Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512

Rolf Rabenseifner

Abstract— **This paper presents an automatic counter instrumentation and profiling module added to the MPI library on our Cray T3E. A statistical summary of the MPI calls of any MPI partition is gathered during execution and written in MPI_Finalize on a special syslog file. The user can get the same statistical information on a file. Weekly and monthly a statistical summary is computed and the user specific part is sent by mail to each user. A summary of 6 month is 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. The software is portable to other systems.**

Keywords— **MPI, Counter Profiling, Instrumentation.**

I. Introduction and Motivation

In the past, instrumentation and profiling was done by the user. There exist good tools for trace-based profiling, e.g. VAMPIR [6], and for counter-based profiling, e.g. the counter instrumentation and analysis from HP [5]. These tools need explicite user intervention and do not give any statistical information over all jobs running on a system.

Based on the question which optimization of MPI routines give the best benefit to our users, we decided to develop a counter-based instrumentation for the whole system, covering all MPI programs with a minimal overhead.

Based on a method developed in the MPI-GLUE project [8] we could add a profiling layer between the MPI calls of the user's application and the MPI library routines without loosing the official PMPI interface. Because we are modifying the official MPI library and adding the profiling layer to that library, the user need not to change anything.

Although the first motivation was to identify mainly used MPI operations, now the main benefit is the automatically generated information for the users. Each week, we produce a summary of all jobs of each user and send it to them. Based on the quotient MPI time to CPU time the user can easily see whether the communication is a bottleneck. Detailed informations about each MPI routine give her/him an impression which part of its application should be optimized.

Rolf Rabenseifner, Rechenzentrum Universität Stuttgart (High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart), 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 (Center for High-Performance Computing, Dresden University of Technology). E-mail: rabenseifner@hlrs.de .

II. The Instrumentation

If the instrumentation should be done transparently to the users one cannot use the standardized MPI profiling interface (PMPI_...) because otherwise the user is no longer able to use the PMPI-interface for privat profiling, e.g. with VAMPIRtrace [6]. 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, the instrumentation has to be added to the MPI library. 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 the following trick:

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 trick. The trick was developed for the MPI-GLUE project [8], 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.

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.

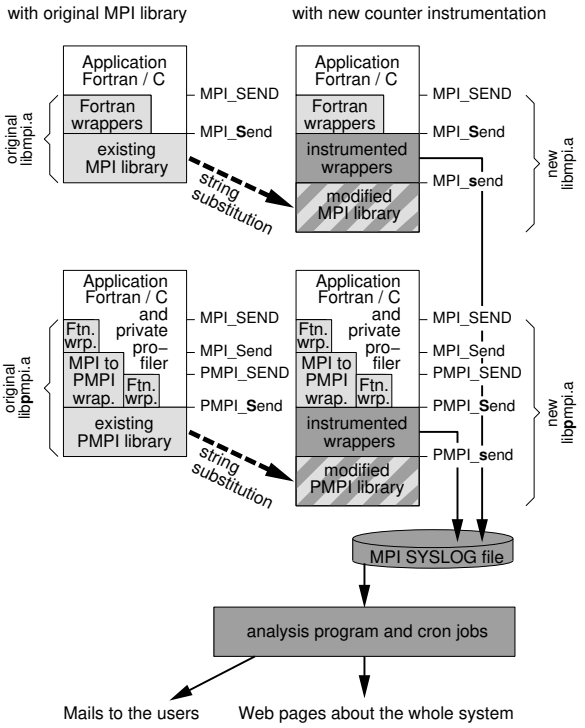

Fig. 1. The software design

III. 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. MPI_BUFFER_MAX that defines the limit for using a buffered protocol for standard mode sends,

```
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. 2. Example of the output of one MPI job

- 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. 2 shows an example of the output produced by MPI_Finalize on the special syslog file.

IV. 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. 3 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 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 pro-

```
     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 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 =transferred bytes	trans- ferred bytes per call	input for col.11-13 laten- cy [us]	input for col.11-13 band wdth [MB /s]	estimated sum of laten- cy CPU time [sec]	estimated sum of trans- ferCPU time [sec]	estimated sum of idle/ sync. time [sec]
	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. 3. Example of a weekly user's analysis

duces 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 4),
- plots as presented in the next section in the figures 5-9,
- statistical details in the form of Fig. 3 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 %.

