 128 MPI-1.2 routines:
  - the number of calls to the MPI routine,
  - the sum of time spent in the MPI routine,
  - the sum of the count arguments (if appropriate) and
  - the sum of the transferred bytes (if appropriate),
- the user identifier (UID),
- all MPI environment switches, e.g., the variable MPI_BUFFER_MAX defines the limit for using a buffered protocol for standard mode sends,
- whether the MPI or the PMPI library is used,
- whether the Fortran or the C language binding is used (indirectly via the usage of argc in MPI_Init),
- MPI implementation release name,
- the overhead added by the instrumentation in [sec] and as percentage of the application time,
- the time and date of the call to MPI_Finalize.

Fig. 3 shows an example of the output produced by MPI_Finalize on the special syslog file.

VI. The Analysis

First, any user can get a copy of the information on the syslog file for each MPI job. The computing center can also use the syslog file to analyze a specific user job if the user calls the hotline. The computing center uses an analysis program that computes each week a summary for each user and sends it as e-mail to the users. Fig. 4 gives an impression about the information sent to the users. The marked lines show the critical values. In the e-mail they are printed bold. Line (1) shows that the user has used 2.2 % of the whole system in that week. Line (2) shows the quotient of MPI to application time that is in the example with 58.5 % very bad. The reasons are visible in the lines (3) - (6). They are marked because most of the MPI time is spent in these MPI routines. Column 13 shows, that most of the MPI time is spent for waiting (synchronization time). This means that the application of this user is badly balanced. This enables the user to decide whether he requires more information or not, i.e. whether he should look

```
     analyzed jobs (partitions started with mpirun or mptrun):    7
     sum of used PEs:   232
     average of PEs used by one job:          33.1
     average of PEs (weighted by CPU time):   95.9
     usage of different MPI routines:           20
     sum of application CPU time: 6.76e+06 s =  1876.99 h
(1)  sum of application CPU time / available CPU time =    2.186 %
     sum of MPI time: 3.95e+06 s =  1098.16 h
(2)  sum of MPI time / sum of CPU time =   58.506 %
     overhead added by the profiling: 2.97e+02 s
     overhead / sum of CPU time = 4.39e-05
     used MPI language binding: C =     7 jobs
                          Fortran =     0 jobs
     the following environment variables are used in ... jobs:
      - MPIPROFOUT         in     2 jobs

     column 1             |   2   |   3   |   4   |   5   |   6   |   7   |   8   |  9  | 10  |  11   |  12   |  13
                          |       |       |       |       |       |       |       |     |     |       |       |
     MPI routine          |number|calls  | sum   | CPU   |sum of|sum of|trans-|input for|  estimated sum of
                          |  of  | per   | of    | time  |COUNT |COUNT*|ferred|col.11-13|laten-|trans-|idle/
                          |calls | PE    | CPU   | per   |argu- |P_SIZE|bytes |lat-|band|cy CPU|ferCPU|sync.
                          |      |       | time  | call  | ments|=trans| per  |ency|wdth|time  |time  |time
                          |      |       |[sec]  |[usec]|      |ferred| call |[us]|[MB |[sec] |[sec] |[sec]
                          |      |       |       |      |      | bytes|      |    | /s]|      |      |
     recv                 |703e+5|303097|263e+1|    37|403e+7|292e+8|   415|  10| 300|703e+0|928e-1|184e+1
     get_count            |278e+5|119857|368e-1|     1|      |      |      |    |    |368e-1|      |
(3)  ssend                |703e+5|303096|134e+4| 19072|403e+7|292e+8|   415|  50| 300|352e+1|928e-1|134e+4
     isend                |137e+6|591252|232e+2|   169|299e+9|243e10| 17734|  10| 100|137e+1|232e+2|-14e+2
     irecv                |137e+6|591252|389e+1|    28|677e10|542e11|394969|  10| 300|137e+1|233e+2|-21e+3
(4)  waitall              |138e+6|592770|403e+3|  2931|285e+6|      |      |  20|    |275e+1|      |400e+3
(5)  reduce               |346176|  1492|104e+4|301e+4|461e+8|181e+9|524126|  51|   6|177e-1|288e+2|101e+4
(6)  allreduce            |122e+5| 52559|114e+4| 93307|122e+5|975e+5|     8|  70|   6|854e+0|155e-1|114e+4
     comm_size            |   464|     2|106e-5|     2|      |      |      |    |    |106e-5|      |
     comm_rank            |   464|     2|685e-6|     1|      |      |      |    |    |685e-6|      |
     cart_create          |   232|     1|595e-2| 25649|      |      |      |    |    |595e-2|      |
     wtime                |274e+5|118253|549e-1|     2|      |      |      |    |    |549e-1|      |
     init                 |   232|     1|549e-2| 23671|      |      |      |    |    |549e-2|      |
     finalize             |   232|     1|762e-5|    33|      |      |      |    |    |762e-5|      |
```

Fig. 4. Example of a weekly user's analysis

at the output of single MPI application runs or whether he should use a trace-based profiling, e.g. VAMPIRtrace.

For most point-to-point and collective operations, an estimate is added that divides the MPI time into a latency time, a transfer time and a synchronization time. The estimate for the latency and transfer time is based on benchmark tests and on the number of counted calls and transferred bytes. The estimate for the synchronization time is the difference of measured MPI time minus the estimate for latency and transfer time.

For the sytems administrator, the analysis program produces weekly and monthly web pages with overviews about the system utilisation of the MPI applications. The web pages include

- a short summary with the most important rates (Fig 5),
- plots as presented in the next section in the figures 6-10,
- statistical details in the form of Fig. 4 of all MPI jobs, of those written in Fortran, and those written in C, separated by the consumed CPU time per job, and separated by the number of used MPI processes,
- a copy of all user's statistics,
- and special user's statistics that summarize only critical jobs, e.g. jobs with an MPI percentage higher than 15 %.

```
start of analysis Jul 12 20:40:03 1998
end   of analysis Jan 16 18:22:19 1999
interval                    187.9 days
cpu seconds              8.31e+09 sec
analyzed seconds         1.53e+09 sec = 18.4% of cpu seconds
MPI whole execution time 2.10e+08 sec = 13.7% of analyzed sec
- non-communication part 3.30e+05 sec =  0.0% of analyzed sec
- communication part:
  -- latency portion     1.05e+07 sec =  0.7% of analyzed sec
  -- transfer time       2.95e+07 sec =  1.9% of analyzed sec
  -- waiting/sync/idle   1.54e+08 sec = 10.1% of analyzed sec
- alltoallv,reduce_sc,scan 1.53e+07 s =  1.0% of analyzed sec
analyzed jobs               10289
nodes per job                32.4
nodes average                99.6 weighted by analyzed time
used MPI routines              92
number of users                68
95.3 % of the appl. CPU time was consumed by the top 21 users
99.2 % of the appl. CPU time was consumed by the top 29 users

Fortran jobs:
- time / analyzed            80.4 %
- jobs                       4865 = 47.3 % of analyzed
- nodes per job              40.7
- nodes average              89.7 weighted by analyzed time
- used MPI routines            67

C & C++ jobs:
- time / analyzed            19.6 %
- jobs                       5424 = 52.7 % of analyzed
- nodes per job              24.9
- nodes average             140.0 weighted by analyzed time
- used MPI routines            87
```

