Profiling and debugging

Yaakoub El Khamra

Texas Advanced Computing Center
Outline

Debugging
- GDB
  - Basic use
  - Attaching to a running job
- DDT
  - Identify MPI problems using Message Queues
  - Catch memory errors
- PTP
  - For the extremely patient*

Profiling
- Command line timing
- gprof
- Advanced Tools
  - Perfexpert
  - IPM
  - Tau (and PAPI)
  - HPCToolkit
- Lessons
- Promising packages
Debugging

gdb and ddt
Why use a debugger?

• You’ve got code -> you’ve got bugs

• Buffered output (printf / write may not help)

• Fast & Accurate

• Many errors are difficult to find without one!
GDB is the GNU Project DeBugger
www.gnu.org/software/gdb/

From the GDB website: GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
Using GDB

Compile with debug flags: `gcc -g -O0 ./srcFile.c`

The `-g` flag generates the symbol table and provides the debugger with line-by-line information about the source code.

Execute debugger loading source dir: `gdb -d srcDir ./exeFile`

The `-d` option is useful when source and executable reside in different directories.

Use the `-q` option to skip the licensing message.

Type `help` at any time to see a list of the debugger options and commands.
Two levels of control

• Basic:
  – Run the code and wait for it to crash.
  – Identify line where it crashes.
  – With luck the problem is obvious.

• Advanced:
  – Set breakpoints
  – Analyze data at breakpoints
  – Watch specific variables
## GDB basic commands

<table>
<thead>
<tr>
<th>command</th>
<th>shorthand</th>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run/kill</td>
<td>r / k</td>
<td>NA</td>
<td>start/end program being debugged</td>
</tr>
<tr>
<td>continue</td>
<td>c</td>
<td>NA</td>
<td>continue running program from last breakpoint</td>
</tr>
<tr>
<td>step</td>
<td>s</td>
<td>NA</td>
<td>take a single step in the program from the last position</td>
</tr>
<tr>
<td>where</td>
<td>NA</td>
<td>NA</td>
<td>equivalent to backtrace</td>
</tr>
<tr>
<td>print</td>
<td>p</td>
<td>variableName</td>
<td>show value of a variable</td>
</tr>
<tr>
<td>list</td>
<td>l</td>
<td>srcFile.c:lineNumber</td>
<td>show the specified source code line</td>
</tr>
<tr>
<td>break</td>
<td>b</td>
<td>srcFile.c:lineNumber functionName</td>
<td>set a breakpoint by line number or function name</td>
</tr>
<tr>
<td>watch</td>
<td>NA</td>
<td>variableName</td>
<td>stops when the variable changes value</td>
</tr>
</tbody>
</table>
GDB example

divcrash.c

#include <stdio.h>
#include <stdlib.h>
int myDiv(int, int);

int main(void)
{
    int res, x = 5, y;

    for(y = 1; y < 10; y++){
        res = myDiv(x,y);
        printf("%d,%d,%d\n",x,y,res);
    }
    return 0;
}

int myDiv(int x, int y){
    return 1/( x - y);
}

divcrash.f90

PROGRAM main

INTEGER :: myDiv
INTEGER :: res, x = 5, y

DO y = 1, 10
    res = myDiv(x,y)
    WRITE(*,*) x,y,res
END DO

END PROGRAM

FUNCTION myDiv(x,y)
   INTEGER, INTENT(IN) :: x, y
   myDiv = 1/(x-y)
   RETURN
END FUNCTION myDiv
GDB example

Compile the program and start the debugger:

% gcc -g -O0 ./divcrash.c
% gdb ./a.out

Start the program:

(gdb) run

The debugger will stop program execution with the following message:

Program received signal SIGFPE, Arithmetic exception.
0x0000000000040051e in myDiv (x=5, y=5) at divcrash.c:28
28 return 1/( x - y);

We can use gdb commands to obtain more information about the problem:

(gdb) where
#0 0x0000000000040051e in myDiv (x=5, y=5) at divcrash.c:28
#1 0x000000000004004cf in main () at divcrash.c:19
GDB example

In this case the problem is clear: a divide-by-zero exception happens in line 28 when variables \( x \) and \( y \) are the same.

This is related to the call to \textit{myDiv} from line 19 that is within a for loop:

\begin{verbatim}
18:   for(y = 1; y < 10; y++){
19:     res = myDiv(x,y);
\end{verbatim}

Eventually the loop sets the value of \( y \) equal to 5 (the value of \( x \)) producing the exception:

\begin{verbatim}
28:   return 1/( x - y);
\end{verbatim}

With the problem identified we can kill the program and exit the debugger:

\begin{verbatim}
(gdb) kill
(gdb) quit
\end{verbatim}
Examining data

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gdb) p x</td>
<td>(gdb) p x</td>
<td>Print scalar data x value</td>
</tr>
<tr>
<td>(gdb) p V</td>
<td>(gdb) p V</td>
<td>Print all vector V components</td>
</tr>
<tr>
<td>(gdb) p V[i]</td>
<td>(gdb) p V(i)</td>
<td>Print element i of vector V</td>
</tr>
<tr>
<td>(gdb) p V[i]@n</td>
<td>(gdb) p V(i)@n</td>
<td>Print n consecutive elements starting with V_i</td>
</tr>
<tr>
<td>(gdb) p M</td>
<td>(gdb) p M</td>
<td>Print all matrix M elements</td>
</tr>
<tr>
<td>(gdb) p M[i]</td>
<td>Not Available</td>
<td>Print row i of matrix M</td>
</tr>
<tr>
<td>(gdb) p M[i]@n</td>
<td>Not Available</td>
<td>Print n consecutive rows starting with row i</td>
</tr>
<tr>
<td>(gdb) p M[i][j]</td>
<td>(gdb) p M(i,j)</td>
<td>Print matrix element Mij</td>
</tr>
<tr>
<td>(gdb) p M[i][j]@n</td>
<td>(gdb) p M(i,j)@n</td>
<td>Print n consecutive elements starting with Mij</td>
</tr>
</tbody>
</table>

