
Improved Performance Scaling on HPCx

Performance Analysis



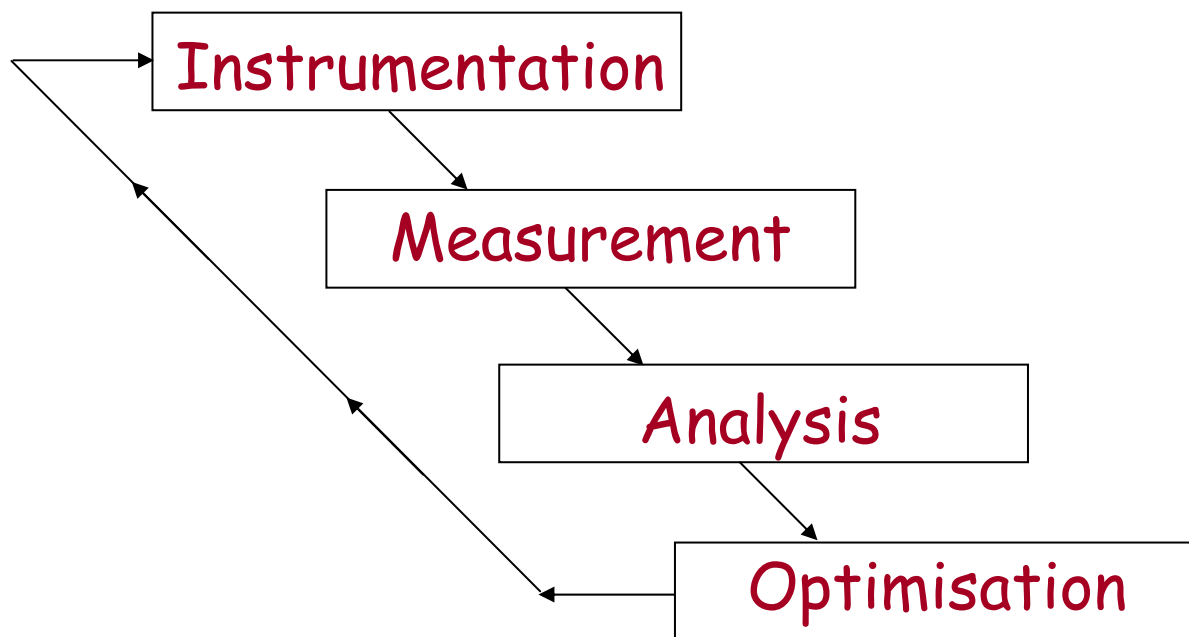
- Introduction and motivation
 - Timers
 - Hardware event counters
 - Profiling tools
 - Communication tools
-

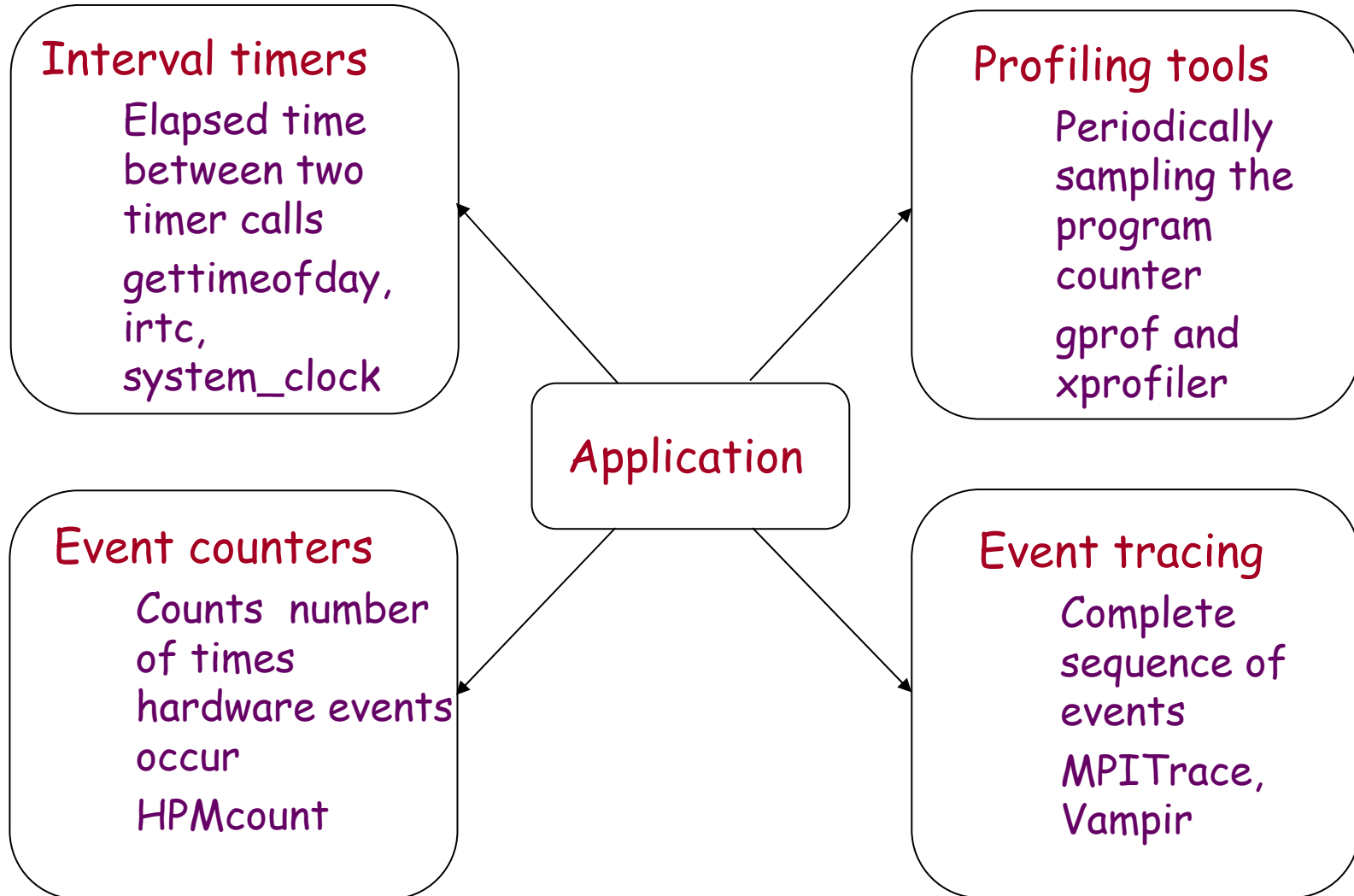
- **Our aim**
 - To make effective use of current HPC architectures (i.e. HPCx)
 - **Hence**
 - Must achieve the best performance from our applications
 - Both serial and parallel performance
 - **Only possible through**
 - Performance analysis and optimisation
 - **This tutorial**
 - Particularly interested in parallel performance
-