Fig. 5. Statistical overview of an analyzed time interval

VII. Results from 6 Months

In the interval reported in Fig. 5 we have profiled 68 users that have computed totally 425,288 hours on T3E processors. Based on an estimate of the latency of each routine, the statistics show that only a minimal portion of the MPI time is caused by latencies. Based on this result, we decided, not to install the mpich library as a second alternative for our users although mpich achieves a three times better latency than Cray's mpt library.[4] Fig. 6 shows that 18.4 % of the CPU time of the whole system was analyzed. The remaining 81.6 % can be split in about 30 % idle time, and in about 50 % for MPI applications that do not finish or do not call MPI_Finalize, or that were not linked with the instrumented default MPI library, or applications that use PVM, HPF or the native message passing library shmem as parallel programming method. This means that about a quarter of the accounted time was consumed by jobs that were analyzed with this counter profiling. Most of the MPI application time is spent in jobs with a CPU time per job larger than 10^5 sec (= 27.8 CPU hours). The figure shows also that most of the time was computed with MPI applications that use the Fortran language binding. Because we are using dedicated processors on the system, all CPU times are measured with the physical wall clock time. Instead of MPI_Wtime we have used a vendor specific intrinsic.

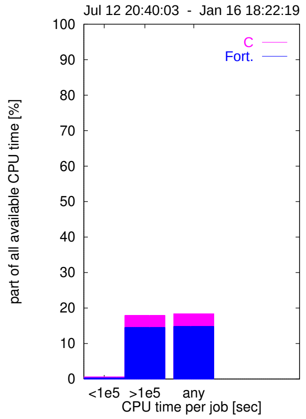

Fig. 6. MPI application time, Fortran and C

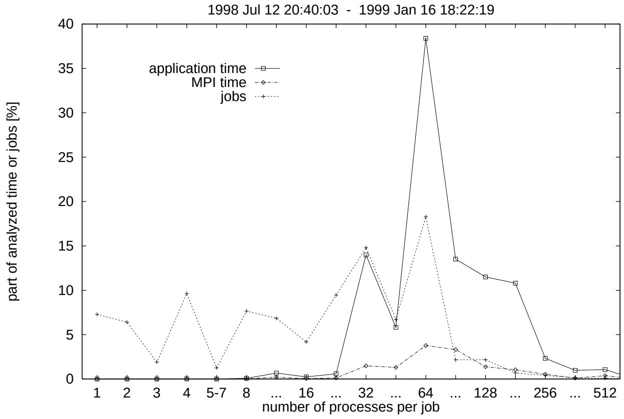

Fig. 8. Usage of different partition sizes

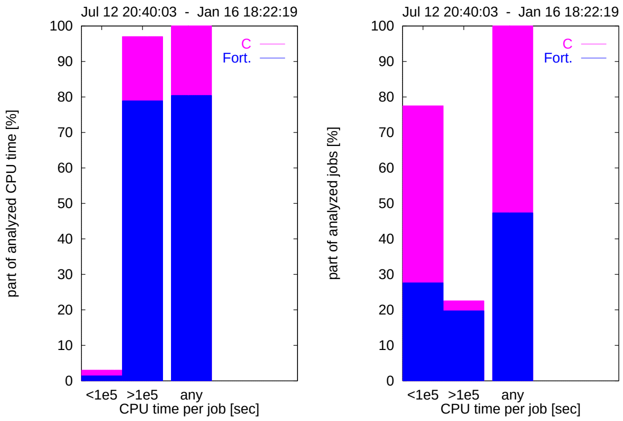

Fig. 7. Comparison of Fortran and C *application time* (left) and *jobs* (right picture)

Fig. 7 compares the parts of small and long jobs and of Fortran and C jobs. In the left picture the percentage is based on the total application time of all MPI applications, in the right picture it is based on the total number of MPI jobs. One can see, that we have a lot of small jobs that consume nearly no CPU time (these may be test runs in the program development phase), and that 2/3 of these jobs are written in C, in contrary to the long production jobs that are mainly written in Fortran.

The solid line in Fig. 8 shows that most (38 %) of the CPU time of MPI applications is spent in jobs with 64 PEs (MPI processes). 11 - 14 % of the CPU time is spent in each of the four categories with 32, 65-127, 128 and 129-255 PEs, followed by 6 % with 33-63 PEs, 2 % with 256 PEs and 1 % with 257-511 PEs and with 512 PEs. Weighted with the application CPU time on average each job has used 99.6 PEs. The dash-dotted line shows that the part of the CPU time of all analyzed MPI applications that is spent in MPI routines is small in all PE-categories. On average, the MPI time is 13.7 % of the analyzed application CPU time. The dotted line shows the percentage of jobs in each PE-category. It is dominated by test runs as mentioned in the explanations to Fig. 7.

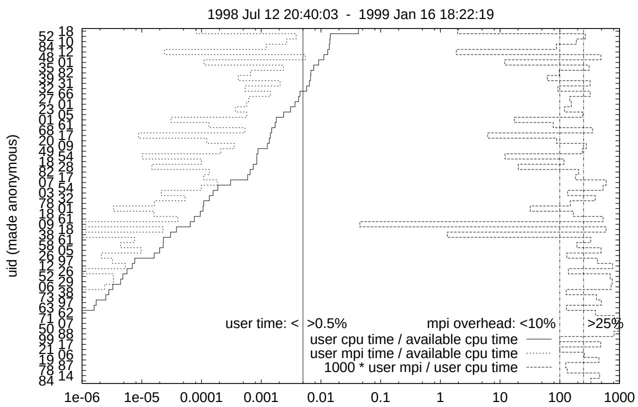

Fig. 9. Users' profiles

The solid line in Fig. 9 shows how much CPU time each user has consumed as part of the total time of the whole system in the analyzed time interval. The users are sorted by this value. The middle vertical bar marks 0.5 % of the total system. Each of the top 12 users has consumed more than 0.5 % of the total system. In total, these 12 users have consumed 15.1 % of the system (the other MPI users have consumed the remaining

3.3 %; in total the MPI users used 18.4 % of the system, see Fig. 5). For a sytem that is installed and reserved for high-performance computing, it is typical that only a few number of users dominate the system.

