Profiling and debugging

John Cazes

Texas Advanced Computing Center
Outline

Debugging

• GDB
  – Basic use
  – Attaching to a running job

• DDT
  – Identify MPI problems using Message Queues
  – Catch memory errors

Profiling

• Timers

• GPROF

• Advanced Tools
  • Gprof
  • IPM
  • PerfExpert
  • Tau (and PAPI)
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!
About GDB

GDB is the GNU Project DeBugger
www.gnu.org/software/gdb/

Looks inside a running program (SERIAL)

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</td>
<td>set a breakpoint by line number or function name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>functionName</td>
<td></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**

```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
",x,y,res);
    }
    return 0;
}

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

**divcrash.f90**

```fortran
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 \texttt{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:

\texttt{(gdb) kill}
\texttt{(gdb) quit}
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. `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 srcFile:lineNumber if a &lt; b</td>
<td></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>
Attaching GDB to a running program

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).

% top

Attach `gdb` to the relevant PID:

% gdb -p <PID>

or:

% 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
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/CPUs/Project
  – Advanced ➔ Memory Checking
Profiling

timers & gprof
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

```fortran
INTEGER :: 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)
```

```c
#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 spend 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:

```bash
% 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:

```bash
% gprof ./exeFile gmon.out > profile.txt
% gprof -l ./exeFile gmon.out > profile_line.txt
% gprof -A ./exeFile gmon.out > profile.annotated.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 seconds</th>
<th>self seconds</th>
<th>total calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.00</td>
<td>2.47</td>
<td>2.47</td>
<td>2</td>
<td>1.24</td>
<td>1.24</td>
<td>matSqrt</td>
</tr>
<tr>
<td>24.70</td>
<td>3.69</td>
<td>1.22</td>
<td>1</td>
<td>1.22</td>
<td>1.22</td>
<td>matCube</td>
</tr>
<tr>
<td>24.70</td>
<td>4.91</td>
<td>1.22</td>
<td>1</td>
<td>1.22</td>
<td>1.22</td>
<td>sysCube</td>
</tr>
<tr>
<td>0.61</td>
<td>4.94</td>
<td>0.03</td>
<td>1</td>
<td>0.03</td>
<td>4.94</td>
<td>main</td>
</tr>
<tr>
<td>0.00</td>
<td>4.94</td>
<td>0.00</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>vecSqrt</td>
</tr>
<tr>
<td>0.00</td>
<td>4.94</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>1.24</td>
<td>sysSqrt</td>
</tr>
<tr>
<td>0.00</td>
<td>4.94</td>
<td>0.00</td>
<td>1</td>
<td>0.00</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>0.00</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></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></td>
<td></td>
<td></td>
<td></td>
<td>-----------------</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>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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-----------------</td>
</tr>
</tbody>
</table>

---

The University of Texas at Austin
Texas Advanced Computing Center
Visual Call Graph

main

- sysSqrt
- matSqrt
- vecSqrt
- matCube
- vecCube
- sysCube
## 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>
</tbody>
</table>

---------------------------------------------------------------
|       | 1.24   | 0.00 | 1/2      |        | main [1]     |
|       | 1.24   | 0.00 | 1/2      |        | sysSqrt [3]  |

---------------------------------------------------------------
|       | 0.00   | 1.24 | 1/1      |        | main [1]     |
| [2]   | 50.0   | 2.47 | 0.00     | 2      | matSqrt [2]  |

---------------------------------------------------------------
|       | 0.00   | 1.24 | 1/1      |        | main [1]     |
| [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  vecSqrt  matCube  vecCube  sysCube

matSqrt
## Call Graph Profile

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

matCube

vecCube

sysCube

vecSqrt
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:
  – `module load java papi perfexpert`

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

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

  Threshold of 0.1 lists only functions and loops that represent ≥ 10% of total runtime
Loop in function main() at Integrator.c:81  (98.9% of the total runtime)

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

```c
loop i { loop j {...} } → loop j { loop i {...} }
```

---

**employ loop blocking**