- **Goals**
 - To reduce the execution time of the code
 - To achieve good scaling to large numbers of processors
 - To reduce memory/disk usage?
 - **Requires identification of bottlenecks before optimisation**
 - Components of the code which take a large amount of time to execute
 - Pieces of code that make inefficient use of underlying architecture
-

- **Your application**
 - Synchronisation, load balance, communication, memory usage, IO usage
 - **The architecture**
 - Memory hierarchy, network bandwidth and latency, the processor architecture, IO system setup
 - **Software**
 - Compiler options, libraries, communication protocols, operating system
 - **To name but a few...**
-

- This is an iterative process
- Involves the use of tools to
 - Instrument code, measure and analyse performance metrics





- Wide range of timers available on HPCx
 - Varying precision, portability, language, ease of use
 - Useful for determining components of the code that take a large amount of time
 - Problems
 - Time consuming to instrument code
 - Parallel applications can generate vast amounts of data
 - Timing results per processor, thousands of processors...
-

Timers on HPCx

Timer	Usage	Wallclock/ CPU	Resolution	Language	Portable
time	shell	both	centisecond	F, C/C++	yes
timex	shell	both	centisecond	F, C/C++	yes
gettimeofday	subroutine	wallclock	microsecond	C/C++	yes
read_real_time	subroutine	wallclock	nanosecond	C/C++	no
rtc	subroutine	wallclock	microsecond	F	no
irtc	subroutine	wallclock	nanosecond	F	no
dtime_	subroutine	CPU	centisecond	F	no
etime_	subroutine	CPU	centisecond	F	no
mclock	subroutine	CPU	centisecond	F	no
timef	subroutine	wallclock	millisecond	F	no
MPI_Wtime	subroutine	wallclock	microsecond	F, C/C++	yes
system_clock	subroutine	wallclock	centisecond	F	yes

- **prof and gprof**
 - Standard unix commands provide simple profiling at the subroutine level
 - **xprofiler**
 - Simple profiler for both serial and parallel applications, GUI alternative to gprof
 - **Both useful for determining components of the code that take a large amount of time**
 - Less time consuming for the programmer
 - **Problems**
 - Intrusive - may increase overall execution time
 - Only profiles CPU (busy) usage, lack of I/O and communication information
-

Profiling - XProfiler

The screenshot displays the XProfiler V1.2 interface on an IBM RS/6000 SP. The main window shows a call graph with nodes representing functions and edges representing calls. A 'Function Menu' is open, showing a 'Statistics Report' for the function '.forces'. The report indicates that the function was called 50 times. Below the menu, the source code for 'forces.f' is displayed in a separate window. The code is a Fortran subroutine that computes forces and accumulates the virial and potential using OpenMP parallelization.

Function Menu

Statistics Report

Show
Function Level Statistics Report

File Utility Help

NO. TICKS
line NOT AVAILABLE source code

```
1 subroutine forces(npart, x, f, vir, epot, side, rcoeff)
2 implicit double precision (a-h, o-z)
3 dimension x(npart, 3), f(npart, 3)
4
5 c compute forces and accumulate the virial and potential.
6 c
7 !$OMP SINGLE
8 vir = 0.0d0
9 epot = 0.0d0
10 !$OMP END SINGLE
11 sideh = 0.5d0*side
12 rcoffs = rcoeff*rcoeff
13 c
14 !$OMP DO PRIVATE(j,xi,yi,zi,fxi,fyi,fzi,xx,yy,
15 !$OMP& zz,rd,rrd,rrd2,rrd3, rrd4,rrd6,rrd7,r148,forcex,forcey,forcez),
16 !$OMP& REDUCTION(-:vir), REDUCTION(+:epot),
17 !$OMP& SCHEDULE (STATIC,16)
18 do 270 i = 1,npart
19 xi = x(i,1)
20 yi = x(i,2)
```

Search Engine: (regular expressions supported)

forces

- Hardware Performance Monitor Toolkit (HPM)
 - hpmcount, libhpm, hpmviz
 - hpmcount provides performance details
 - Execution wall clock time
 - Hardware performance counter information (e.g. cache misses)
 - Derived hardware metrics (e.g. loads per cache miss)
 - Resource utilisation statistics (e.g. no of page faults)
-

- Interval timers, event counters and profiling tools are all useful for serial and parallel performance analysis
 - Have reviewed previously on the performance optimisation course
 - Event Tracing tools on HPCx
 - MPITrace
 - Vampir
 - Particularly useful for analysing scaling
 - Detailed information about communication time
-

- Allows low-overhead MPI elapsed-time measurements
 - IBM specific
 - Text based
 - Link special library in at compile time
 - L/usr/local/lib -lmpitrace
 - Run the application as usual
 - Information written to different file for each rank: mpi_profile.X
 - Simple and easy to use
 - Awkward to produce collective processor information e.g. averages over processors
-

- **VAMPIR (Visualization and Analysis of MPI Programs)**
 - Displays timeline of code execution and timing statistics
 - Allows communications in an MPI code to be visualised
 - Third party application - PALLAS (<http://www.pallas.com/>)
 - Portable
 - **Vampirtrace (VT) library**
 - A library which produces a tracefile containing information about MPI communications and timings
 - **Vampir**
 - A graphical user interface for visualisation of the tracefile generated by Vampirtrace
-

- Setup environment

```
export PAL_ROOT=/usr/local/packages/vampir/  
export VT_ROOT=/usr/local/packages/vampir/  
export PAL_LICENSEFILE=$PAL_ROOT/etc/license.dat  
export PATH=$PATH:$PAL_ROOT/bin
```

- This must be done in the Loadleveler script as well as in the interactive shell!