The left dotted line shows the time, the user applications have consumed in MPI routines. The right dashed line shows the ratio of MPI time to application time. The vertical bars mark a ration of 10 and 25 %. Looking at the top 12 users, one can see that 7 users have an MPI percentage less than 10 %, 1 user is in the range 10 - 25 %, 3 users have an MPI percentage of about 30 % and 1 user has about 50 %. The users with a very small MPI ratio mostly use direct shmem communication inside of their MPI programs.

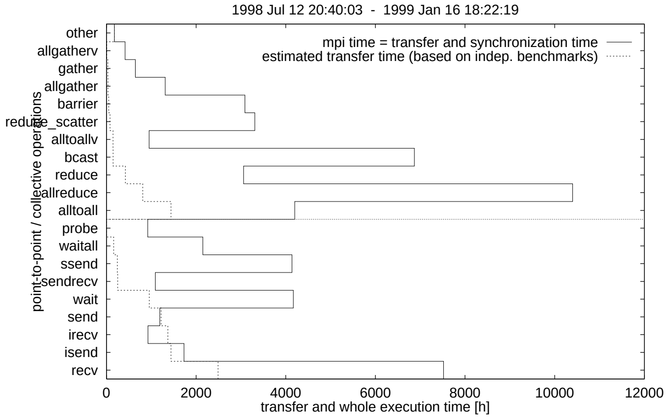

Fig. 10. MPI routines' profiles

The solid line in Fig. 10 shows the total time spent in the different MPI routines. The "other" entry summarizes all MPI routines that consumed less than 0.3 % of the total time spent in MPI routines, that was 58,260 hours. The picture shows that more than 99 % of the MPI time is consumed by only 19 different MPI routines. The other 73 used MPI routines consumed only 176 h, i.e. only 0.3 % of the MPI time. The dotted line presents an estimate of the underlaying transfer time. The difference between both lines is an estimate of the synchronization time. The estimate is the sum of the number of calls multiplied with the latency of the routine and the number of transferred bytes divided by the bandwidth. The latency and bandwidth estimates result from independent benchmarks.

The lower part of the picture shows point-to-point communication routines that have consumed about 23,836 h with a transfer part of 8,128 h and a synchronization part of 15,708 h. The upper part shows collective communication routines, that have consumed about 34,251 h with a transfer part of 3,167 h and a synchronization part of 31,085 h. This means that 72 % of the transfer time is consumed with point-to-point communication, the remaining 28 % are mainly used in MPI_Alltoall and MPI_Allreduce. 66 % of the synchronization idle time is wasted with collective communication routines, mainly with MPI_Allreduce, MPI_Bcast and MPI_Reduce_scatter. A significant part is also wasted in MPI_Barrier, MPI_Alltoall and MPI_Reduce. This indicates that we need parallel I/O, because an often used programming schema makes I/O in the root node and then uses MPI_Bcast or point-to-point communication to distribute the data among the nodes.

Because some users have combined MPI programming with direct shmem programming, the above mentioned transfer times present only the MPI based transfers and not the total transfer time of the analyzed applications.

MPI routines' group	in ...% of all jobs	weighted by CPU time
blocking point to point	66.304 %	65.858 %
nonblocking pt-to-pt	29.264 %	22.007 %
persistent pt-to-pt	0.010 %	0.000 %
pack and unpack	2.352 %	0.118 %
collective communication	92.545 %	99.996 %
derived datatype	23.530 %	1.468 %
group and sub-communicator	28.953 %	8.945 %
inter-communicator	0.029 %	0.009 %
attribute caching / inquiry	7.872 %	0.459 %
error handler handling	0.019 %	0.000 %
topology creation	13.568 %	15.685 %
wtime measurement	24.521 %	12.671 %

Fig. 11. Usage of the different MPI routines' groups.

Figure 11 shows which MPI routines' groups are called in how many jobs. In the right column the percentage is computed by weighting each job by its application CPU time. The routines MPI_Init, MPI_Finalize, MPI_Comm_size and MPI_Comm_rank are not included in any of these groups. The table shows that persistent communication, inter-communicator handling and individual error handling are not used on our system in the analyzed time interval. Pack and unpack are used very seldom. Derived datatypes, sub-groups and sub-communicators, attribute caching and inquiry, topology functions and wtime measurement are used mainly in applications that are still in the development phase and only seldom in production jobs. The right column presents the production jobs. Nearly every production job uses collective communication. But **nonblocking** point-to-point communication is used only in jobs that consume totaly 18.9 % of the analyzed CPU time.

