

Tools



- Debuggers
 - dbx and pdbx
- Profilers
 - gprof, xprofiler
 - mpitrace
 - Vampir
- Hardware Performance Monitor Toolkit
 - Counter Library (hpmcount)
 - libhpm and hpmviz
- Timers
- Conclusions

- Text based tool for debugging sequential executables and core files
- For Fortran, C and C++
- Compile with `-g` flag
- **Functionality**
 - execute code within dbx
 - set and remove breakpoints
 - display values of variables and expressions
 - display and edit the source file
 - list current file and function

```
xlf90_r -qsuffix=f=f90 -g -o example example.f90
```

```
./example
```

```
Trace/BPT trap(coredump)
```

```
dbx ./example ./core
```

```
Type 'help' for help.
```

```
reading symbolic information ...
```

```
[using memory image in ./core]
```

```
Trace/BPT trap in example at line 9
```

```
    9      z = x/y
```

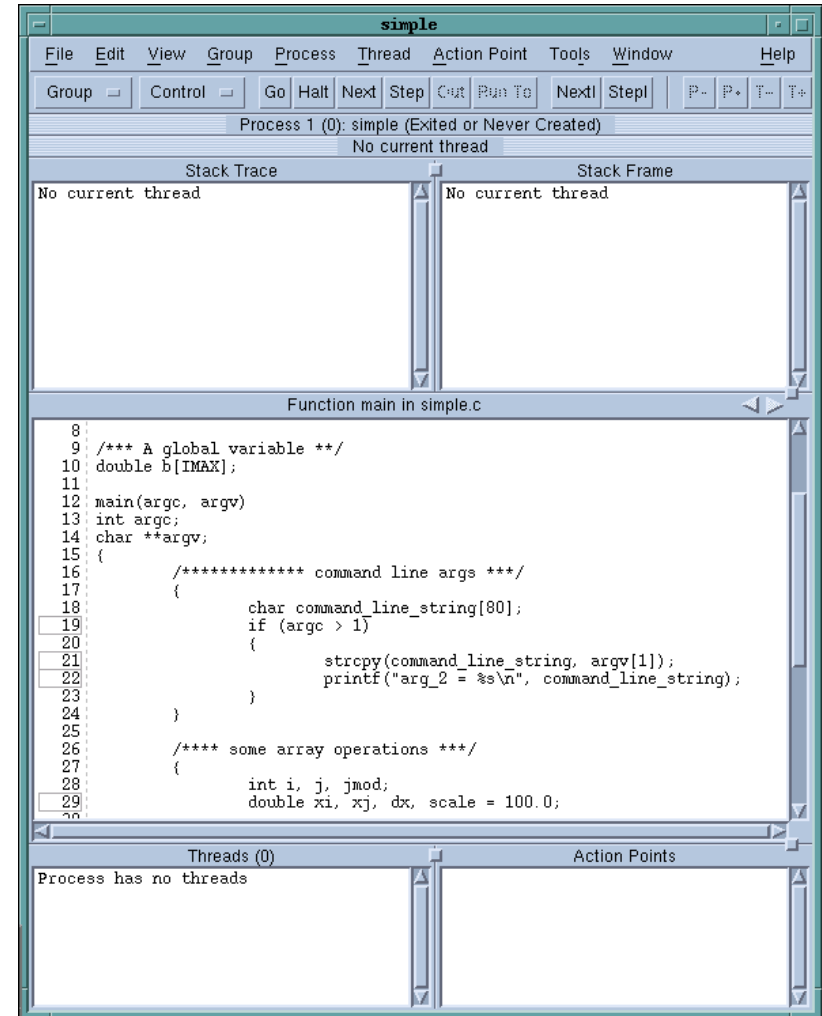
```
(dbx) print y
```

```
0
```

- Text based parallel debugger for core files and executables
- Compile with `-g`
- For Fortran, C and C++
 - supports most of the dbx debugger subcommands
 - can execute code within pdbx
 - can attach pdbx to a code while running
 - can debug core files with pdbx
- Example

```
pdbx ./prog.exe -llfile ./llscriptfile -procs N
```
- Runs as interactive parallel job
 - **N** must match number of CPUs requested

- Standard parallel debugger
- Some slightly messy set-up required: see www.hpcx.ac.uk/support/FAQ/totalview
- To start totalview, type `/usr/local/packages/totalview/tv6`



- **prof and gprof**
 - standard unix commands provide simple profiling at the subroutine level

- **Example**

```
mpxlf90 -qsuffix=f=f90 -pg -o example example.f90  
poe -nprocs 3 example
```