- No simple way to print columns in C or rows in Fortran
- Some debuggers print array slices (pgdbg, dbx), i.e. \texttt{p M(1:3,3:7)}
Breakpoint control

- Stop the execution of the program
- Allow you to examine the execution state in detail
- Can be assigned to a line or function
- Can be set conditionally

<table>
<thead>
<tr>
<th>command</th>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>breakpoints/b/br</td>
<td>Prints to screen all breakpoints</td>
</tr>
<tr>
<td>breakpoint</td>
<td>srcFile:lineNumber if a &lt; b</td>
<td>Conditional insertion of breakpoint</td>
</tr>
<tr>
<td>enable/disable</td>
<td>breakpointNumber</td>
<td>Enable/disable a breakpoint</td>
</tr>
<tr>
<td>delete</td>
<td>breakpointNumber</td>
<td>Delete a breakpoint</td>
</tr>
<tr>
<td>clear</td>
<td>srcFile:lineNumber functionName</td>
<td>Clear breakpoints at a given line or function</td>
</tr>
</tbody>
</table>
Use top to find out the PID of the tasks run by your program (in the top listing PIDs appear on the left, job names on the right).

```bash
% top
```

Attach `gdb` to the relevant PID:

```bash
% gdb -p <PID>
```

or:

```bash
% gdb
(gdb) attach <PID>
```

Once attached the debugger pauses execution of the program.
Same level of control than in a standard debugging session.
Attaching GDB to a running program

Best way to debug production runs. Don’t wait for your wall time to run out!

From the output of `qstat` obtain the node name where your code is running. In the `queue` field you will find an entry like

```
development@i182-103.ta
```

queue name

```
partial node name: i182-103.tacc.utexas.edu
```
GDB Summary

- Compile using debug flags:
  \% icc -g -O0 ./srcFile.c

- Run indicating the directory where the source is:
  \% gdb -d srcDir ./exeFile

- Main commands:
  - run/kill
  - continue/next/step
  - break/watch
  - print
  - where
  - help
DDT: Parallel Debugger with GUI

Allinea Distributed Debugger Tool

- Multiplatform
- Supports all MPI distributions
- Capable of debugging large scale OMP/MPI
- Comprehensive
  - Memory checking
  - MPI message tracking
- Useful Graphical User Interface

www.allinea.com
DDT - Run

- General Options
- Queue Submission Parameters
- Processor and thread number
- Advanced Options
DDT - Run

- General Options
- Queue Submission Parameters
- Processor and thread number
- Advanced Options
- DDT 3.2
DDT – Queue Parameters

Each of these parameters may be changed

Project must be set!!
DDT: The debug session

Process controls
Process groups window
Project navigation window
Code window
Variable window
Stack view and output window
Evaluation window
DDT: Memory Leaks

Go to View -> Current Memory Usage

Process 0 is using much more memory than the others.

This looks like a memory leak.
DDT Summary

• ssh to Lonestar allowing X11 forwarding:
  \% ssh -X username@lonestar.tacc.utexas.edu

• Compile with debugging flags:
  \% mpicc -g -O0 ./srcFile.c

• Load the ddt module
  \% module load ddt

• Run ddt
  \% ddt ./exeFile

• Configure ddt properly before submission:
  – Options ➔ MPI version
  – Queue Parameters ➔ Wallclock/CPU/Project
  – Advanced ➔ Memory Checking
Notes on Eclipse PTP

• Eclipse PTP is **FREE**
• Eclipse PTP is part of XSEDE: your tickets about Eclipse PTP will be answered by the Eclipse PTP developers (they are good about tickets)
• Eclipse PTP is a great tool to debug code locally (i.e. on your own workstation/laptop) before moving to production on XSEDE systems
• Eclipse PTP supports remote development, with existing configurations for all XSEDE machines including TACC systems
• Power IDE with refactoring, code completion, static analysis, collaborative code development...
/*
 * This file is part of the MPI Pi C project.
 * 
 * Name: MPI Pi C
 * Author: Yacoub El Kharra
 * Version: 1.0
 * Copyright: Your copyright notice
 * Description: Calculate Pi using MPI
 *
*/

#include <mpi.h>
#include <stdio.h>
#include <string.h>

void calc_pi(int rank, int num_procs)
{
    int i;
    int num_intervals;
    double h;
    double mpi_pi;
    double pi;
    double sum;
    double x;

    /* set number of intervals to calculate */
    if (rank == 0)
    {
        num_intervals = 1000000000;
    }

    /* tell others how many intervals */
    MPI_Bcast(&num_intervals, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* now everyone does their calculation */
    h = 1.0 / (double) num_intervals;
    sum = 0.0;
    for (i = rank + 1; i <= num_intervals; i += num_procs)
    {
        x = (i - rank) * h;
        mpi_pi = 4.0 * h / x;
        pi += mpi_pi;
    }
    MPI_Reduce(&pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    printf("Pi = %f\n", pi);
}
/* find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

if (my_rank != 0) {
    Expression Type Value
    == my_rank | int 0
}
else {
    Name: my rank
    Details: 0
    Default: 0
    Decimal: 0
    Hex: 0
    Binary: 0
}

Another transport will be used instead, although this may result in lower performance.