Fig. 12 and Fig. 13 give the full information about the usage of MPI on our T3E in the half-year interval mentioned in Fig. 5. For each MPI routine that was used in that interval, one line is printed in the table. The MPI routines are grouped and each group is sorted by the accumulated CPU time spent in each routine. The columns 1, 2, 4, 6 and 7 summarize the original data on the syslog file. Column 3 prints the average number of calls in each PE, i.e. the value in column 2 divided by the sum of used PEs. In column 4, the CPU time means here physical wall clock time, i.e. the sum of real time spent in all calls to that MPI routine in all PEs. Column 5 gives the CPU time per call in micro-seconds, i.e. the value of column 4 divided by the value of column 2. Column 6 is the sum of the values of the count argument in each call. The COUNT argument is profiled in: Send, Recv, Bsend, Ssend, Rsend, Isend, Ibsend, Issend, Irsend, Irecv, Waitany, Testany, Waitall, Testall, Send_init, Bsend_init, Ssend_init, Rsend_init, Recv_init, Startall, Sendrecv_replace, Type_contiguous, Type_vector, Type_hvector, Type_indexed, Type_hindexed, Type_struct, Bcast, Alltoall, Allreduce, Scan. The INCOUNT argument is profiled in: Waitsome, Testsome, Pack. The OUTCOUNT argument is profiled in: Unpack. The SENDCOUNT argument is profiled in: Sendrecv, Gather, Gatherv, Allgather, Allgatherv, Alltoall. The RECVCOUNT argument is profiled in: Scatter, Scatterv. Column 7 is the sum of the product COUNT*MPI_Pack_size(DATATYPE) of the arguments COUNT and DATATYPE in each call to that MPI routine. If there are two datatype arguments then the datatype argument is used that corresponds to the count argument used in column 6. Exceptions are: This column represents the number of transferred bytes, except in MPI_Recv and MPI_Irecv, because there the COUNT denotes the length of the buffer and not the message length. In the Fortran language binding of MPI_Send and MPI_Recv the value of MPI_Pack_size(DATATYPE) is substituted by the length of a numerical Fortran unit. In column 8, the average of the transferred bytes per call is computed, i.e. the value of column 7 devided by the value of column 2. The columns 9 and 10 are an input for the estimate how the MPI time (column 4) is used, as latency (time needed to transfer zero bytes), as data transfer time, or as synchronization (idle) time. The values are the result of a benchmark with 2 (point-to-point) or 48 (collective) PEs. Column 9 represents the latency value of the benchmark in micro-seconds. In column 10, the bandwidth value of the benchmark (in MBytes/sec) is stored. The columns 11-13 give an estimate, how the MPI time (column 4) is used: as latency in column 11, as data transfer time in column 12, or as synchronization (idle) time in column 13. The value in column 11 is the product of the values in the columns 2 and 9. The value in column 12 is the product of the values in the columns 7 and 10. And the value in column 13 is the value of column 4 minus the values in the columns 11 and 12. Exceptions are: The transfer time values of Reduce_scatter, Alltoallv, Alltoall, Recv and Irecv are only rough estimates. For Reduce_scatter and Alltoallv there does not exist a *transferred-bytes* counter, therefore only a fixed, rough latency estimate can be used. With every call to Alltoall only the transferred bytes between a pair of processes is counted, but not the number of processes of the communicator. Therefore the size of the whole partition (MPI_COMM_WORLD) is used instead of the size of the actually used communicator. With Recv and Irecv often the count argument is larger than needed for the message

```
analyzed jobs (partitions started with mpirun or mptrun): 10289
sum of used PEs: 333323
average of PEs used by one job:          32.4
average of PEs (weighted by CPU time):   99.6
usage of different MPI routines:           92
sum of application CPU time: 1.53e+09 s = 425287.96 h
sum of application CPU time / available CPU time =   18.419 %
sum of MPI time: 2.10e+08 s = 58259.64 h
sum of MPI time / sum of application CPU time =   13.699 %
overhead added by the profiling: 4.56e+05 s =   126.77 h
overhead / sum of application CPU time = 2.98e-04

the following environment variables are used in ... jobs:
 - MPIPROFOUT         in  1260 jobs
 - MPI_BUFFER_MAX     in  1001 jobs
 - MPI_BUFFER_TOTAL   in     1 jobs
 - MPI_SM_POOL        in   469 jobs
 - MPI_SM_TRANSFER    in    21 jobs
the PMPI interface was used in    36 jobs
```

column 1	2	3	4	5	6	7	8	9	10	11	12	13
MPI routine	number of calls	calls per PE	sum of CPU time [sec]	CPU time per call [usec]	sum of COUNT arguments	sum of COUNT*P_SIZE =transferred bytes	transferred bytes per call	input for col.11-13 latency [us]	input for col.11-13 bandwdth [MB/s]	estimated sum of latency CPU time [sec]	estimated sum of transferCPU time [sec]	estimated sum of idle/sync. time [sec]
recv	544e+8	163091	271e+5	498	695e13	556e14	102e+4	10	300	543e+3	841e+4	181e+5
wait	217e+9	650230	150e+5	69				16		344e+4		116e+5
ssend	421e+7	12636	149e+5	3537	265e11	212e12	50449	50	300	209e+3	662e+3	140e+5
waitall	290e+8	86974	775e+4	267	626e+8			20		580e+3		717e+4
isend	130e+9	390022	622e+4	48	553e11	408e12	3142	10	100	130e+4	390e+4	102e+4
send	670e+8	200887	427e+4	64	496e11	397e12	5925	10	100	650e+3	373e+4	-10e+4
sendrecv	111e+8	33222	391e+4	353	125e11	998e11	9010	20	140	221e+3	680e+3	301e+4
irecv	151e+9	452074	333e+4	22	140e12	108e13	7185	10	300	149e+4	344e+4	-16e+5
probe	150e+6	451	331e+4	22052				8		120e+1		331e+4
bsend	387e+7	11624	170e+3	44	161e10	129e11	3331	23	100	891e+2	123e+3	-42e+3
waitany	292e+5	87	124e+3	4255	482e+6			20		583e+0		123e+3
test	181e+8	54218	110e+3	6						110e+3		
iprobe	797e+7	23906	630e+2	8						630e+2		
get_count	784e+7	23535	878e+1	1						878e+1		
buffer_detach	348e+6	1044	780e+0	2						780e+0		
buffer_attach	348e+6	1044	266e+0	1						266e+0		
ibsend	747194	2	209e+0	280	189e+7	189e+7	2530	23	100	172e-1	180e-1	174e+0
waitsome	1093	0	631e-1	57731	10900			20		219e-4		631e-1
testsome	146e+4	4	545e-1	37	800e+4					545e-1		
sendrecv_replace	16	0	277e-5	173	160000	160000	10000	20	140	320e-6	109e-5	136e-5
start	16	0	480e-6	30						480e-6		
request_free	14	0	115e-6	8						115e-6		
issend	4	0	101e-6	25	40000	40000	10000	50	300	200e-6	127e-6	-23e-5
startall	4	0	968e-7	24	4					968e-7		
send_init	4	0	812e-7	20	40000	40000	10000			812e-7		
ssend_init	2	0	548e-7	27	20000	20000	10000			548e-7		
bsend_init	2	0	460e-7	23	20000	20000	10000			460e-7		
rsend_init	2	0	416e-7	21	20000	20000	10000			416e-7		
type_commit	109e+5	33	382e+0	35						382e+0		
type_struct	102e+5	30	346e+0	34	230e+6					346e+0		
type_indexed	451961	1	129e+0	286	361e+6	576e+8	127502			129e+0		
type_free	106e+5	32	129e+0	12						129e+0		
type_vector	229528	1	107e-1	46	313e+4	454e+5	198			107e-1		
type_size	717e+4	22	266e-2	0						266e-2		
address	117e+5	35	189e-2	0						189e-2		
type_contiguous	10003	0	475e-4	5	293e+4	197e+5	1965			475e-4		
type_hvector	1536	0	139e-4	9	34640	821e+4	5348			139e-4		
type_extent	1536	0	252e-5	2						252e-5		