- creates a series of files: `gmon.out.taskid`

```
gprof example gmon.out.0 gmon.out.1 gmon.out.2
```

- **Provides CPU (busy) usage only**
 - no I/O or communication analysed

- Simple profiler for both serial and parallel applications
- GUI alternative to gprof
- Only profiles CPU (busy) usage
 - lack of I/O and communication information
- Profiles application at the source statement level
- Example

```
mpxlf90_r -qsuffix=f=f90 -pg -o example example.f90  
... run the code ...  
xprofiler example gmon.out.0 gmon.out.1 gmon.out.2
```

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 overlaid on the graph, listing functions such as '.forces', '6.40 self', and 'called 50'. A 'Statistics Report' window is also visible, showing 'Function Level Statistics Report'. The 'Source Code for forces.f' window is open, displaying the following code:

```
File      Utility
line     NO. TICKS  source code
1         1         subroutine forces(np, x, f, vir, epot, side, rcoeff)
2         2         implicit double precision (a-h, o-z)
3         3         dimension x(np, 3), f(np, 3)
4         4
5         5         c
6         6         compute forces and accumulate the virial and potential.
7         7         c
8         8         !$OMP SINGLE
9         9         vir = 0.0d0
10        10        epot = 0.0d0
11        11        !$OMP END SINGLE
12        12        sideh = 0.5d0*side
13        13        rcoeffs = rcoeff*rcoeff
14        14        c
15        15        !$OMP DO PRIVATE(j, xi, yi, zi, fxi, fyi, fzi, xx, yy,
16        16        !$OMP& zz, rd, rrd, rrd2, rrd3, rrd4, rrd6, rrd7, r148, forcex, forcey, forcez),
17        17        !$OMP& REDUCTION(-:vir), REDUCTION(+:epot),
18        18        !$OMP& SCHEDULE (STATIC, 16)
19        19        do 270 i = 1, np
20        20        xi = x(i, 1)
                yi = x(i, 2)
```

At the bottom of the interface, there is a search engine section with the text 'Search Engine: (regular expressions supported)' and a search box containing the text 'forces'.

- Can use the Filter menu to
 - Uncluster Functions (Initially routines are grouped by library.
 - Hide library calls
- The view assembly code option is more useful than it sounds as it also shows the corresponding source code.
 - If used with -g can also see time spend in each line of code.
- At high levels of optimisation function are inlined into their parent and disappear from the profile

- **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

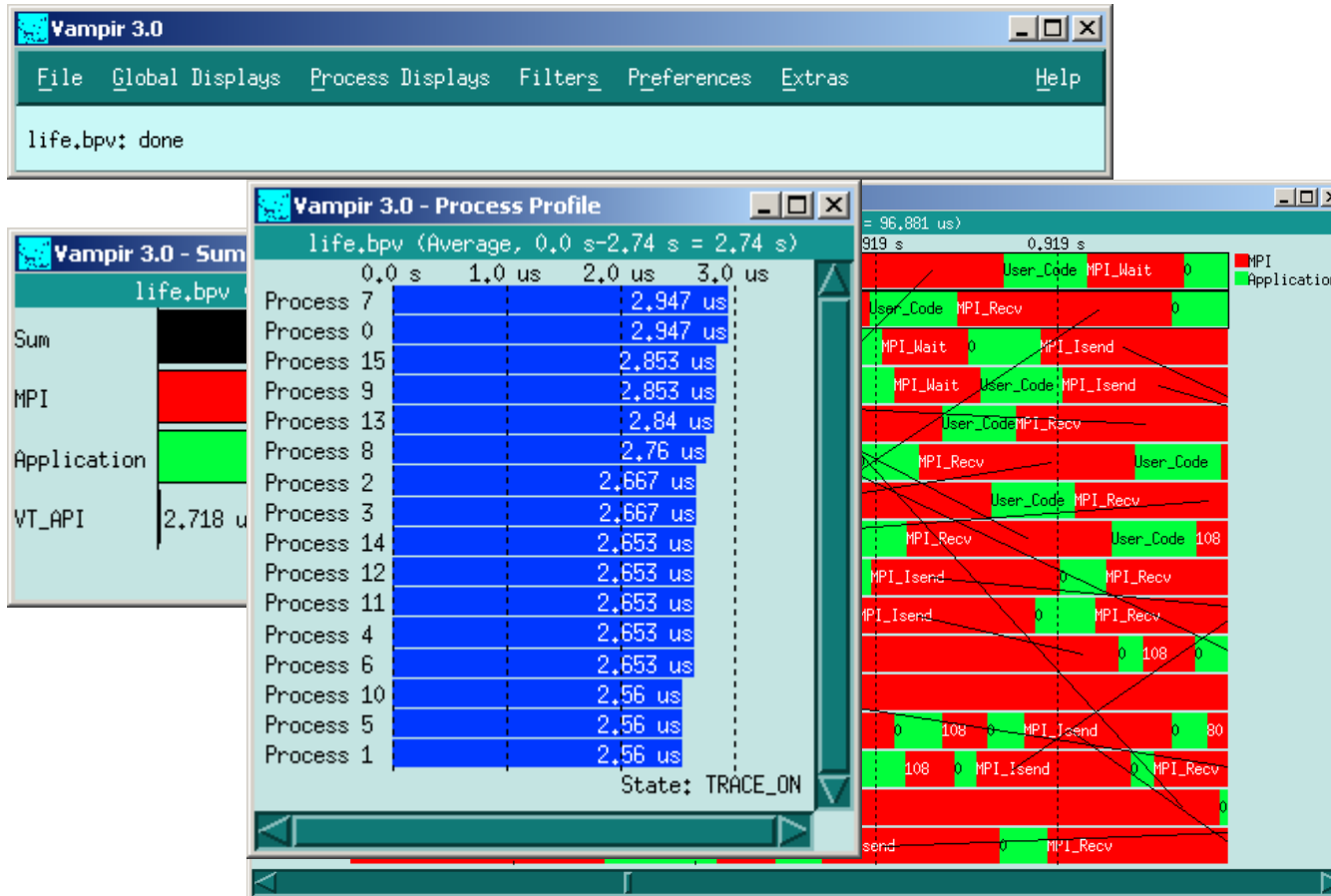
```
mpxlf90_r -O3 -qsuffix=f=f90 -o example example.f90
-I$PAL_ROOT/include -L$PAL_ROOT/lib -lVT -lm -lld
```