Module: OpenFabrics (openib)
Host: dauntless.tacc.utexas.edu
Choose Resource Manager Type
Select the type of resource manager to use

<table>
<thead>
<tr>
<th>Resource Manager Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid Engine-Generic-Batch</td>
</tr>
<tr>
<td>IBM LoadLeveler</td>
</tr>
<tr>
<td>IBM LoadLeveler (Blue Gene)</td>
</tr>
<tr>
<td>IBM Parallel Environment</td>
</tr>
<tr>
<td>IBM Parallel Environment (proxy)</td>
</tr>
<tr>
<td>MPICH2</td>
</tr>
<tr>
<td>MPICH2-Generic-Interactive</td>
</tr>
<tr>
<td>Open MPI</td>
</tr>
<tr>
<td>Open MPI-Generic-Interactive</td>
</tr>
<tr>
<td>PBS-BGP-Batch</td>
</tr>
<tr>
<td>PBS-BGP-Q-Batch</td>
</tr>
<tr>
<td>PBS-Generic-Batch</td>
</tr>
<tr>
<td>PBS-Generic-Interactive</td>
</tr>
<tr>
<td>Remote Launch</td>
</tr>
<tr>
<td>Remote-Generic-Interactive</td>
</tr>
<tr>
<td>SLURM</td>
</tr>
<tr>
<td>SLURM-BGP-Batch</td>
</tr>
<tr>
<td>SLURM-BGP-Q-Batch</td>
</tr>
<tr>
<td>SLURM-Generic-Batch</td>
</tr>
<tr>
<td>Torque-Generic-Batch</td>
</tr>
<tr>
<td>Torque-Generic-Interactive</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns.pbs batchSize.xe</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns.pbs batchSize.xk</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns.pbs batchSize.openmp2</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns.pbs batchSize.openmp</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns.pbs batchSize.openmpi</td>
</tr>
<tr>
<td>edu.uiuc.edu.ncsa.bluewaterns.pbs batchSize.sge master</td>
</tr>
<tr>
<td>edu.utexas.tacc.lonestarsgebatch</td>
</tr>
<tr>
<td>edu.utexas.tacc.ranger.sgebatch</td>
</tr>
<tr>
<td>edu.utknics.keeneland.pbsbatch</td>
</tr>
<tr>
<td>edu.utknics.kraken.pbsbatch</td>
</tr>
<tr>
<td>gov.anl.alcf.bgp.pbsbatch</td>
</tr>
<tr>
<td>gov.anl.alcf.bgp.pbsbatch</td>
</tr>
</tbody>
</table>
Debugging: Conclusions

• You ALWAYS want to debug your code if:
  o Program exhibits erratic, random behavior when you are not using random variables in control statements (hint: memory corruption/stomping)
  o Program appears to "hang" somewhere: attach a debugger and see "where" it is stuck. Profiling will NOT help with "stuck" program flow
  o Program exits way too fast (and you do not have exit error messages built into your code, which you want to consider)
  o Wrong/bad results: unless you are working with infinite or semi-infinite Cantor sets, or attempting to use GiNaC, you should not be getting NaN's or Inf's in your variables: add conditional watch expressions to halt execution when you encounter these values (or use isnan, isinf functions in C)

• This is not to say that you cannot debug your code with print statements. Whatever gets your code operating at designed parameters: performance, readability, maintainability, rapid development, informative crashes/bug-free/idiot-proof etc... that is what you want to be doing. Debuggers get you there quicker
Profiling

timers & gprof
Disclaimer: counting flops

• Lonestar: Cannot count FLOPS accurately, but can make reasonable estimates
• Stampede (Sandy Bridge): Cannot count FLOPS accurately
  – Results always overcount, by amounts that depend on how busy the memory system is.
• Stampede (Xeon Phi): Cannot count FLOPS at all (
Lonestar != Stampede

**Lonestar**
- Command line timing
- gprof
- Perfexpert (fairly reliable)
- IPM
- mpiP
- Tau
- HPCToolkit

**Stampede**
- Command line timing
- gprof
- Perfexpert (ignore metrics relying on flops)
- Tau
Timers: Command Line

- The command `time` is available in most Unix systems.
- It is simple to use (no code instrumentation required).
- Gives total execution time of a process and all its children in seconds.

```
% /usr/bin/time -p ./exeFile
real 9.95
user 9.86
sys 0.06
```

Leave out the `-p` option to get additional information:

```
% time ./exeFile
% 9.860u 0.060s 0:09.95 99.9%0+0k 0+0io 0pf+0w
```
Timers: Code Section

INTEGRER :: rate, start, stop
REAL    :: time

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
CALL SYSTEM_CLOCK(COUNT = start)

! Code to time here

CALL SYSTEM_CLOCK(COUNT = stop)
time = REAL((stop - start)/rate)

#include <time.h>

double start, stop, time;
start = (double)clock()/CLOCKS_PER_SEC;
/* Code to time here */

stop = (double)clock()/CLOCKS_PER_SEC;
time = stop - start;
About GPROF

GPROF is the GNU Project PROFiler.  
gnu.org/software/binutils/

• Requires recompilation of the code.

• Compiler options and libraries provide wrappers for each routine call and periodic sampling of the program.