Notation: e.g. 670e+8 = 670.0e+8 = 6.70e+10 = 67,000,000,000

Fig. 12. Statistical details for the whole time interval (part 1)

column 1	2	3	4	5	6	7	8	9	10	11	12	13
MPI routine	number of calls	calls per PE	sum of CPU time [sec]	CPU time per call [usec]	sum of COUNT argu- ments	sum of COUNT* P_SIZE =trans ferred bytes	trans- ferred bytes per call	input for col.11-13 lat- ency [us]	band wdth [MB /s]	estimated sum of laten- cy CPU time [sec]	trans- ferCPU time [sec]	idle/ sync. time [sec]
unpack	385e+7	11551	403e+1	1	265e+8	212e+9	55			403e+1		
pack	385e+7	11551	334e+1	1	265e+8	212e+9	55			334e+1		
pack_size	170390	1	384e-4	0						384e-4		
allreduce	213e+8	63861	375e+5	1761	654e+9	904e10	425	70	6	148e+4	143e+4	346e+5
bcast	119e+7	3581	247e+5	20716	251e10	206e11	17267	35	40	414e+2	490e+3	242e+5
alltoall	425e+6	1277	151e+5	35534	741e11	595e12	140e+4	350	50	613e+2	512e+4	994e+4
+ column 6 and 7 (cnt,lng) on the alltoall line present an estimation of the whole length of + transfered data. The value is computed be multiplying the cnt and lng of each call to + MPI_ALLTOALL with the size of MPI_COMM_WORLD, instead of the actually used communicator.												
reduce_scatter	574e+6	1722	119e+5	20774				500		287e+3		116e+5
+ due to the lack of the value in col.7, only a fixed, rough latency estimate can be used for col.11+13												
barrier	601e+7	18022	111e+5	1852				30		180e+3		109e+5
reduce	927e+6	2781	110e+5	11864	141e10	925e10	9982	51	6	470e+2	147e+4	948e+4
allgather	586e+6	1757	471e+4	8046	239e+9	193e10	3292	110	32	644e+2	574e+2	459e+4
alltoallv	106e+7	3184	343e+4	3229				500		529e+3		290e+4
+ due to the lack of the value in col.7, only a fixed, rough latency estimate can be used for col.11+13												
gather	119e+7	3565	232e+4	1951	468e+9	365e10	3070	75	160	888e+2	217e+2	221e+4
allgatherv	834e+5	250	149e+4	17893	602e+8	454e+9	5437	110	32	918e+1	135e+2	147e+4
scatter	363623	1	146e+1	4011	230e+8	175e+9	481522	125	166	455e-1	101e+1	407e+0
scatterv	94813	0	566e+0	5970	633e+7	745e+7	78593	125	166	119e-1	428e-1	511e+0
gatherv	35874	0	185e+0	5170	704e+7	770e+7	214561	75	160	269e-2	459e-1	137e+0
scan	59391	0	153e+0	2583	144e+7	146e+7	24505	60	4	356e-2	347e+0	-20e+1
op_create	8393	0	317e-4	4						317e-4		
op_free	4054	0	142e-4	4						142e-4		
comm_create	97751	0	100e+3	102e+4						100e+3		
comm_split	69193	0	188e+0	2718						188e+0		
comm_dup	94267	0	349e-1	370						349e-1		
comm_rank	768e+4	23	176e-1	2						176e-1		
comm_size	444e+4	13	104e-1	2						104e-1		
group_excl	29371	0	233e-2	79						233e-2		
comm_free	103118	0	222e-2	22						222e-2		
group_incl	38412	0	788e-3	21						788e-3		
comm_group	79674	0	709e-3	9						709e-3		
group_free	69096	0	387e-3	6						387e-3		
group_range_incl	29968	0	363e-3	12						363e-3		
comm_compare	11552	0	560e-4	5						560e-4		
group_size	4697	0	964e-5	2						964e-5		
group_rank	640	0	172e-5	3						172e-5		
comm_test_inter	144	0	889e-6	6						889e-6		
group_translate_ranks	84	0	602e-6	7						602e-6		
attr_get	196225	1	820e-3	4						820e-3		
attr_put	52070	0	410e-3	8						410e-3		
keyval_create	976	0	452e-5	5						452e-5		
attr_delete	14	0	137e-6	10						137e-6		
cart_create	61052	0	774e+0	12679						774e+0		
cart_rank	138e+5	41	242e-1	2						242e-1		
graph_create	5247	0	214e-1	4074						214e-1		
cart_shift	125353	0	165e-2	13						165e-2		
cart_coords	224853	1	108e-2	5						108e-2		
dims_create	7545	0	219e-3	29						219e-3		
cart_get	2517	0	521e-4	21						521e-4		
init	333290	1	146e+2	43741						146e+2		
finalize	333265	1	131e+2	39370						131e+2		
wtime	108e+8	32305	929e+1	1						929e+1		
get_processor_name	26146	0	580e-1	2218						580e-1		
initialized	29955	0	565e-4	2						565e-4		
wtick	1670	0	205e-5	1						205e-5		
pcontrol	1807	0	920e-6	1						920e-6		
errhandler_set	12	0	364e-7	3						364e-7		

Fig. 13. Statistical details for the whole time interval (part 2)

that is received. Therefore for each job the sum of the length of all sent messages is computed and used as a maximum for the *transferred bytes* of Recv and Irecv. The idle time estimate in column 13 may be negative if the estimated latency and bandwidth in the columns 9 and 10 are too small, or at non-blocking routines, if the transfer time is hidden.

VIII. Adding Hardware Performance Counters

Each micro-processor has implemented some (2-4) hardware performance counters that are able to count one type of event among a given larger set of hardware events, e.g. completed instructions, floating point instructions, loads, stores or cache misses. The PCL library [1] developed by the Forschungszentrum Jülich is a common interface for currently 6 different micro-processors and access methods of the operating systems. PerfAPI [9] is a standardization effort of The Parallel Tools Consortium [19] to achieve a common interface to access the hardware performance counters.

To use these counters to measure the applications' efficiency, it is necessary to separate the hardware events generated by the application code and by the MPI library routines. This is not necessary for events that are generated very seldom by the MPI routines, such as *floating point instructions*, but the separation is absolutely necessary for events that are often generated by MPI, such as those *load instructions* used in the busy-wait implementation of MPI_Receive. To separate MPI and application events, it is necessary to have an extremely fast access to the hardware counters and to minimize data cache misses inside the instrumented wrappers (e.g. by minimizing any access to global variables).