- Execution

```
poe ./example -nprocs 16
```

- creates file: `example.stf`
- View this using: `vampir example.stf` then load full trace using the LOAD menu

Profiling - Vampir Example



- 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`

```
-----  
MPI Routine                #calls      avg. bytes      time(sec)  
-----  
MPI_Comm_size              1            0.0             0.000  
MPI_Comm_rank              1            0.0             0.000  
MPI_Sendrecv               6526         1024.0          0.061  
MPI_Gather                  1            98304.0         0.000  
MPI_Scatter                 1            98304.0         0.001  
MPI_Allreduce               3263         8.0             0.068  
-----
```

```
total communication time = 0.131 seconds.  
total elapsed time       = 1.220 seconds.  
user cpu time            = 1.160 seconds.  
system time              = 0.000 seconds.  
maximum memory size     = 6940 KBytes.
```

...

- Toolkit for performance measurement of applications on Power 3 and 4 architectures
- **hpmcount**
 - utility which reports performance information
- **libhpm**
 - instrumentation library which reports performance information for instrumented code
- **hpmviz**
 - A graphical user interface for visualisation of the performance file generated by libhpm

- Supports serial, parallel (MPI , threaded, mixed-mode) applications in C,C++ and Fortran
- Add `/usr/local/packages/actc/hpmtk/pwr4/bin` to your path
- Serial Example
`hpmcount example`
- Select groups of counters, eg for L2/L3 misses
`hpmcount -g 5 example`
- Parallel Example, place the following within a LoadLeveler script
`poe hpmcount -o output_file example`
- No code re-compilation required

- **hpmcount provides performance details**
 - execution wall clock time
 - hardware performance counter information (e.g. cache misses)
 - derived hardware metrics (e.g. load per cache miss)
 - resource utilisation statistics (e.g. no of page faults)
- **Highlights**
 - Float point instructions + FMA rate (Mflip/s)
 - PM_DTLB_MISSES (Data Translation Lookaside Buffer (TLB) misses)
 - Avg number of loads per TLB miss
 - Computation intensity
 - ratio of load and store operations to floating point operations

- Sparse Matrix Multiplication:

```
do i=1,n
  do k=1,n
    do j= 1,n
      matrix3(i,j) = matrix3(i,j) +
matrix1(i,k)*matrix2(k,j)
    enddo
  enddo
enddo
```

- Float point instructions + FMA rate: 30.5 Mflip/s
 - PM_DTLB_MISSES (Data TLB misses): 1891682624
 - Avg number of loads per TLB miss: 1.0
 - Total execution time (wall clock time): 3.01 seconds
-

- Reorder loop indices:

```
do j=1,n
  do k=1,n
    do i= 1,n
      matrix3(i,j) = matrix3(i,j) +
        matrix1(i,k)*matrix2(k,j)
    enddo
  enddo
enddo
```