• Provides three types of profiles
  - Flat profile
  - Call graph
  - Annotated source
Types of Profiles

• Flat Profile
  – CPU time spent in each function (self and cumulative)
  – Number of times a function is called
  – Useful to identify most expensive routines

• Call Graph
  – Number of times a function was called by other functions
  – Number of times a function called other functions
  – Useful to identify function relations
  – Suggests places where function calls could be eliminated

• Annotated Source
  – Indicates number of times a line was executed
Profiling with gprof

Use the `-pg` flag during compilation:

```
% gcc -g -pg ./srcFile.c
% icc -g -p ./srcFile.c
% pgcc -g -pg ./srcFile.c
```

Run the executable. An output file `gmon.out` will be generated with the profiling information.

Execute `gprof` and redirect the output to a file:

```
% gprof ./exeFile gmon.out > profile.txt
% gprof -l ./exeFile gmon.out > profile_line.txt
% gprof -A ./exeFile gmon.out > profile_anotated.txt
```
Flat profile

In the flat profile we can identify the most expensive parts of the code (in this case, the calls to matSqrt, matCube, and sysCube).

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>seconds</th>
<th>total time</th>
<th>seconds</th>
<th>calls</th>
<th>s/call</th>
<th>s/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.00</td>
<td>2.47</td>
<td>2.47</td>
<td>1.24</td>
<td></td>
<td>2</td>
<td>1.24</td>
<td>matSqrt</td>
</tr>
<tr>
<td>24.70</td>
<td>3.69</td>
<td>1.22</td>
<td>1.22</td>
<td></td>
<td>1</td>
<td>1.22</td>
<td>matCube</td>
</tr>
<tr>
<td>24.70</td>
<td>4.91</td>
<td>1.22</td>
<td>1.22</td>
<td></td>
<td>1</td>
<td>1.22</td>
<td>sysCube</td>
</tr>
<tr>
<td>0.61</td>
<td>4.94</td>
<td>0.03</td>
<td>0.03</td>
<td></td>
<td>1</td>
<td>0.03</td>
<td>main</td>
</tr>
<tr>
<td>0.00</td>
<td>4.94</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>2</td>
<td>0.00</td>
<td>vecSqrt</td>
</tr>
<tr>
<td>0.00</td>
<td>4.94</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>1</td>
<td>0.00</td>
<td>sysSqrt</td>
</tr>
<tr>
<td>0.00</td>
<td>4.94</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>1</td>
<td>0.00</td>
<td>vecCube</td>
</tr>
</tbody>
</table>
## Call Graph Profile

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>&lt;hicore&gt; (8)</td>
</tr>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.03</td>
<td>4.91</td>
<td>1</td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>1.24</td>
<td>1/1</td>
<td></td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td></td>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td></td>
<td>1.22</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>sysCube [5]</td>
</tr>
<tr>
<td></td>
<td>1.22</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>matCube [4]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td>vecSqrt [6]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>vecCube [7]</td>
</tr>
<tr>
<td></td>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td>[2]</td>
<td>50.0</td>
<td>2.47</td>
<td>0.00</td>
<td>2</td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>1.24</td>
<td>1/1</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>[3]</td>
<td>25.0</td>
<td>0.00</td>
<td>1.24</td>
<td>1</td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td></td>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td>vecSqrt [6]</td>
</tr>
</tbody>
</table>
Visual Call Graph
# Call Graph Profile

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>&lt;hicore&gt;</td>
<td>(8)</td>
</tr>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.03</td>
<td>4.91</td>
<td>1</td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td>1.24</td>
<td>1/1</td>
<td></td>
<td></td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td>1.22</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>sysCube [5]</td>
</tr>
<tr>
<td>1.22</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>matCube [4]</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>vecSqrt [6]</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>vecCube [7]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>--------</td>
<td>------------</td>
</tr>
<tr>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td>[2]</td>
<td>50.0</td>
<td>2.47</td>
<td>0.00</td>
<td>2</td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>----------</td>
<td>--------</td>
<td>------------</td>
</tr>
<tr>
<td>0.00</td>
<td>1.24</td>
<td>1/1</td>
<td></td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>[3]</td>
<td>25.0</td>
<td>0.00</td>
<td>1.24</td>
<td>1</td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>vecSqrt [6]</td>
</tr>
</tbody>
</table>

---

**Note:** The table above represents a call graph profile, showing the time profile of function calls in a program. Each row represents a call, with columns for index, percentage of time, self time, children time, called functions, and their indices.
Visual Call Graph
# Call Graph Profile

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td>&lt;hicore&gt;</td>
<td>(8)</td>
</tr>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.03</td>
<td>4.91</td>
<td>1</td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>1.24</td>
<td>1/1</td>
<td></td>
<td>sysSqrt [3]</td>
</tr>
<tr>
<td>1.24</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>matSqrt [2]</td>
</tr>
<tr>
<td>1.22</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>sysCube [5]</td>
</tr>
<tr>
<td>1.22</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>matCube [4]</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td></td>
<td></td>
<td>vecSqrt [6]</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td></td>
<td></td>
<td>vecCube [7]</td>
</tr>
</tbody>
</table>

-----------------------------------------------
| [2]   | 50.0   | 2.47 | 0.00     | 2      | matSqrt [2]|

-----------------------------------------------
| [3]   | 25.0   | 0.00 | 1.24     | 1      | sysSqrt [3]|
| 1.24  | 0.00   | 1/2  |          |        | matSqrt [2]|
| 0.00  | 0.00   | 1/2  |          |        | vecSqrt [6]|