V. Results from 6 Months

In the interval reported in Fig. 4 we have profiled 68 users that have computed totally 425,288 hours on T3E processors. Fig. 5 shows that this means 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 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. 7 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

```
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. 4. Statistical overview of an analyzed time interval

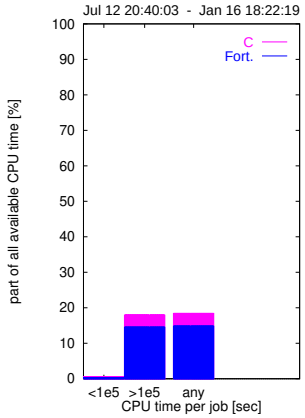

Fig. 5. MPI application time, Fortran and C

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. 6.

The solid line in Fig. 8 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

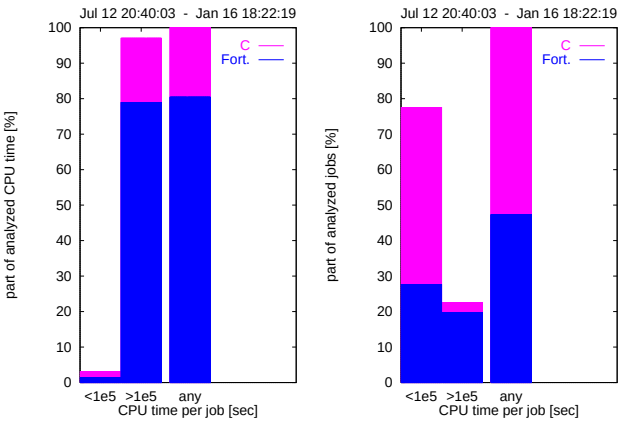

Fig. 6. Comparison of Fortran and C *application time* (left) and *jobs* (right picture)

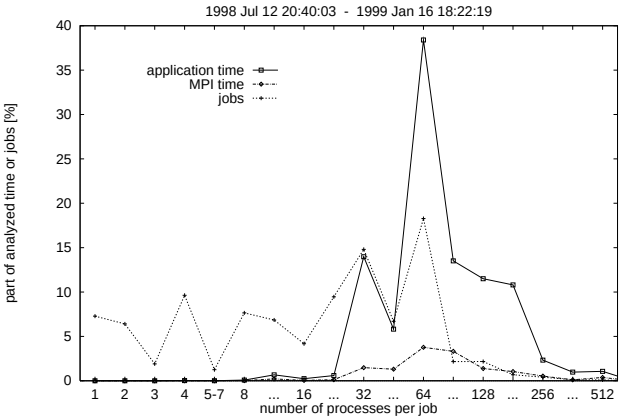

Fig. 7. Usage of different partition sizes

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. 4). 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.

The solid line in Fig. 9 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

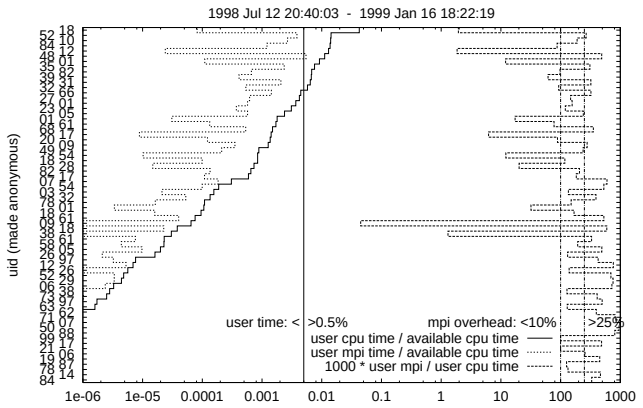

Fig. 8. Users' profiles

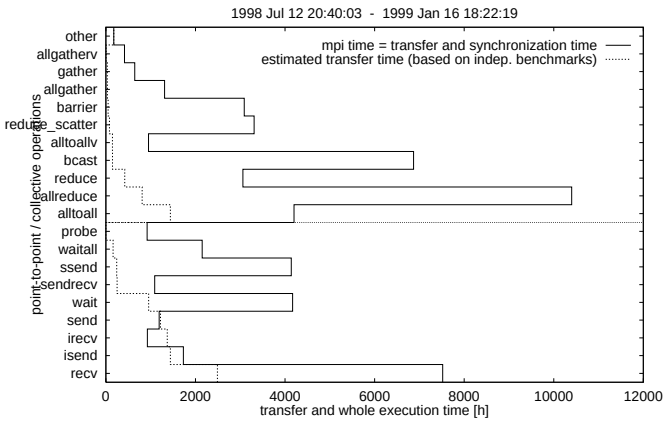

Fig. 9. MPI routines' profiles

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.

TABLE I
USAGE OF THE DIFFERENT MPI ROUTINES' GROUPS.

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 %

Table I 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, intercommunicator 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, subgroups 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. 10 and Fig. 11 give the full information about the usage of MPI on our T3E in the half-year interval mentioned in Fig. 4. For each MPI routine that was used in that interval, one line is printed in the table. 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, Gath-

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

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

Fig. 10. 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 =transferred 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]
pack	385e+7	11551	334e+1	1	265e+8	212e+9	55			334e+1		
unpack	385e+7	11551	403e+1	1	265e+8	212e+9	55			403e+1		
pack_size	170390	1	384e-4	0						384e-4		
barrier	601e+7	18022	111e+5	1852				30		180e+3		109e+5
bcast	119e+7	3581	247e+5	20716	251e10	206e11	17267	35	40	414e+2	490e+3	242e+5
gather	119e+7	3565	232e+4	1951	468e+9	365e10	3070	75	160	888e+2	217e+2	221e+4
gatherv	35874	0	185e+0	5170	704e+7	770e+7	214561	75	160	269e-2	459e-1	137e+0
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
allgather	586e+6	1757	471e+4	8046	239e+9	193e10	3292	110	32	644e+2	574e+2	459e+4
allgatherv	834e+5	250	149e+4	17893	602e+8	454e+9	5437	110	32	918e+1	135e+2	147e+4
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.												
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												
reduce	927e+6	2781	110e+5	11864	141e10	925e10	9982	51	6	470e+2	147e+4	948e+4
op_create	8393	0	317e-4	4						317e-4		