- Compilation

```
mpxlf90_r -O3 -qsuffix=f=f90 -o example example.f90  
-L$PAL_ROOT/lib -lVT -lld
```

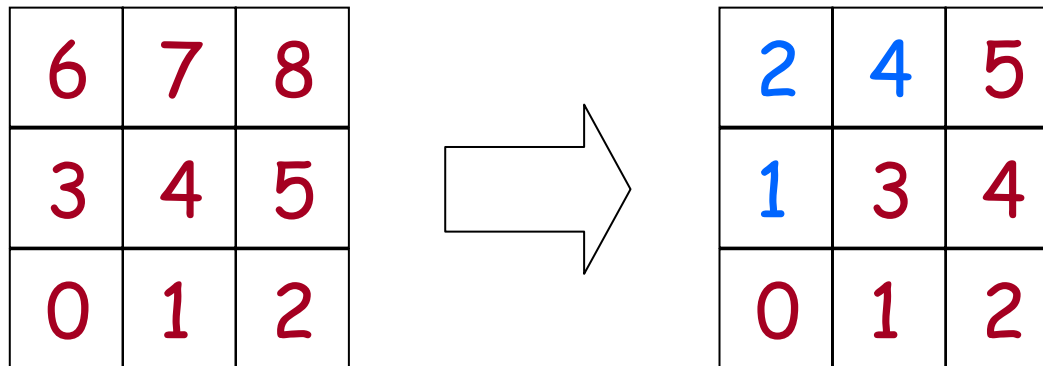
- Execution

```
poe ./example
```

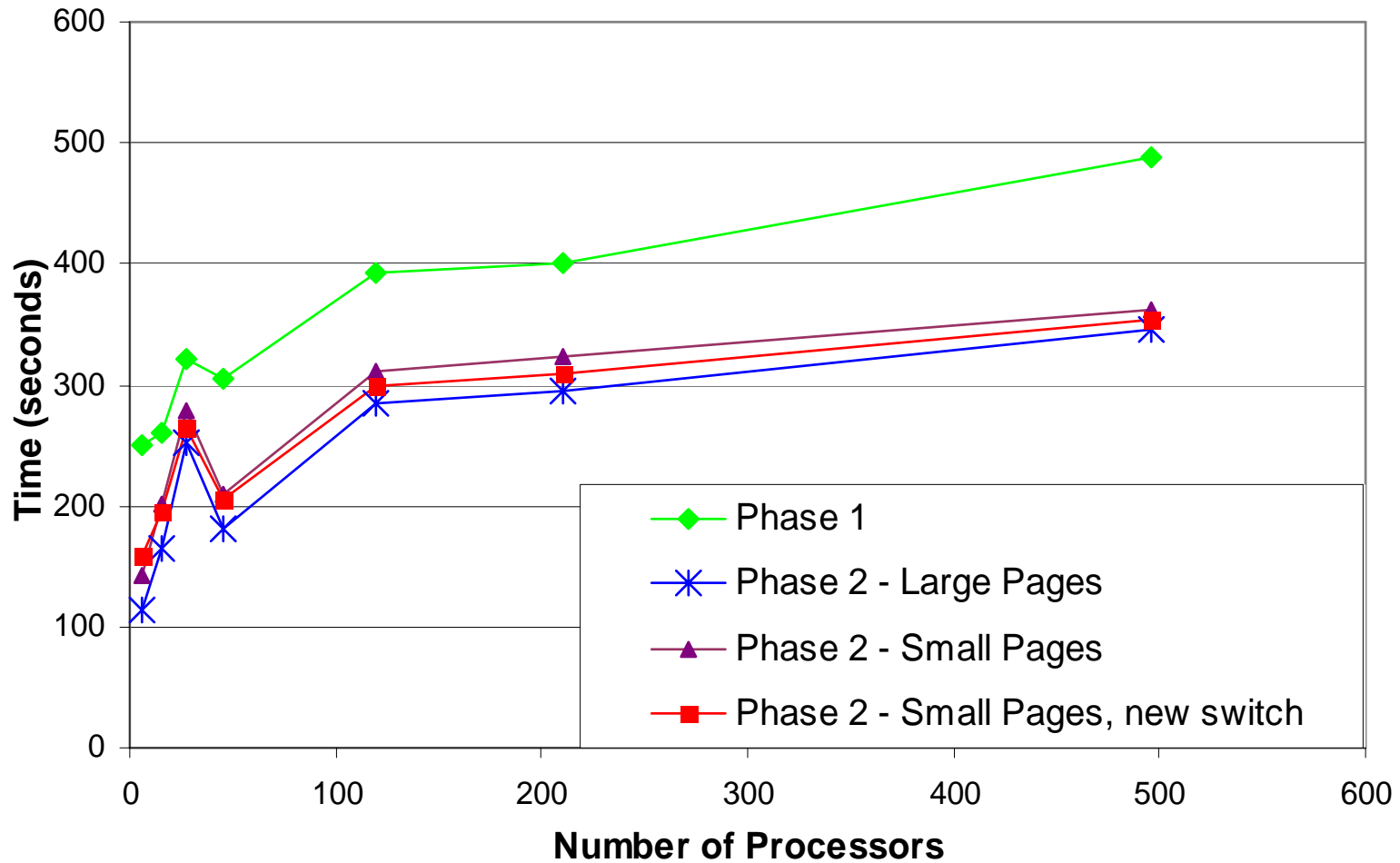
- creates file: example.stf
- View this using: vampir

- **The Code**
 - From the Multiphoton and Electron Collision Consortium (CCP2)
 - Written by Ken Taylor & Daniel Dundas, Queens University Belfast
 - Solves the time-dependent Schrodinger equation
 - **Parallelisation**
 - The Z domain of the grid is distributed amongst an array of processors
 - The code specifies a constant number of grid points per processor in the Z-direction
 - Hence perfect scaling would be represented by a flat timing profile across the different processor counts
-