-----------------------------------------------
Visual Call Graph

main

sysSqrt

matSqrt

matSqrt

vecSqrt

matCube

vecCube

sysCube
PERF-EXPERT
About PerfExpert

- Brand new tool, locally developed at UT
- Easy to use and understand
- Great for quick profiling and for beginners
- Provides recommendation on “what to fix” in a subroutine
- Collects information from PAPI using HPCToolkit
- No MPI specific profiling, no 3D visualization, no elaborate metrics
- Combines ease of use with useful interpretation of gathered performance data
- Optimization suggestions!!!
Profiling with PerfExpert: Compilation

• Load the java, papi, and perfexpert modules:
  – \texttt{module load java papi perfexpert}

• Compile the code with full optimization and with the -g flag:
  – \texttt{mpicc -g -O3 source.c}
  – \texttt{mpif90 -g -O3 source.f90}

• In your job submission script:
  \begin{verbatim}
  perfexpert_run_exp ./<executable name> <executable args>
  perfexpert 0.1 experiment-*.xml
  \end{verbatim}

Threshold of 0.1 lists only functions and loops that represent ≥ 10% of total runtime
PerfExpert Analysis Output

Loop in function main() at Integrator.c:81 (98.9% of the total runtime)

<table>
<thead>
<tr>
<th>Ratio to total instrns</th>
<th>%</th>
<th>0........</th>
<th>25........</th>
<th>50........</th>
<th>75........</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating point</td>
<td>30</td>
<td>**********</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data accesses</td>
<td>71</td>
<td>********************</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GFLOPS (% max)</td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance assessment:
- LCPI: good......okay......fair......poor......bad....
- Overall: 4.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+

Upper bound estimates:
- Data accesses: 33.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
  - L1d hits: 2.2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
  - L2d hits: 2.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
  - L2d misses: 28.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- Instruction accesses: 0.4 >>>>>>>>
  - L1i hits: 0.4 >>>>>>>>
  - L2i hits: 0.0 >
  - L2i misses: 0.0 >
- Data TLB: 0.0 >
- Instruction TLB: 0.0 >
- Branch instructions: 0.1 >>
  - Correctly predicted: 0.1 >>
  - Mispredicted: 0.0 >
- Floating-point instr: 1.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  - Fast FP instr: 1.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
  - Slow FP instr: 0.0 >

Overall loop performance is bad.

Biggest problem is data accesses that miss in the L2 cache.

Remaining performance categories are good.
PerfExpert Summary

• Load the papi, java, and perfexpert modules:
  ```
  % module load papi java perfexpert
  ```

• In your job submission script, make sure you have:
  ```
  perfexpert_run_exp ./<executable name> <executable args>
  perfexpert 0.1 experiment-*.xml
  ```

• Send output to AutoSCOPE for optimization suggestions:
  ```
  perfexpert 0.1 experiment-integrator.xml | autoscope
  ```

• Apply suggestions from autoscope and run again. Check to see if the wall clock time is reduced or not
Optimization Suggestions

Code Section: Loop in function main() at Integrator.c:81  (98.9% of the total runtime)

change the order of loops

loop i { loop j {...} } → loop j { loop i {...} }

employ loop blocking

loop i {loop k {loop j {c[i][j] = c[i][j] + a[i][k] * b[k][j];}}}
→
loop k step s {loop j step s {loop i {
    for (kk = k; kk < k + s; kk++) {
        for (jj = j; jj < j + s; jj++) {
            c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];
        }
    }
}}}

apply loop fission so every loop accesses just a couple of different arrays

loop i {a[i] = a[i] * b[i] - c[i];} →
loop i {a[i] = a[i] * b[i];} loop i {a[i] = a[i] - c[i];}
IPM: INTEGRATED PERFORMANCE MONITORING
IPM: Integrated Performance Monitoring

• “IPM is a portable profiling infrastructure for parallel codes. It provides a low-overhead performance summary of the computation and communication in a parallel program”

• IPM is a quick, easy and concise profiling tool

• The level of detail it reports is smaller than TAU, PAPI or HPCToolkit
IPM: Integrated Performance Monitoring

• IPM features:
  – easy to use
  – has low overhead
  – is scalable

• Requires no source code modification, just adding the “-g” option to the compilation

• Produces XML output that is parsed by scripts to generate browser-readable html pages
IPM: Integrated Performance Monitoring

- Available on Ranger and Lonestar with the mvapich libraries
- Create ipm environment with module command before running code: "module load ipm"
- In your job script, set up the following ipm environment before the ibrun command:

  ```
  module load ipm
  export LD_PRELOAD=$TACC_IPM_LIB/libipm.so
  export IPM_REPORT=full
  ibrun <my executable> <my arguments>
  ```
IPM: Integrated Performance Monitoring

- Export `LD_PRELOAD=$TACC_IPM_LIB/libipm.so`
  - must be inside job script

- **IPM_REPORT**: full, terse or none are the levels of information

- **IPM_MPI_THRESHOLD**: Reports only routines using this percentage (or more) of MPI time.
  - e.g. “**IPM_MPI_THRESHOLD 0.3**” report subroutines that consume more than 30% of the total MPI time.