op_free	4054	0	142e-4	4						142e-4		
allreduce	213e+8	63861	375e+5	1761	654e+9	904e10	425	70	6	148e+4	143e+4	346e+5
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												
scan	59391	0	153e+0	2583	144e+7	146e+7	24505	60	4	356e-2	347e+0	-20e+1
group_size	4697	0	964e-5	2						964e-5		
group_rank	640	0	172e-5	3						172e-5		
group_translate_ranks	84	0	602e-6	7						602e-6		
comm_group	79674	0	709e-3	9						709e-3		
group_incl	38412	0	788e-3	21						788e-3		
group_excl	29371	0	233e-2	79						233e-2		
group_range_incl	29968	0	363e-3	12						363e-3		
group_free	69096	0	387e-3	6						387e-3		
comm_size	444e+4	13	104e-1	2						104e-1		
comm_rank	768e+4	23	176e-1	2						176e-1		
comm_compare	11552	0	560e-4	5						560e-4		
comm_dup	94267	0	349e-1	370						349e-1		
comm_create	97751	0	100e+3	102e+4						100e+3		
comm_split	69193	0	188e+0	2718						188e+0		
comm_free	103118	0	222e-2	22						222e-2		
comm_test_inter	144	0	889e-6	6						889e-6		
keyval_create	976	0	452e-5	5						452e-5		
attr_put	52070	0	410e-3	8						410e-3		
attr_get	196225	1	820e-3	4						820e-3		
attr_delete	14	0	137e-6	10						137e-6		
cart_create	61052	0	774e+0	12679						774e+0		
dims_create	7545	0	219e-3	29						219e-3		
graph_create	5247	0	214e-1	4074						214e-1		
cart_get	2517	0	521e-4	21						521e-4		
cart_rank	138e+5	41	242e-1	2						242e-1		
cart_coords	224853	1	108e-2	5						108e-2		
cart_shift	125353	0	165e-2	13						165e-2		
get_processor_name	26146	0	580e-1	2218						580e-1		
errhandler_set	12	0	364e-7	3						364e-7		
wtime	108e+8	32305	929e+1	1						929e+1		
wtick	1670	0	205e-5	1						205e-5		
init	333290	1	146e+2	43741						146e+2		
finalize	333265	1	131e+2	39370						131e+2		
initialized	29955	0	565e-4	2						565e-4		
pcontrol	1807	0	920e-6	1						920e-6		

Fig. 11. Statistical details for the whole time interval (part 2)

erv, 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 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.

VI. Related Work

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. The existing MPI instrumentation libraries and profiling tools are trace-based, like ParaGraph [2], [3], Upshot [4] and VAMPIR [6], [7], or counter-based, as described e.g. in [5]. In all cases they 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 [5], simplifies also the comparison of the usage of MPI on different systems. Riek et al. give in [9] a comprehensive overview about monitoring and profiling systems.

VII. Current Work and Outlook

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.

Integrating the PCL library [1] will allow the automatic instrumentation with the microprocessor's hardware performance counters (e.g. completed floating point or integer instructions) to get hints about the computational efficiency of each program. For this, it was necessary and possible to improve the hardware counters' reading routine of the PCL library on the T3E from about $35\,\mu$s to about $0.5 - 1.0\,\mu$s by removing the operating system calls. This will allow to differentiate between counting hardware events inside and outside of the MPI routines. This is important, because otherwise some hardware counters (load, integer instruction, successful jump, any instructions) could not be used for measuring the user application because the busy wait operations of MPI would inflate their values.