- Interesting communication pattern due to:
- A series of Bsend and receive operations
 - Overlapped with some computation
- Due to symmetry we need only store the lower half of the grid: needs less processors



- For example - send to the right
 - no periodic boundary conditions
 - 0 -> 1, 1 -> 2, 3 -> 4 and 1 -> 3, 2 -> 4, 4 -> 5



Communication Time - MPITrace



MPI Routine	#calls	avg. bytes	time(sec)
MPI_Comm_size	1	0.0	0.000
MPI_Comm_rank	1	0.0	0.000
MPI_Bsend	1400	2376000.0	1.249
MPI_Recv	840	2376000.0	7.996
MPI_Buffer_attach	1	0.0	0.000
MPI_Bcast	8	3088.0	0.000
MPI_Barrier	437	0.0	2.247
MPI_Allreduce	415	15.5	0.027

total communication time = 11.520 seconds.
total elapsed time = 112.998 seconds.
user cpu time = 108.970 seconds.
system time = 1.050 seconds.
maximum memory size = 803344 KBytes.

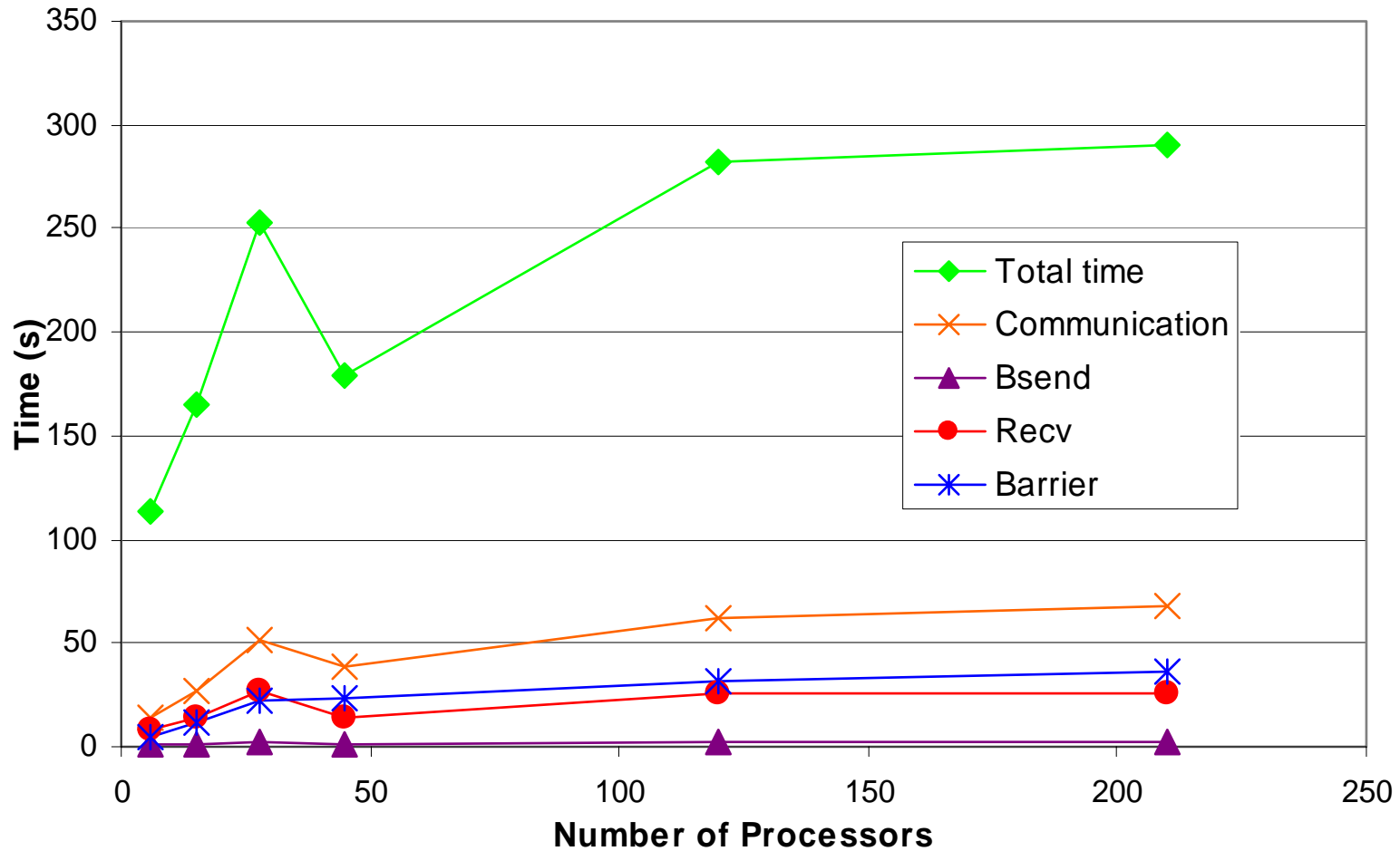
Communication Time - MPITrace



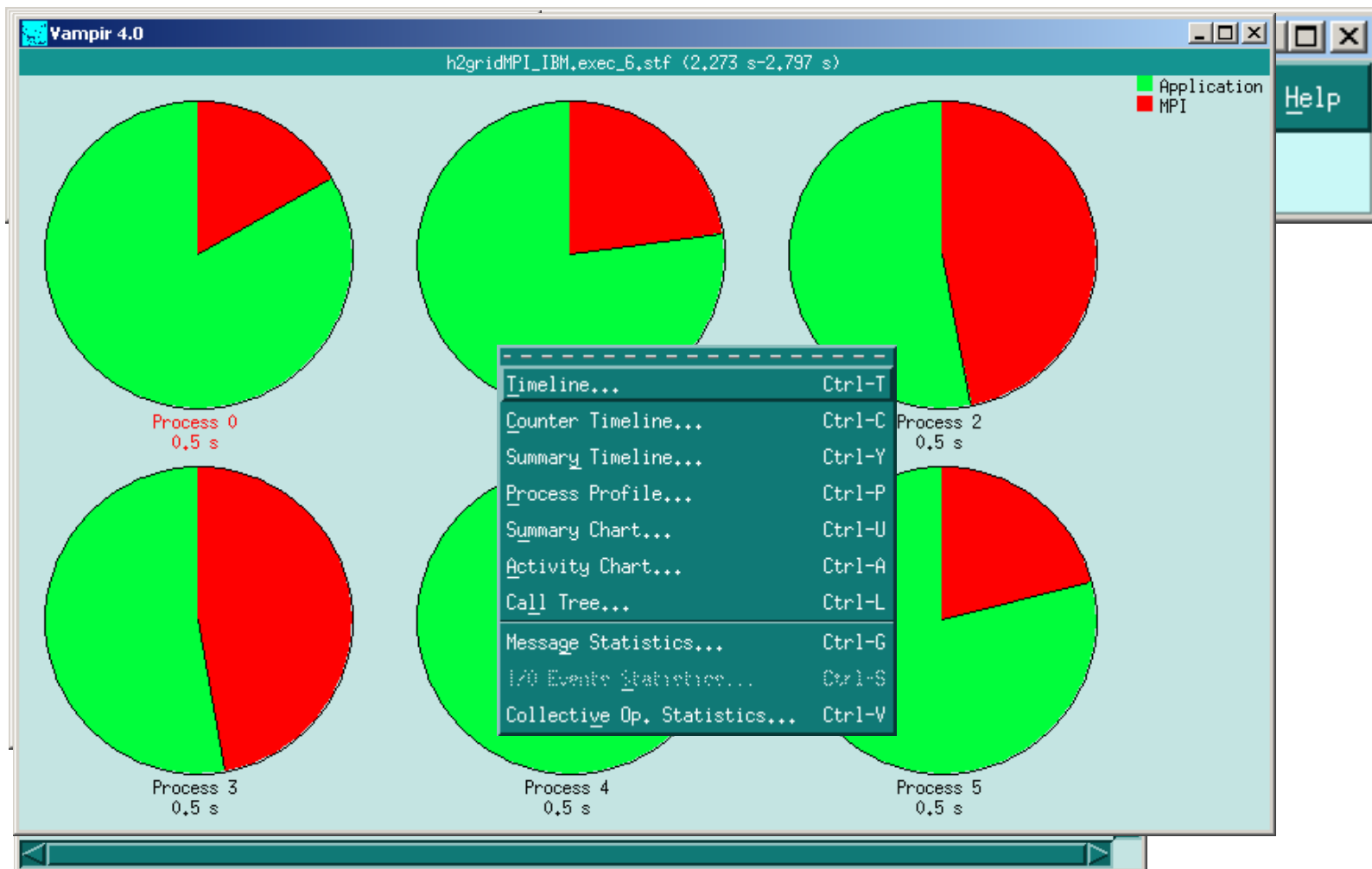
Message size distributions:

MPI_Bsend	#calls	avg. bytes	time(sec)
	1400	2376000.0	1.249
MPI_Recv	#calls	avg. bytes	time(sec)
	840	2376000.0	7.996
MPI_Bcast	#calls	avg. bytes	time(sec)
	1	88.0	0.000
	3	240.0	0.000
	1	360.0	0.000
	1	1936.0	0.000
	1	7200.0	0.000
	1	14400.0	0.000
MPI_Allreduce	#calls	avg. bytes	time(sec)
	24	8.0	0.001
	391	16.0	0.026