- Important details: “**module help ipm**”
IPM: Text Output

##IPMv0.922###########################################################################
# command : /work/01125/yye00/ICAC/cactus_SandTank SandTank.par
# host : i101-309/x86_64_Linux mpi_tasks : 32 on 2 nodes
# start : 05/26/09/11:49:06 wallclock : 2.758892 sec
# stop : 05/26/09/11:49:09 %comm : 2.01
# gbytes : 4.38747e+00 total gflop/sec : 9.39108e-02 total
#
##########################################################
# region : * [ntasks] = 32
#
# [total] <avg> min max
# entries 32 1 1 1
# wallclock 88.2742 2.75857 2.75816 2.75889
# user 5.51634 0.172386 0.148009 0.200012
# system 1.771 0.0553438 0.0536683 0.056717
# %comm 2.00602 1.94539 2.05615
# gflop/sec 0.0939108 0.00293471 0.00293338 0.002952
# gbytes 4.38747 0.137109 0.136581 0.144985
#
##########################################################
# PAPI_FP_OPS 2.5909e+08 8.09655e+06 8.09289e+06 8.14685e+06
# PAPI_TOT_CYC 6.80291e+09 2.12591e+08 2.02236e+08 2.19109e+08
# PAPI_VEC_INS 5.95596e+08 1.86124e+07 1.85964e+07 1.8756e+07
# PAPI_TOT_INS 4.16377e+09 1.30118e+08 1.0987e+08 1.35676e+08
#
# 
## MPI_Allreduce 0.978938 53248 55.28 1.11
# MPI_Comm_rank 0.316355 6002 17.86 0.36
# MPI_Barrier 0.247135 3680 13.95 0.28
# MPI_Allgatherv 0.16621 2848 9.39 0.19
# MPI_Bcast 0.0217298 576 1.23 0.02
# MPI_Allgather 0.0216982 672 1.23 0.02
# MPI_Recv 0.0186796 32 1.05 0.02
# MPI_Comm_size 0.000139921 2112 0.01 0.00
# MPI_Send 0.000115622 32 0.01 0.00

###########################################################################
## IPM: Event Statistics

<table>
<thead>
<tr>
<th>Event</th>
<th>Buffer Size</th>
<th>N calls</th>
<th>Total Time</th>
<th>Min Time</th>
<th>Max Time</th>
<th>% MPI</th>
<th>% Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce</td>
<td>8</td>
<td>79680</td>
<td>4.178</td>
<td>8.225e-06</td>
<td>8.882e-04</td>
<td>14.87</td>
<td>1.72</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>4</td>
<td>1024</td>
<td>4.047</td>
<td>5.914e-08</td>
<td>6.413e-02</td>
<td>14.35</td>
<td>1.65</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>512</td>
<td>39936</td>
<td>3.803</td>
<td>1.660e-05</td>
<td>1.170e-01</td>
<td>13.49</td>
<td>1.56</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>4</td>
<td>25472</td>
<td>2.250</td>
<td>6.012e-07</td>
<td>1.552e-02</td>
<td>7.98</td>
<td>0.92</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>0</td>
<td>64</td>
<td>1.176</td>
<td>1.814e-02</td>
<td>1.865e-02</td>
<td>4.17</td>
<td>0.48</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>8</td>
<td>630</td>
<td>1.028</td>
<td>3.427e-07</td>
<td>1.647e-02</td>
<td>3.55</td>
<td>0.42</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>4</td>
<td>4556</td>
<td>0.943</td>
<td>2.738e-07</td>
<td>1.833e-02</td>
<td>3.34</td>
<td>0.39</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>14976</td>
<td>144</td>
<td>0.722</td>
<td>1.030e-03</td>
<td>7.308e-03</td>
<td>2.56</td>
<td>0.30</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>0</td>
<td>106948</td>
<td>0.620</td>
<td>3.725e-08</td>
<td>9.872e-03</td>
<td>2.20</td>
<td>0.25</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>0</td>
<td>6093</td>
<td>0.542</td>
<td>4.615e-07</td>
<td>1.358e-02</td>
<td>1.92</td>
<td>0.22</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1248</td>
<td>27462</td>
<td>0.519</td>
<td>5.183e-07</td>
<td>2.723e-04</td>
<td>1.84</td>
<td>0.21</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>16224</td>
<td>144</td>
<td>0.517</td>
<td>4.263e-04</td>
<td>7.129e-03</td>
<td>1.83</td>
<td>0.21</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1352</td>
<td>20370</td>
<td>0.496</td>
<td>5.197e-07</td>
<td>5.783e-03</td>
<td>1.76</td>
<td>0.20</td>
</tr>
<tr>
<td>MPI_Start</td>
<td>0</td>
<td>269196</td>
<td>0.396</td>
<td>3.623e-07</td>
<td>3.685e-05</td>
<td>1.40</td>
<td>0.16</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>13824</td>
<td>48</td>
<td>0.298</td>
<td>4.035e-03</td>
<td>7.227e-03</td>
<td>1.06</td>
<td>0.12</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1152</td>
<td>10980</td>
<td>0.243</td>
<td>5.383e-07</td>
<td>2.310e-04</td>
<td>0.86</td>
<td>0.10</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>216</td>
<td>576</td>
<td>0.231</td>
<td>2.302e-06</td>
<td>5.843e-03</td>
<td>0.82</td>
<td>0.09</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>4</td>
<td>9088</td>
<td>0.215</td>
<td>5.118e-07</td>
<td>1.793e-03</td>
<td>0.76</td>
<td>0.09</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>184</td>
<td>11</td>
<td>0.210</td>
<td>1.633e-02</td>
<td>2.135e-02</td>
<td>0.74</td>
<td>0.09</td>
</tr>
<tr>
<td>MPI_Scan</td>
<td>4</td>
<td>384</td>
<td>0.144</td>
<td>2.259e-05</td>
<td>1.600e-03</td>
<td>0.51</td>
<td>0.06</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>147</td>
<td>453</td>
<td>0.141</td>
<td>4.866e-07</td>
<td>1.406e-02</td>
<td>0.50</td>
<td>0.06</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>4</td>
<td>448</td>
<td>0.132</td>
<td>4.345e-07</td>
<td>5.805e-03</td>
<td>0.47</td>
<td>0.05</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>320</td>
<td>18</td>
<td>0.120</td>
<td>3.002e-06</td>
<td>1.284e-02</td>
<td>0.42</td>
<td>0.05</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>17576</td>
<td>42</td>
<td>0.108</td>
<td>6.682e-05</td>
<td>6.406e-03</td>
<td>0.38</td>
<td>0.04</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>72</td>
<td>5</td>
<td>0.103</td>
<td>1.547e-02</td>
<td>2.038e-02</td>
<td>0.36</td>
<td>0.04</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>96</td>
<td>38</td>
<td>0.091</td>
<td>2.882e-06</td>
<td>1.563e-02</td>
<td>0.32</td>
<td>0.04</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>624</td>
<td>140</td>
<td>0.088</td>
<td>1.126e-06</td>
<td>7.880e-03</td>
<td>0.31</td>
<td>0.04</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>8</td>
<td>9</td>
<td>0.085</td>
<td>8.373e-07</td>
<td>7.696e-02</td>
<td>0.30</td>
<td>0.03</td>
</tr>
</tbody>
</table>
Load balance by task: memory, flops, timings