Typically the operating system exports one of two different interface types: a) the hardware counters can be *read* like a clock, i.e. they are not reset after reading, and b) they can be *read out*, i.e. they are always reset to zero after reading them. For both interfaces, the instrumented wrappers have to implement one integer operation for each hardware counter before calling the original MPI routine and another one after returning from it:
In case a) events_in_mpi $-=$ counter must be issued before each MPI call and events_in_mpi $+=$ counter after its return and events_in_application $=$ $-$counter at the beginning and events_in_application $+=$ (counter $-$ events_in_mpi) at the end of the whole application.
In case b) events_in_application $+=$ counter must be issued before each MPI call and events_in_mpi $+=$ counter after its return and counter $=$ events_in_mpi $=$ events_in_application $=$ 0 at the beginning and events_in_application $+=$ counter at the end of the whole application. This means that the major requirement of a common **low-level** interface to access the micro-processors' hardware counters is that the two operations plus and minus must exist in the form

$$\begin{array}{l}\mathsf{for\ (i{=}0;\ i{<}number\_of\_hardware\_counters;\ i{+}{+})}\\ \quad \mathsf{local\_event\_counter[i]} \pm= \mathsf{hardware\_counter[i]}\end{array} \tag{1}$$

and also the information must be available whether this is a *read* or *read out*.

Unfortunately, neither the PCL library nor the PerfAPI interface design meet this requirement. E.g., the PCL library only has a *read out* interface, which adds additional operations, cache misses and resets of the hardware counter if the hardware supports the *read* interface. Additionally, PCL only has a high level interface that implements a matrix operation on the set of hardware counters and that implies at least an additional load/store for each counter. This matrix operation implements the mapping of the counters defined by PCL to any hardware counter or any difference or sum of several hardware counters. The matrix operation would be done twice for each MPI wrapper call, if PCL were used, instead of only once at the end of the application, which happens if the method (1) is used and the matrix operation is done only once before writing the results of events_in_mpi and events_in_application to the syslog-file. To integrate the hardware performance counters into the automatic MPI profiling, we have added an interface to the PCL library for the Cray T3E that is similar to the required inter-

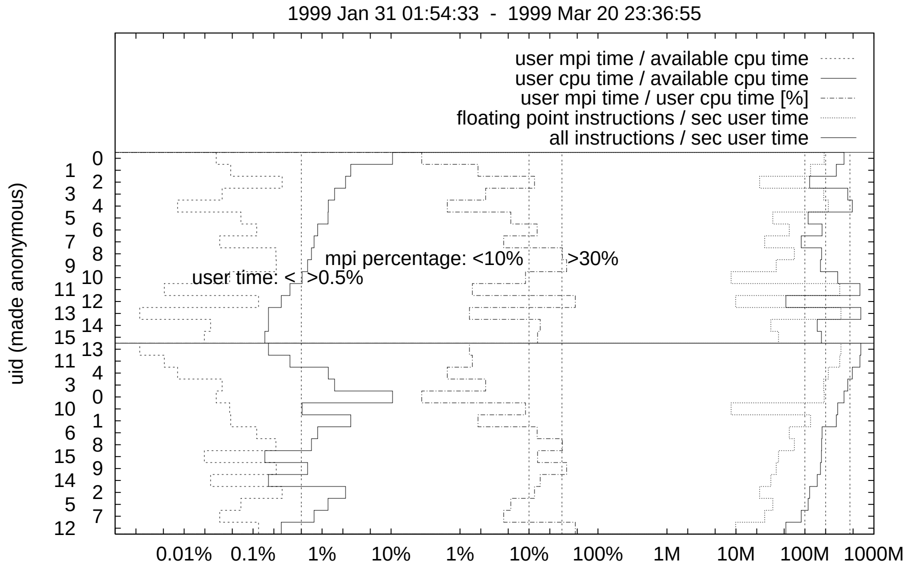

Fig. 14. Users' profiles, sorted by CPU time (upper part) and by instruction rate (lower part) – description see Sec. IX

face (1). With this and by removing all unnecessary operating system calls, the time to access the hardware performace counters could be reduced from 35 μs to about 0.5-1.0 μs.[2]

IX. First Results with Hardware Counters

Fig. 14 shows the users' profiles in the first seven weeks we used the hardware performance counters. Each row represents one user. The upper and lower part plot the same information, but differently sorted. The upper part is sorted by the CPU time consumed by each user, and the lower part by the total instruction rate. Each part combines three different plots:

The solid line in the left diagram represents the CPU time each user has consumed as part of the total time of the whole system in the analyzed time interval. The users are sorted by this value. The vertical bar marks 0.5 % of the total system. The figure represents the top 16 users of the HLRS that have used the new MPI library which was now instrumented with the hardware counters. In total, the 16 users have consumed 24.3 % of the whole system. The left dashed line shows the percentage of the time the user applications have consumed in MPI routines.

The diagram in the middle shows the ratio of MPI time to application time. The vertical bars mark a ratio of 10 and 30 %. The MPI percentage varies between 0.3 % and 46.8 %. On average, the MPI percentage was 5.2 %.

The diagram on the right shows the floating point instruction rate (dotted line) and the total instruction rate (solid line). The numbers are averages referring to one processor. On a T3E, each floating point instruction can execute one or two floating point operations. The floating point operation rate cannot be measured, but lies between the floating point instruction rate and twice the floating point rate. About half of the users' application runs could be used to analyse the hardware counters. The other jobs could not read the counters because the application was not yet relinked with the new library or else the partition was moved to other processors during execution. The floating point instruction rate of these 16 users varies between 9 and 333 MFL_instructions/sec (MFLips); weighted with the CPU time, the average is 138 MFLips, which implies that the MFLOP rate is between 138 and 276 MFLOPS, i.e. between 15 and 30 % of the peak performance of 900 MFLOPS. The vertical bars mark 100, 200 and

[2]Meanwhile, this system call is also removed in the new version of PCL

450 MFLips. The total instruction rate of the application code, except the MPI routines (solid line), is computed by dividing the number of instructions in the whole application minus the number of instructions executed in the MPI routines by the whole execution time. The total instruction rate varies between 53 and 647 M_instructions/sec.