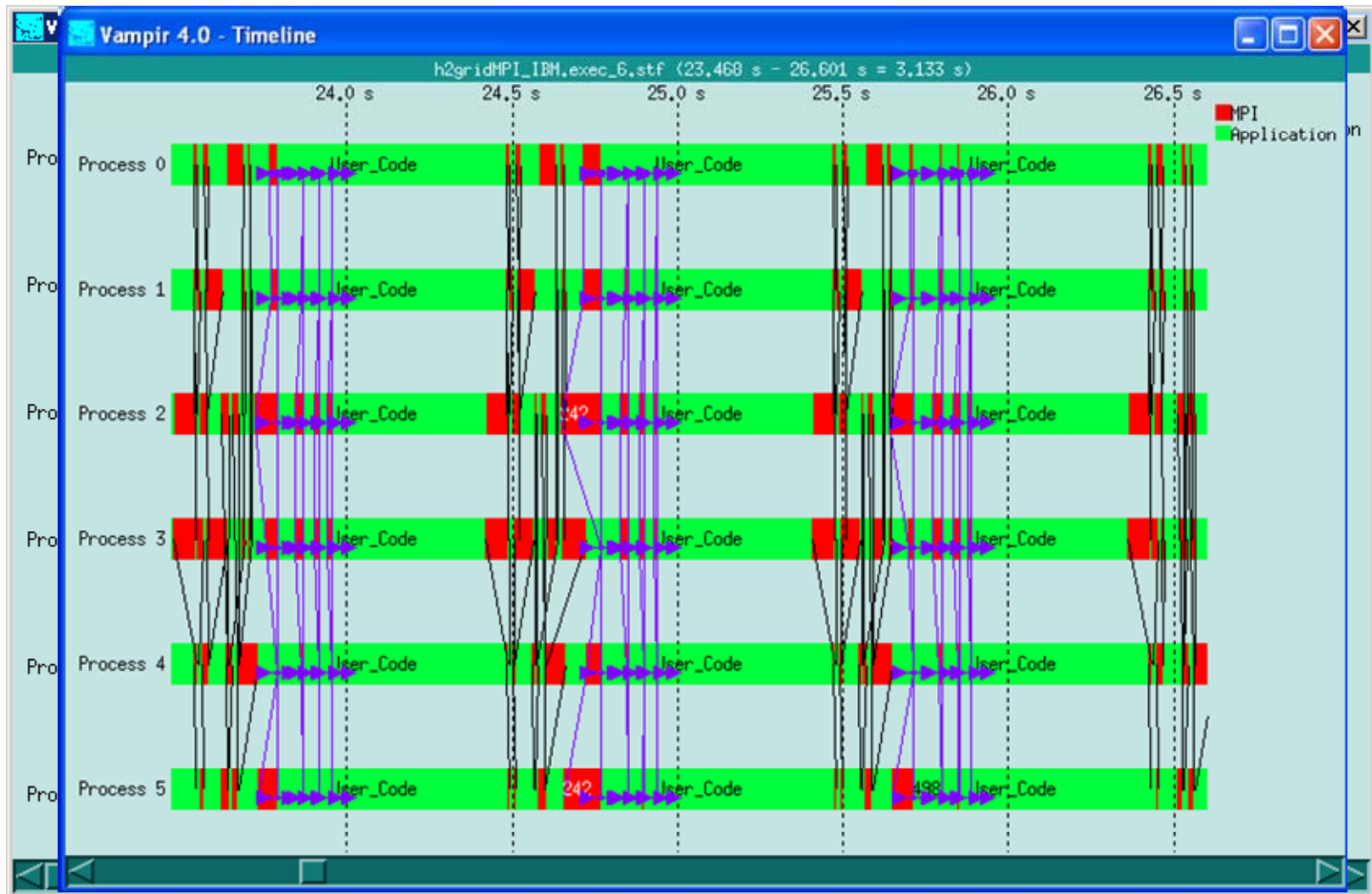
Communication Time - MPITrace



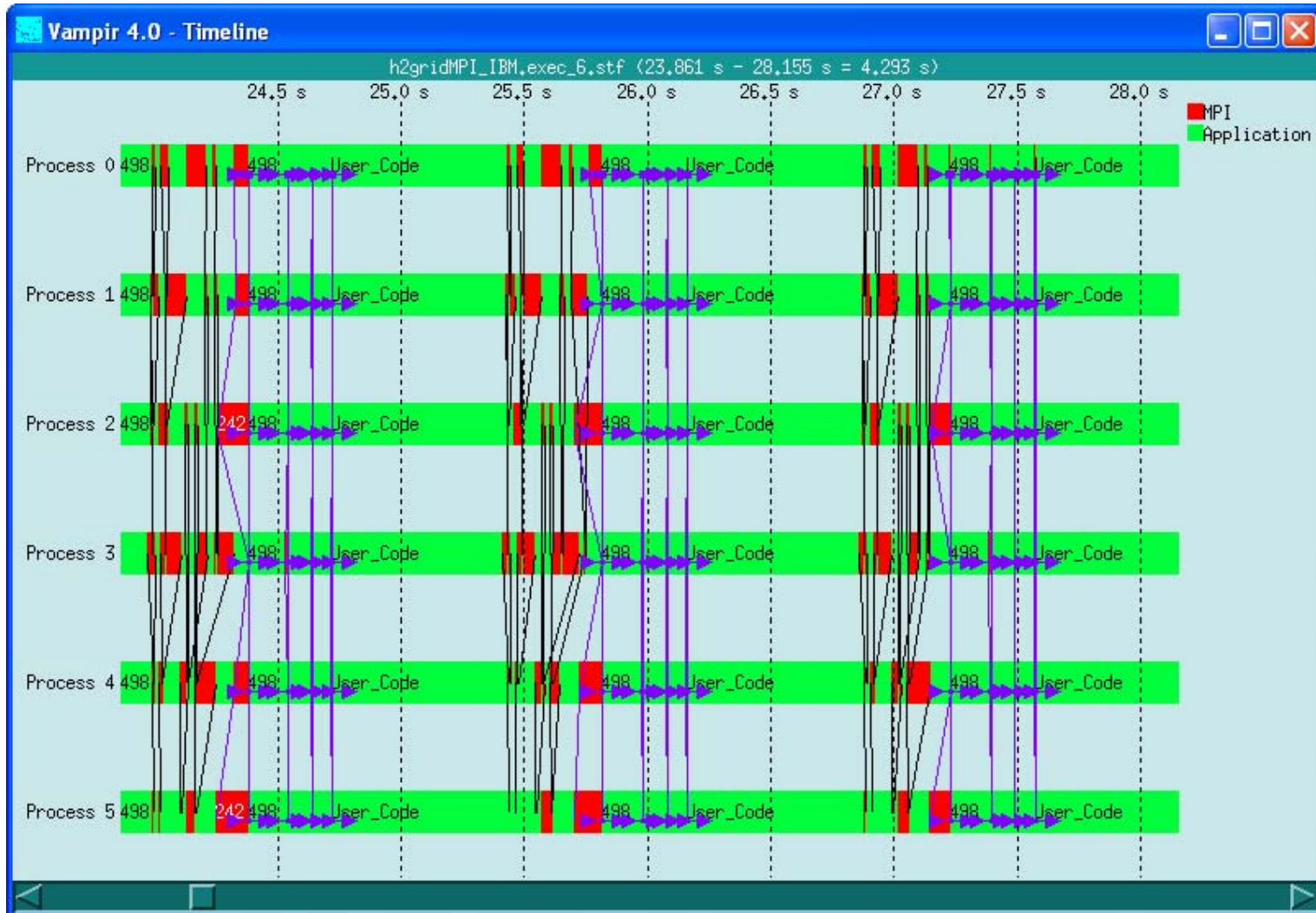
Communication Time - Vampir



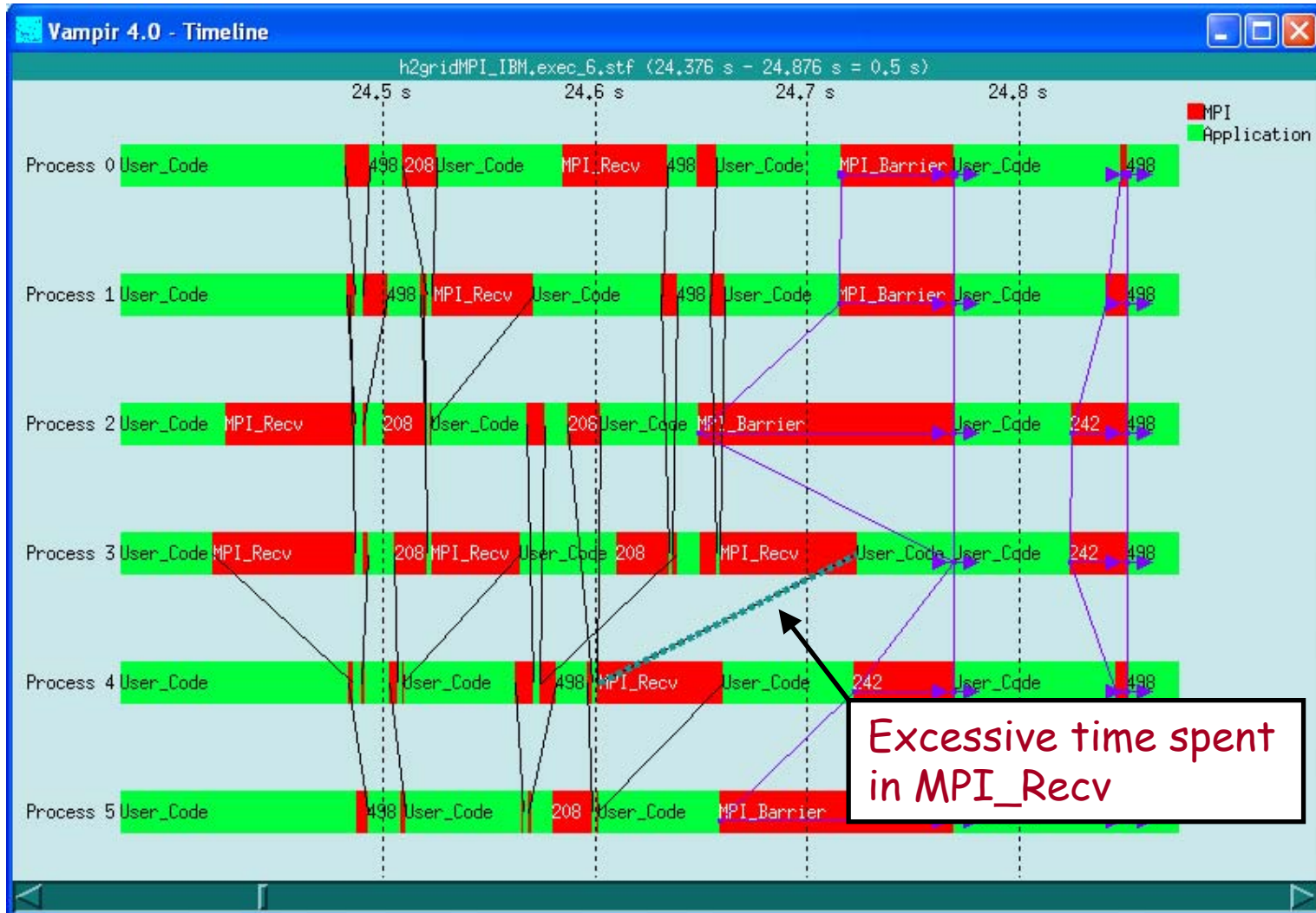
Communication Time - Load Imbalance



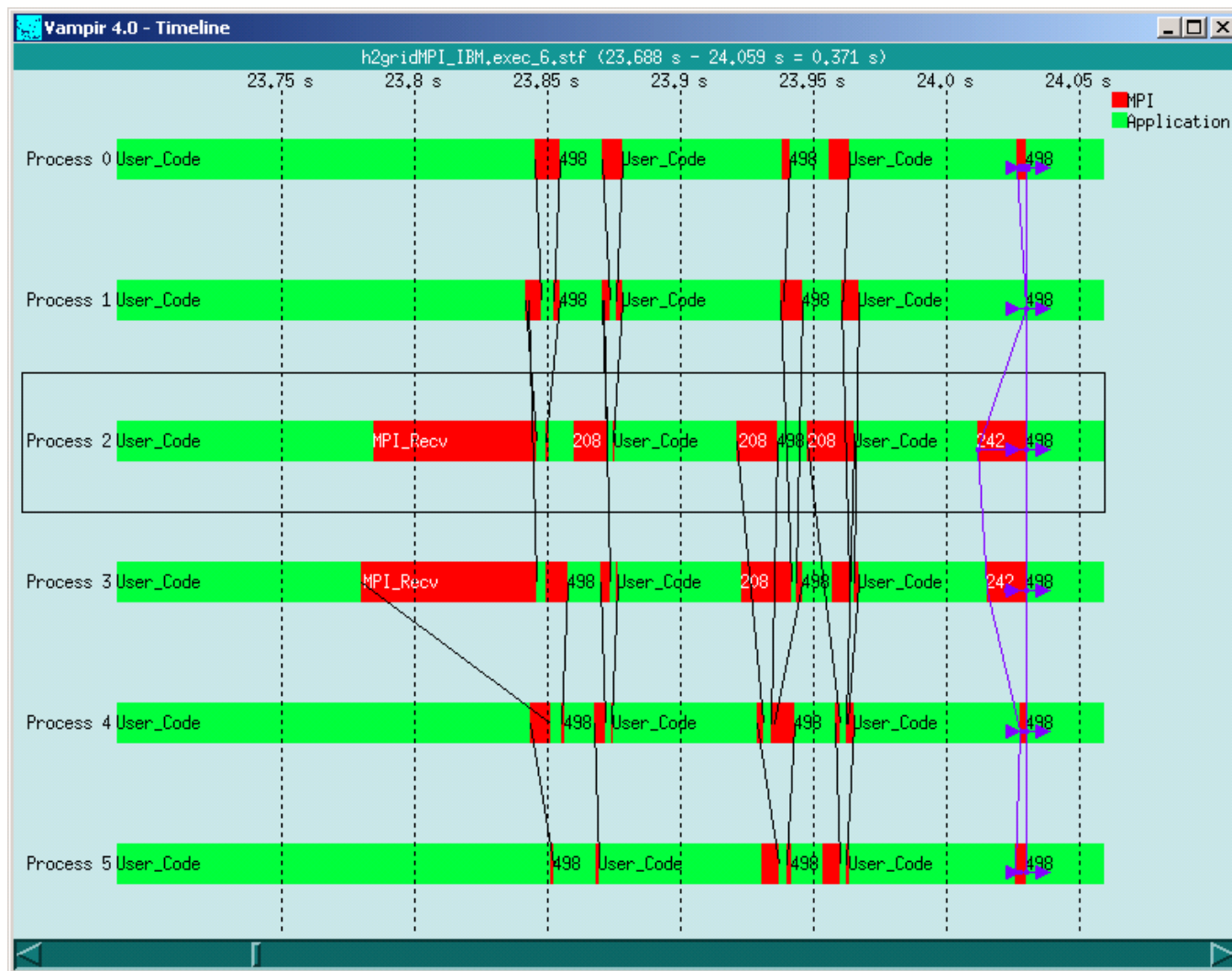
Communication Time - Load Imbalance



Communication Time - MPI_Bsend



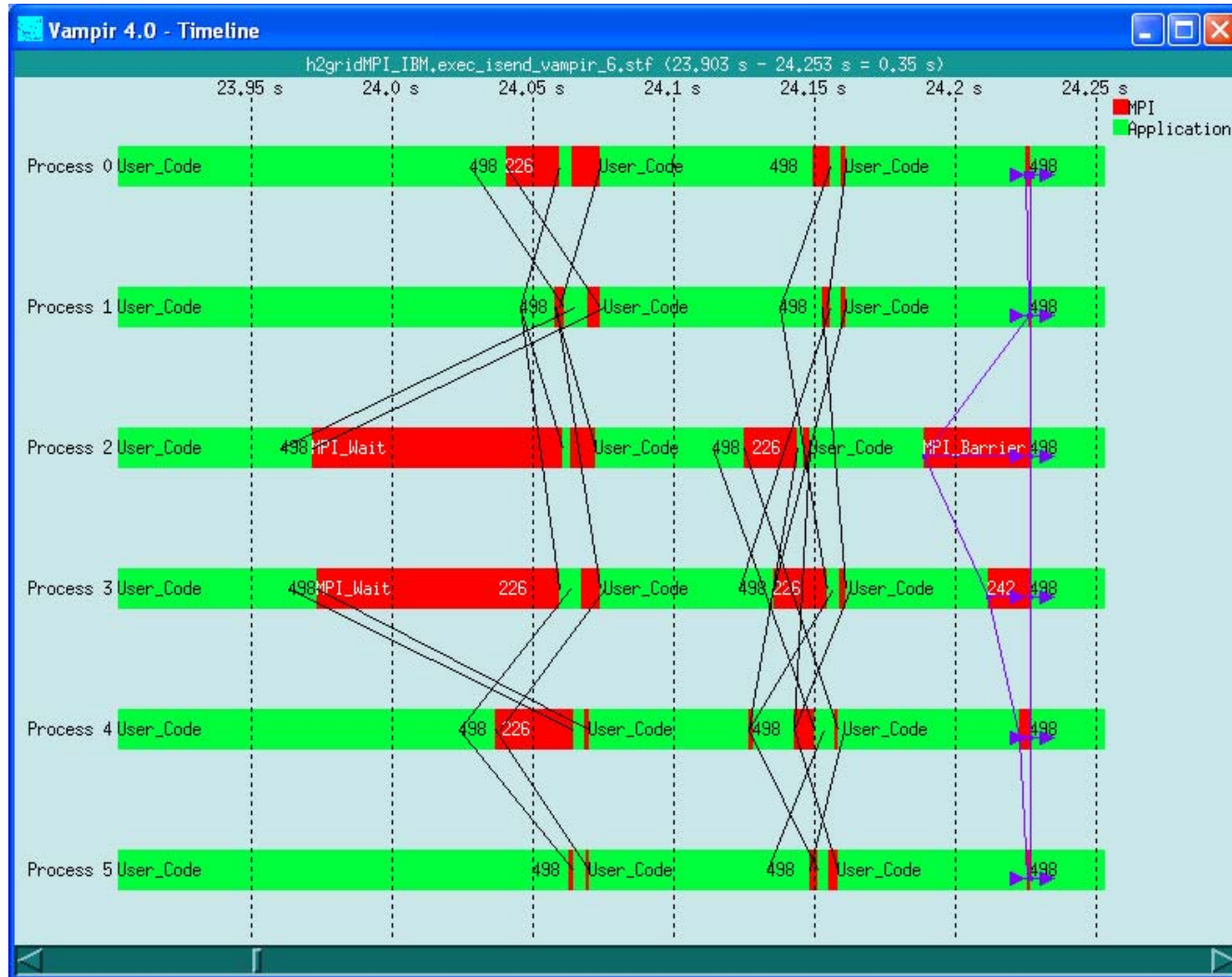
Communication Time - MPI_Bsend, MP_CSS INTERRUPT



Communication Time - MPI_Bsend, MP_CSS INTERRUPT, 120 processes



Communication Time - MPI_Isend



Communication Time

Operation	Time (s)	
	Proc = 6	Proc = 120