Communication balance by task (sorted by MPI time)
IPM Buffer Size Distribution: % of Comm Time

Message Buffer Size Distributions: time

Buffer size (bytes)

 cumulative values, values

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER
Buffer Size Distribution: Ncalls

Message Buffer Size Distributions: Ncalls

- % calls ≤ buffer size
- Buffer size (bytes)

cumulative values, values
Communication Topology

Communication Topology: point to point data flow

- **data sent**, **data recv**, **time spent**, **map data file**, **map adjacency file**
IPM: Integrated Performance Monitoring

• When to use IPM?
  – To quickly find out where your code is spending most of its time (in both computation and communication)
  – For performing scaling studies (both strong and weak)
  – When you suspect you have load imbalance and want to verify it quickly
  – For a quick look at the communication pattern
  – To find out how much memory you are using per task
  – To find the relative communication & compute time
IPM: Integrated Performance Monitoring

• When IPM is NOT the answer
  – When you already know where the performance issues are
  – When you need detailed performance information on exact lines of code
  – When want to find specific information such as cache misses
Advanced Profiling Tools
: the next level
Advanced Profiling Tools

• Can be intimidating:
  – Difficult to install
  – Many dependences
  – Require kernel patches
  Not your problem!!

• Useful for serial and parallel programs

• Extensive profiling and scalability information

• Analyze code using:
  – Timers
  – Hardware registers (PAPI)
  – Function wrappers
PAPI

PAPI is a Performance Application Programming Interface
icl.cs.utk.edu/papi

• API to use hardware counters
• Behind Tau, HPCToolkit
• Multiplatform:
  – Most Intel & AMD chips
  – IBM POWER 4/5/6
  – Cray X/XD/XT
  – Sun UltraSparc I/II/III
  – MIPS
  – SiCortex
  – Cell
• Available as a module on Lonestar and Stampede (mind the hardware counters issue with sandybridge xeaons).
TAU is a suite of Tuning and Analysis Utilities
www.cs.uoregon.edu/research/tau

• 12+ year project involving
  – University of Oregon Performance Research Lab
  – LANL Advanced Computing Laboratory
  – Research Centre Julich at ZAM, Germany

• Integrated toolkit
  – Performance instrumentation
  – Measurement
  – Analysis
  – Visualization
Tau: Measurements

• Parallel profiling
  – Function-level, block (loop)-level, statement-level
  – Supports user-defined events
  – TAU parallel profile data stored during execution
  – Hardware counter values (multiple counters)
  – Support for callgraph and callpath profiling

• Tracing
  – All profile-level events
  – Inter-process communication events
  – Trace merging and format conversion
PDT is used to instrument your code.

**Replace mpicc and mpif90 in make files with tau_f90.sh and tau_cc.sh**

It is necessary to specify all the components that will be used in the instrumentation (mpi, openmp, profiling, counters [PAPI], etc. However, these come in a limited number of combinations.)

Combinations: First determine what you want to do (profiling, PAPI counters, tracing, etc.) and the programming paradigm (mpi, openmp), and the compiler. PDT is a required component:

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Parallel Paradigm</th>
<th>Collectors</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDT</td>
<td>MPI</td>
<td>PAPI</td>
<td>Intel</td>
</tr>
<tr>
<td>Hand-coded</td>
<td>OMP</td>
<td>Callpath</td>
<td>PGI</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>GNU (gcc)</td>
</tr>
</tbody>
</table>
You can view the available combinations
(alias tauTypes 'ls -C1 $TAU | grep Makefile ').

Your selected combination is made known to the compiler wrapper through the TAU_MAKEFILE environment variable.