- Float point instructions + FMA rate: 663.8 Mflip/s
 - PM_DTLB_MISSES (Data TLB misses): 4971692
 - Avg number of loads per TLB miss: 403
 - Total execution time (wall clock time): 2.99 seconds
-

- Replace this with a function from the IBM Engineering and Scientific Subroutine Library (ESSL):

```
call DGEMM('n','n',n,n,n,1.0d0,matrix1,n, &  
matrix2,n,0.0d0,matrix3,n)
```

- DGEMM (double precision general matrix-matrix multiplication)
- Float point instructions + FMA rate: 2375.8 Mflip/s
- PM_DTLB_MISS (Data TLB misses): 3057304
- Avg number of loads per TLB miss: 507
- Total execution time (wall clock time): 0.84 seconds

- Provides performance information for instrumented regions of code
- Supports serial, parallel (MPI, threaded, mixed-mode) applications in C, C++ and Fortran
- Collects information + summarisation during run-time
 - possible overhead, especially within loops
- Creates two output files
 - one contains readable performance data
 - one created for use with hpmviz

```
#include <f_hpm.h>
call f_hpminit(0, 'matmult3')
...
call f_hpmstart(1, 'matrix multiplication')
do j=1,n
  do k=1,n
    do i= 1,n
      matrix3(i,j) = matrix3(i,j) +
        matrix1(i,k)*matrix2(k,j)
    enddo
  enddo
enddo
call f_hpmstop(1)
...
call f_hpmterminate(0)
```

- **Compilation**

```
xlf90_r -qsuffix=cpp=f90 -O4 -o matmult3  
matmult3.f90 -I<HPM-DIR>/include -L<HPM-  
DIR>/lib -lhpm -lpmpi -lm
```

- **Execution**

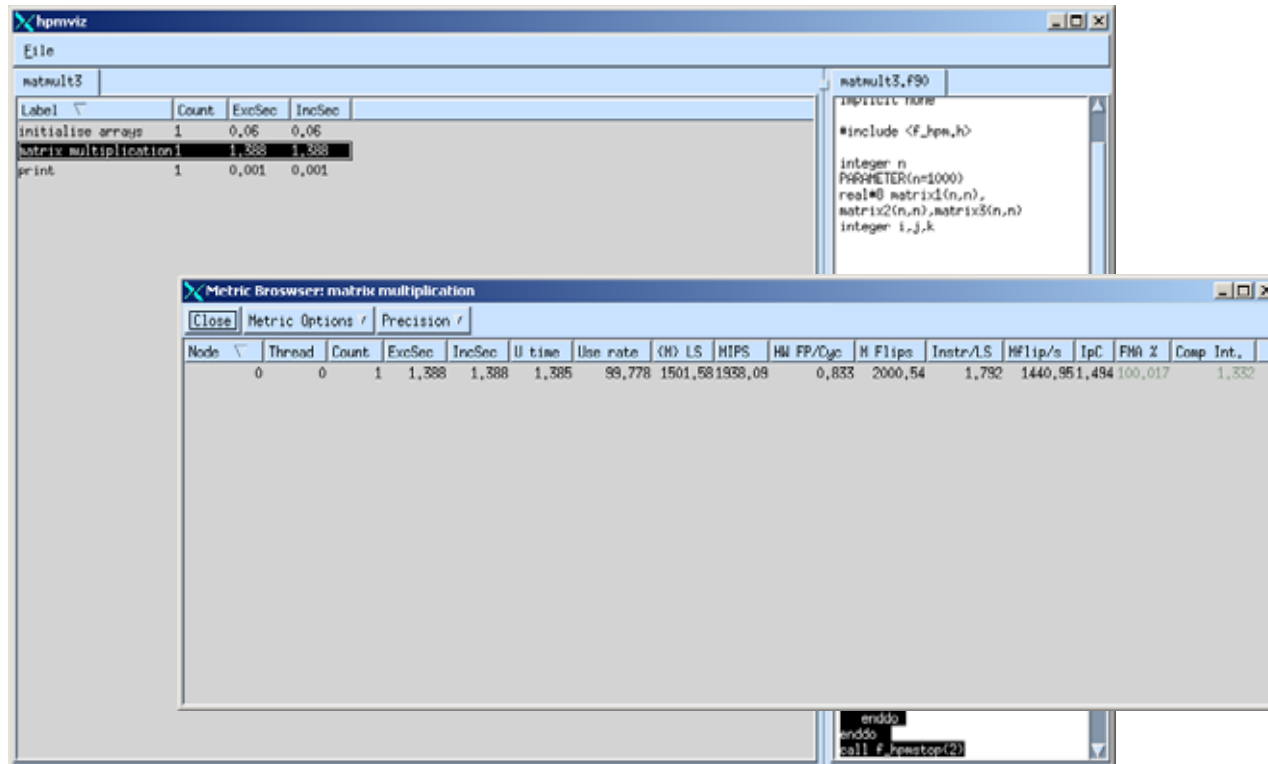
- produces two files

- perfhpm0000.<pid> - provides similar information to hpmcount, for each instrumented section
- hpm0000_matmult3_<pid>.viz - can be displayed using hpmviz