MPI_Bsend, MPI_Recv	144	299
MPI_Bsend, MPI_Recv, MP_CSS_INTERRUPT	135	295
MPI_Send, MPI_Recv	135	303
MPI_Send, MPI_Recv, MP_CSS_INTERRUPT	134	303
MPI_Isend, MPI_Irecv	131	280
MPI_Isend, MPI_Irecv, MP_CSS_INTERRUPT	133	280

- Load imbalance due to processor allocation on MCMs
 - MPI_BSend results in excessive amounts of time spent in MPI_Recv
 - MP_CSS_INTERRUPT fixes this problem on small processor numbers
 - Using MPI_IRecv and MPI_Isend allows overlapping of communications
 - Without excessive time spent in the receive call
 - Shows slight performance improvement
-

- **MPITrace**
 - Demonstrates scaling issues are due to communication
 - Highlights MPIRecv and MPIBarrier as most expensive routines
 - **VAMPIR**
 - Provides detailed view of individual communications
 - Shows complexity of communication patterns in a highly visual way
-

- Performance analysis tools
 - Help identify bottlenecks for optimisation
 - HPCx has a range of tools to help you analyse the performance of your code
 - Timers, xprofiler, hpmcount
 - Vampir, MPITrace
 - For constellation systems
 - Vampir and MPITrace help identify communication issues
 - E.g. slower MPI operations between nodes
-