E.g. the PDT instrumentation (pdt) for the Intel compiler (icpc) with MPI (mpi) is set with this command:

```
setenv TAU_MAKEFILE $TAU/Makefile.tau-icpc-mpi-pdt
```

Other run-time and instrumentation options are set through TAU_OPTIONS. For verbose:

```
setenv TAU_OPTIONS ‘-optVerbose’
```
Tau Paraprof Overview

- Raw files
- PerfDMF managed (database)
- Application
- Experiment
- Trials

HPMToolkit
Metadata
MpiP
TAU
Tau Paraprof Manager Window

Provides Machine Details

Organizes Runs as: Applications, Experiments and Trials.
Routine Time Experiment

Profile Information is in “GET_TIME_OF_DAY” metric
Mean and Standard Deviation Statistics given.
Multiply_Matrices Routine Results

Function Data Window gives a closer look at a single function:
Float Point OPS trial

Hardware Counters provide Floating Point Operations (Function Data view).
L1 Data Cache Miss trial

Hardware Counters provide L1 Cache Miss Operations.
Call Path

Call Graph Paths (Must select through “thread” menu.)
Call Path

TAU_MAKEFILE =
...Makefile.tau-callpath-icpc-mpi-pdt
Derived Metrics

Select Argument 1 (green ball); Select Argument 2 (green ball); Select Operation; then Apply. Derived Metric will appear as a new trial.
Derived Metrics

Since FP/Miss ratios are constant—must be memory access problem.

Be careful→ even though ratios are constant, cores may do different amounts of work/operations per call.
HPCToolkit

• “HPCToolkit is an open-source suite of multi-platform tools for profile-based performance analysis of applications. The figure below provides an overview of the toolkit components and their relationships.”

• HPCToolkit is under intense development and shows great potential

• HPCToolkit is a mid-range tool: providing detailed profiling information without the need to rewrite code
HPCToolkit: Overview

• To prepare the executable for profiling, compile the code with full optimization as usual but include the “-g” option

• Using HPCToolkit involves four stages:
  – We start by performing a static binary analysis of the compiled executable (hence the need for the “-g” option) using “hpcstruct”
  – You run the code with “hpcrun” to collect the stack profile
  – You interpret the stack profile and correlate it with the source using “hpcprof” to generate the profiling database
  – You view the profiling database with the “hpcviewer”
<table>
<thead>
<tr>
<th>Counter/Event Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_TCM</td>
<td>Level 2 cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>Level 3 cache misses</td>
</tr>
<tr>
<td>PAPI_FPU_IDL</td>
<td>Cycles floating point units are idle</td>
</tr>
<tr>
<td>PAPI_TLB_DM</td>
<td>Data translation lookaside buffer misses</td>
</tr>
<tr>
<td>PAPI_TLB_IM</td>
<td>Instruction translation lookaside buffer misses</td>
</tr>
<tr>
<td>PAPI_STL_ICY</td>
<td>Cycles with no instruction issue</td>
</tr>
<tr>
<td>PAPI_HW_INT</td>
<td>Hardware interrupts</td>
</tr>
<tr>
<td>PAPI_BR_TKN</td>
<td>Conditional branch instructions taken</td>
</tr>
<tr>
<td>PAPI_BR_MSP</td>
<td>Conditional branch instructions mispredicted</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>Instructions completed</td>
</tr>
<tr>
<td>PAPI_FP_INS</td>
<td>Floating point instructions</td>
</tr>
<tr>
<td>PAPI_BR_INS</td>
<td>Branch instructions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Counter/Event Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_VEC_INS</td>
<td>Vector/SIMD instructions</td>
</tr>
<tr>
<td>PAPI_RES_STL</td>
<td>Cycles stalled on any resource</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>Total cycles</td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>Level 1 data cache accesses</td>
</tr>
<tr>
<td>PAPI_L2_DCA</td>
<td>Level 2 data cache accesses</td>
</tr>
<tr>
<td>PAPI_L2_ICH</td>
<td>Level 2 instruction cache hits</td>
</tr>
<tr>
<td>PAPI_L1_ICA</td>
<td>Level 1 instruction cache accesses</td>
</tr>
<tr>
<td>PAPI_L2_ICA</td>
<td>Level 2 instruction cache accesses</td>
</tr>
<tr>
<td>PAPI_L1_ICR</td>
<td>Level 1 instruction cache reads</td>
</tr>
<tr>
<td>PAPI_L2_TCA</td>
<td>Level 2 total cache accesses</td>
</tr>
<tr>
<td>PAPI_L3_TCR</td>
<td>Level 3 total cache reads</td>
</tr>
<tr>
<td>PAPI_FML_INS</td>
<td>Floating point multiply instructions</td>
</tr>
<tr>
<td>PAPI_FAD_INS</td>
<td>Floating point add instructions (Also includes subtract instructions)</td>
</tr>
<tr>
<td>PAPI_FDV_INS</td>
<td>Floating point divide instructions (Counts both divide and square root instructions)</td>
</tr>
<tr>
<td>PAPI_FSQ_INS</td>
<td>Floating point square root instructions (Counts both divide and square root instructions)</td>
</tr>
<tr>
<td>PAPI_FP_OPS</td>
<td>Floating point operations</td>
</tr>
</tbody>
</table>
Profiling dos and don’ts

**DO**
- Test every change you make
- Profile typical cases
- Compile with optimization flags
- Test for scalability

**DO NOT**
- Assume a change will be an improvement
- Profile atypical cases
- Profile *ad infinitum*
  - Set yourself a goal or
  - Set yourself a time limit
Other tools

• **Valgrind** [valgrind.org]
  – Powerful instrumentation framework, often used for debugging memory problems

• **MPIP** [mpip.sourceforge.net]
  – Lightweight, scalable MPI profiling tool

• **Tau** [www.cs.uoregon.edu/research/tau]
  – Suite of Tuning and Analysis Utilities

• **Scalasca** [www.fz-juelich.de/jsc/scalasca]
  – Similar to Tau, complete suit of tuning and analysis tools.

• **HPCToolkit** [www.hpctoolkit.org]
  – Interesting tool with a lot of promise