- **hpmviz**

- A graphical user interface for visualisation of the performance file generated by libhpm

```
hpmviz hpm0000_matmult3_<pid>.viz
```



The screenshot shows the hpmviz application interface. The main window displays a code editor with the following code:

```
matmult3
#practic none
#include <F_hpm.h>
integer n
PARAMETER(n=1000)
real*8 matrix1(n,n),
matrix2(n,n),matrix3(n,n)
integer i,j,k

integer n
PARAMETER(n=1000)
real*8 matrix1(n,n),
matrix2(n,n),matrix3(n,n)
integer i,j,k
```

The main window also shows a table with the following data:

Label	Count	ExcSec	IncSec
initialise arrays	1	0,06	0,06
matrix multiplication	1,388	1,388	1,388
print	1	0,001	0,001

A "Metric Browser: matrix multiplication" window is open, displaying a table with the following data:

Node	Thread	Count	ExcSec	IncSec	U time	Use rate	(H) LS	MIPS	HW FP/Cyc	M Flips	Instr/LS	HWFlp/s	IpC	FMA 2	Comp Int.
0	0	1	1,388	1,388	1,385	99,778	1501,581938,09	0,833	2000,54	1,792	1440,961,494	100,017	1,332		

The bottom right corner of the code editor shows the following code:

```
enddo
enddo
call F_hpmstop(2)
```

- Wide range of timers available on the HPCx system
- Varying
 - precision
 - portability
 - language
 - ease of use

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
system_clock(*)	subroutine	wallclock	microsecond	F	yes

- `system_clock`
 - portable, Fortran only
 - Compile with `-qsclk=micro` for increased resolution
- `gettimeofday`
 - portable, C/C++ only
- `irtc`
 - nanosecond precision
 - non portable, Fortran only
- `read_real_time`
 - nanosecond precision
 - non portable, C/C++ only
- Also `MPI_WTime`
 - portable, Fortran and C/C++

```
integer :: clock0,clock1,clockmax,clockrate,ticks
real    :: secs
call system_clock(count_max=clockmax,
                  count_rate=clockrate)

call system_clock(clock0)
! code to be timed
call system_clock(clock1)

ticks = clock1-clock0
! reset negative numbers
ticks = mod(ticks+clockmax, clockmax)
secs = float(ticks)/float(clockrate)

print*, 'Total time ', secs, ' seconds'
```

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

struct timeval start_time, end_time;
main() {
    int total;
    gettimeofday(&start_time, (struct timeval*)0);

    /* code to be timed */

    gettimeofday(&end_time, (struct timeval*)0);

    total = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
            (end_time.tv_usec-start_time.tv_usec);

    printf("Total time %d microseconds \n", total);
}
```

- May have used irtc previously on the Cray

```
integer(8) :: start, end, irtc
```

```
start = irtc()
```

```
! code to be timed
```

```
end = irtc()
```

```
print*, ' Total time ', end - start, 'nanoseconds'
```

```
timebasestruct_t start, end;
int seconds, n_seconds;

read_real_time(&start, TIMEBASE_SZ);

/* code to be timed */

read_real_time(&end, TIMEBASE_SZ);

/* Ensure values are in seconds and nanoseconds */
time_base_to_time(&start, TIMEBASE_SZ);
time_base_to_time(&end, TIMEBASE_SZ);
```

```
/* Subtract starting time from ending time */

seconds = end.tb_high - start.tb_high;
n_seconds = end.tb_low - start.tb_low;

/* Fix carry from low to high order */

if (n_seconds < 0) {
    seconds--;
    n_seconds += 1000000000;
}

printf("Total time %d seconds %d nanoseconds\n",
seconds, n_seconds);
```

- Considered:
- Debuggers
- Profilers
- Hardware Performance Monitor
- Timers

- IBM Parallel Environment for AIX Manuals: Operation and Use Volume 1 and Volume 2
 - http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/pe.html
- The HPCx Service User Guide
 - <http://www.hpcx.ac.uk/support/documentation/>