Moving the initialization and counter-writing from MPI_Init/Finalize to the beginning and exit of the user's program and implementing the reduction operation of the profiling counters by a low-level communication routine will allow to profile at least the performance counters in any application. 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.

Another goal is to analyze also the calls to shmem routines. It is important here to differentiate between shmem calls inside the MPI routines and those outside. Analyzing also non-MPI applications we can get an estimate of the total usage of the communication and computation hardware.

VIII. Conclusions

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 non-blocking 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 and with the vendor's one-sided communication (shmem).

Acknowledgments

The author would like to acknowledge his colleagues and all the people that supported this project with suggestions, help and discussions. He would like to thank especially K. Feind who gave the impetus for this work, Th. Beisel and W. E. Nagel 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, R. Rühle and the HLRS for giving me the resources for this project and W. E. Nagel for his invitation to implement and test the new features on their system in Dresden.

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. http://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, p. 175-200.

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

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

[6] W. E. Nagel et al., *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Volume XII, Number 1, Jan. 1996, pp. 69-80. http://www.kfa-juelich.de/zam/docs/autoren95/nagel2.html and Technical Report KFA-ZAM-IB-9528, Jan. 1996 ftp://ftp.zam.kfa-juelich.de/pub/zamdoc/ib/ib-95/ib-9528.ps

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

[8] 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.

[9] 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

Rolf Rabenseifner has studied mathematics and physics at the University of Stuttgart. He works at the High-Performance Computing-Center Stuttgart (HLRS). After developing the DFN-RPC, a remote procedure call tool, and developing a controlled logical clock as global time for trace-based profiling and designing and implementing MPI-GLUE, the first meta-computing MPI combining different vendor's MPIs without loosing the full MPI interface, he is now responsible for MPI at the HLRS. From January to April 1999 he is invited for a research stay at the Center for High-Performance Computing at Dresden University of Technology.