The upper part of the picture helps to review the efficiency of the most relevant users. The lower part of the picture gives an insight into the correlation of MPI percentage and instruction rate. The numbers presented for each user (i.e. the MPI percentage, the floating point instruction rate and the total instruction rate) are a major information base for decisions with respect to programming and optimization investment since these numbers give a good overview of the achieved efficiency in computation and communication. The analysis tool sends these numbers and additional details to each user in a weekly mail.

X. User Feedback

The first feedback we received from our users by calling our MPI hotline to get help in optimizing their MPI programs after they noticed in the profiling e-mail that their MPI percentage was worse than expected. For a detailed feedback from our users we made a survey with 20 questions; finally, we received results from 28 of our users.[15] Major results of the survey are:

- The automatic profiling was regarded as *useful* in most answers.
- In the past, automatic profiling was only occasionally used for optimization because it came too late for most users, i.e. the application development was already completed and the programs are used in production.
- 75 % of those interviewed believe that the counter profiling can help to improve their applications in the future.
- Our weekly mail should be improved, too.
- We should provide the users with reference values. This would simplify the decision as to whether one should focus on further optimization.
- The analysis of the instruction rates of the applications' numerics and the MPI timings of the applications' communication are similarly important for the users.

For this survey, we have chosen customers who had computed at least four parallel jobs between February and April '99 and who already had received weekly mails from the automatic counter profiling system. The survey was taken by phone. Out of the 28 customers, 26 users are familiar with the automatic profiling. Out of these 26 users, 9 users always read the weekly mail, 9 users only sometimes, and 7 users rarely. Additional answers have shown that this depends on whether the users are testing their applications or, whether they have production jobs. This means that the level 1 and 2 of the profiling system is well accepted, which is also expressed by the average rank of 1.9 chosen by the 26 customers in the range of 1=*very useful* to 6=*not useful*. The third level was used by 8 users. The fourth level was used by one person. The fifth level never was used (perhaps because of the default that shows the total instruction rate and the floating point instruction rate). But the answers to the question whether instruction rates or MPI timings are more important were: *MPI timings are more important* (37 %), *both equal important* (53 %) and *instruction rates are more important* (10 %). The counter profiling is used to get information about the behavior of the users' programs (18 answers) and for optimizing the programs (11 answers). 4 users said that the profiling had really helped to improve their programs, 2 persons indicated that the MPI profiling was available too late, i.e. after they had finished the development. For the future, 21 users believe that the profiling will or may help to improve their programs. Additional optimization and analyses tools were used by 11 customers, i.e. only about 42 % have stepped to level 6. But only one user said that the MPI profiling had helped for the decision to use additional tools and 4 users told us that the profiling was available too late so that it could not help make this decision. On our system, mainly Vampir and Apprentice are used as additional optimization tools.

The last questions examined aspects of the interaction of the computing center with its customers. We also asked whether our customers – they all have publicly funded computing time on our systems – feel disturbed because the computing center also might look at the profiling information. Nobody was upset about this. Should the computing center use the overview and select optimal applications to learn about the optimization strategies used by the developers? 19 users would appreciate this, 5 don't care and 2 would be bothered. Should the computing center use the profiling information to contact the user with applications that could be inefficient? To our surprise, 22 users would like this, 2 don't care and 2 would be bothered.

Major wishes mentioned by the users are: To obtain reference values for the instruction rates, e.g. the minimum, maximum and average achieved by the other users; the layout of the weekly mail should be improved and should support html-based and ASCII-based mail readers; and it would be nice if the per-job information could be acquired from a database with a web interface.

XI. Related and Future Work

Riek et al.[17] give a comprehensive overview about monitoring and profiling systems. The hardware counters are accessed by using the PCL library.[1] The PerfAPI project [9] is a standardization effort for accessing of the hardware performance counters. Trace-based profiling and analysis systems are described in the references [2], [3], [5], [10], [11]. HP[6] has developed a local, user callable MPI counter profiling. Li and Zhang[7] combine counter profiling and a virtual clock approach to minimize the intrusiveness of the instrumentation. The new syslog-based MPI-counter-profiling closes a gap between the existing system-accounting analysis tools and the user-MPI-profiling tools. The system accounting gives an overview over the whole system, mainly about the usage of CPU and I/O resources, but does not report any details about specific MPI routines. In all cases, the existing MPI instrumentation libraries and profiling tools must be enabled by the user and they report only to the user. The key issue for syslog-based MPI-counter-profiling is, that the overhead of the instrumentation must be reduced to a minimum in contrast to the typical overhead of trace-based instrumentation. The usage of a portable instrumentation in contrast to proprietary profiling, as e.g. in [6], simplifies also the comparison of the usage of MPI on different systems. The scalability of the user interface has an analogy in the level-structured approach to learning, described in Shneiderman's[18] Principle 1. The optimization of a user application according to the levels of the profiling interface is described in the reference [16].

In the future we plan to generate global half-year statistics that include hardware counters. Also, it is planned to implement a user interface, which allows to read the data on the syslog file directly via a web interface. This reduces costs in level 3 to few clicks in the web browser. Based on the technology of TOPAS[8], the hardware counter profiling can be extended from the MPI applications to *all* applications. Besides the port to other platforms, we plan to add additional counters that allow to count the point-to-point and collective communication calls separated by the two's logarithm of the transferred bytes. This would add about 30 kbytes to the memory needs of the profiling module, but only a few instructions to the instrumentation of each routine (the two's logarithm can be computed mainly in 4 instructions). The results of this instrumentation can be used to optimize the default switch points between the different protocols used for standard mode message sending. The problem of correct profiling of checkpointed jobs, or jobs that are moved to another partition while the application is running, must be solved also.

XII. Conclusion

This project shows that combining the methods of counter profiling, job accounting and accessing the hardware performance counters can give more insight into the users' applications than achievable by previously used tools with similar costs.

The paper has shown two different methods to integrate an additional profiling level into existing MPI archives and dynamic shared objects without losing the standardized MPI profiling interface for other profiling tools. The automatic MPI profiling is a method to get enough information to decide whether the application is running as expected, one more chance to detect major bottlenecks and a basis to decide whether trace based tools should be used to optimize the communication pattern or whether direct hardware counter instrumentation should be applied to optimize the computational part.

On our Cray T3E 900-512 about a quarter of the accounted time was consumed by MPI jobs that were analyzed by this counter profiling. It is possible to analyze the usage of all MPI routines without adding a significant overhead to the consumed CPU time. The statistical analysis gives an overview, how MPI is used on the system. In six months on our T3E 900-512 it has turned out, that mainly the MPI Fortran language binding is used in production. Most data transfer is done with point-to-point communication, but nonblocking are used probably too seldom. Most synchronization time is consumed with collective routines. Most CPU time spent in MPI routines is consumend by 19 different MPI routines. In program development more than 50 % of the existing 128 MPI Fortran routines and more than 65 % of the MPI C routines are used. Because the counter profiling adds only a few data for each job to the special syslog file, the log files consume in compressed form only about 400 kbytes each month. We are mailing each week the individual users' statistics to each user. This is an effictive basis for the user to decide with which methods he can optimize his MPI application. The statistics can be used for well-directed advice. The automatic mailing reduces the inhibition level of using profiling tools. In the analyzed time interval 12 % of the jobs were profiled by the users by setting the environment variable to write the counter information directly to a user's file. But only 0.3 % of the jobs were instrumented with more sophisticated and complex methods like VAMPIRtrace. The MPI counter profiling is added to the current MPI release on our T3E 900-512 and installed as the default library. The instrumentation method is portable and can be ported to each system on which an additional routine call and two calls to the local clock do not add a significant overhead to MPI. The results have indicated, that the implementation and usage of the MPI-2 I/O chapter is more important for us than optimizing any of the existing MPI-1 routines. The current work can give more insight about the communication pattern used with MPI communication.

Acknowledgments

The author would like to acknowledge his colleagues at ZHR and HLRS and all the people that supported this project with suggestions and helpful discussions. He would like to thank especially K. Feind who gave the impetus for this work, Th. Beisel for proposing new ideas to the current work, U. Abele, H. Ohno, R. Supper and M. Wierse for giving me a lot of insight into details of the Cray operating system, M. Heine and E. Salo for the hints on SGI's MPI/DSO, R. Berrendorf and H. Ziegler for his support of the PCL library, W.E. Nagel for productive discussions and for the invitation to Dresden, and R. Rühle for giving major resources for this project.

References

[1] R. Berrendorf and H. Ziegler, *PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors*, internal report FZJ-ZAM-IB-9816, Forschungszentrum Jülich, Oct. 1998. www.fz-juelich.de/zam/docs/autoren98/berrendorf3.html

[2] M. Heath and J. Etheridge, *Visualizing Performance of Parallel Programs*, technical report TM-11813, Oak Ridge National Laboratory, TN, May 1991.

[3] M. T. Heath, *Recent Developments and Case Studies in Performance Visualization using ParaGraph*, Proceedings of the Workshop Performance Measurement and Visualization of Parallel Systems, G. Haring and G. Kotsis (ed.), Moravany, Czechoslovakia, Oct. 1992, pp 175–200.

[4] L. S. Hebert, W. G. Seefeld and A. Skjellum, *MPICH on Cray T3E*, Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99), Atlanta, USA, March 1999, pp 69–76.

[5] V. Herrarte and E. Lusk, *Studying Parallel Program Behavior with Upshot*, Argonne National Laboratory, technical report ANL-91/15, Aug. 1991

[6] *HP MPI User's Guide*, 4.1 Using counter instrumentation, HP, B6011-90001, Third Ed., June 1998.

[7] K-C. Li and K. Zhang, *Supporting Scalable Performance Monitoring and Analysis of Parallel Programs*, The Journal of Supercomputing, 13, pp 5–31, 1999.

[8] B. Mohr, *TOPAS – Automatic Performance Statistics Collection on the CRAY T3E*, Proceedings of the 5th European SGI/Cray MPP Workshop, Sept. 9–10, 1999. www.cineca.it/mpp-workshop/abstract/bmohr.htm and www.fz-juelich.de/zam/docs/autoren99/mohr.html

[9] P. J. Mucci, S. Browne, G. Ho and C. Deane, *PerfAPI - Performance Data Standard and API.* http://icl.cs.utk.edu/projects/papi/

[10] W.E. Nagel et al., *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Volume XII, Number 1, Jan. 1996, pp 69–80, and technical report KFA-ZAM-IB-9528, www.fz-juelich.de/zam/docs/printable/ib/ib-95/ib-9528.ps

[11] W.E. Nagel and A. Arnold, *Performance Visualization of Parallel Programs: The PARvis Environment*, technical report, Forschungszentrum Jülich, 1995. www.fz-juelich.de/zam/PT/ReDec/SoftTools/PARtools/PARvis.html

[12] R. Rabenseifner, *MPI-GLUE: Interoperable High-Performance MPI Combining Different Vendor's MPI Worlds*, Proceedings of the Euro-Par'98, 4th International Euro-Par Conference, D. Pritchard, J. Reeve (ed.), Southampton, UK, Sept. 1998, LNCS 1470, pp 563–569.

[13] R. Rabenseifner, *Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512*, Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99), Atlanta, USA, March 1999, pp 77–85. www.hlrs.de/people/rabenseifner/publ/publications.html

[14] R. Rabenseifner, *Automatic Profiling of MPI Applications with Hardware Performance Counters*, in J. Dongarra et al. (eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 6th PVM/MPI European Users' Group Meeting, EuroPVM/MPI'99, Barcelona, Spain, Sept. 26–29, 1999, LNCS 1697, pp 35–42. www.hlrs.de/people/rabenseifner/publ/publications.html

[15] R. Rabenseifner, *Umfrage zum automatischen MPI Counter Profiling auf der T3E*, only online: www.hlrs.de/mpi/umfrage_results.html

[16] R. Rabenseifner, S. Seidl and W. E. Nagel, *Effective Performance Problem Detection of MPI Programs on MPP Systems: From the Global View to the Detail*, Parallel Computing '99 (ParCo99), Delft, the Netherlands, August 1999. www.hlrs.de/people/rabenseifner/publ/publications.html

[17] M. van Riek, B. Tourancheau and X.-F. Vigouroux, *Monitoring of Distributed Memory Multicomputer Programs*, University of Tennessee, technical report CS-93-204, and Center for Research on Parallel Computation, Rice University, Houston Texas, technical report CRPC-TR93441, 1993. http://www.netlib.org/tennessee/ut-cs-93-204.ps and ftp://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93441.ps.gz

[18] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer-Interaction*, 3rd ed., Addison-Wesley, March 1998.

[19] The Parallel Tools Consortium – www.ptools.org

Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he has worked at the High-Performance Computing-Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without loosing the full MPI interface. In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum. From January to April 1999, he was an invited reseacher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is responsible for message passing programming models at the HLRS and he is involved in MPI-I/O, MPI profiling, benchmarking and teaching projects.