```c
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**

```c
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: Integrated Performance Monitoring

Command: /work/01125/yye00/IPM/Benchmark/Defiant.exe perturb -runperturbed -da_grid_x 50 -da_grid_y 50 -da_grid_z 50 -seed_phi 3454345 -seed_k11 56756756 -seed_k22 235759 -seed_k33 23456 -seed_flowmask 3222111 -percentage_phi 0.1 -percentage_k11 0.1 -percentage_k22 0.1 -percentage_k33 0.1 -percentage_flowmask 0.15 -endtime 1.0 -ksp_type bicg -pc_type jacobi

codename: unknown state: running
username: yye00 group: G-801077
host: i115-108 (x86_64_Linux) mpi_tasks: 64 on 4 hosts
start: 07/13/10/00:28:10 wallclock: 3.80580e+00 sec
stop: 07/13/10/00:28:13 %comm: 11.5752798360397

total memory: 10.85705 gbytes total gflop/sec: 1.02192900310053

### Computation

<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
<th>Pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_FP_OPS</td>
<td>388925686</td>
<td>*</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>93055641837</td>
<td>*</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>82058705179</td>
<td>*</td>
</tr>
<tr>
<td>PAPI_VEC_INS</td>
<td>8293137711</td>
<td>*</td>
</tr>
</tbody>
</table>

### Communication

% of MPI Time

- MPI_Allreduce
- MPI_Bcast
- MPI_Isend
- MPI_Send
- MPI_Barrier
- MPI_Comm_rank
- MPI_Start
- MPI_Sendinit
- MPI_Recvinit
- MPI_Recv
- MPI_Scatter
- MPI_Scatterinit

### HPM Counter Statistics

<table>
<thead>
<tr>
<th>Event</th>
<th>Ntasks</th>
<th>Avg</th>
<th>Min(rank)</th>
<th>Max(rank)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_FP_OPS</td>
<td>*</td>
<td>60769638.53</td>
<td>53254674 (63)</td>
<td>68822056 (21)</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>*</td>
<td>1453994403.70</td>
<td>1346848050 (23)</td>
<td>1646491906 (12)</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>*</td>
<td>1282167268.42</td>
<td>1134309464 (0)</td>
<td>1477795580 (12)</td>
</tr>
<tr>
<td>PAPI_VEC_INS</td>
<td>*</td>
<td>129580276.73</td>
<td>113002002 (63)</td>
<td>146915653 (21)</td>
</tr>
<tr>
<td>Communication Event Statistics (100.00% detail, 9.9012e-06 error)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td>N calls</td>
<td>Total Time</td>
<td>Min Time</td>
<td>Max Time</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>8</td>
<td>79680</td>
<td>4.178</td>
<td>8.225e-06</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>4</td>
<td>1024</td>
<td>4.047</td>
<td>5.914e-08</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>512</td>
<td>39936</td>
<td>3.803</td>
<td>1.660e-05</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>4</td>
<td>25472</td>
<td>2.250</td>
<td>6.012e-07</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>0</td>
<td>64</td>
<td>1.176</td>
<td>1.814e-02</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>8</td>
<td>630</td>
<td>1.028</td>
<td>3.427e-07</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>4</td>
<td>4556</td>
<td>0.943</td>
<td>2.738e-07</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>14976</td>
<td>144</td>
<td>0.722</td>
<td>1.030e-03</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>0</td>
<td>106948</td>
<td>0.620</td>
<td>3.725e-08</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>0</td>
<td>6093</td>
<td>0.542</td>
<td>4.615e-07</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1248</td>
<td>27462</td>
<td>0.519</td>
<td>5.183e-07</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>16224</td>
<td>144</td>
<td>0.517</td>
<td>4.263e-04</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1352</td>
<td>20370</td>
<td>0.496</td>
<td>5.197e-07</td>
</tr>
<tr>
<td>MPI_Start</td>
<td>0</td>
<td>269196</td>
<td>0.396</td>
<td>3.623e-07</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>13824</td>
<td>48</td>
<td>0.298</td>
<td>4.035e-03</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1152</td>
<td>10980</td>
<td>0.243</td>
<td>5.383e-07</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>216</td>
<td>576</td>
<td>0.231</td>
<td>2.302e-06</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>4</td>
<td>9088</td>
<td>0.215</td>
<td>5.118e-07</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>184</td>
<td>11</td>
<td>0.210</td>
<td>1.633e-02</td>
</tr>
<tr>
<td>MPI_Scan</td>
<td>4</td>
<td>384</td>
<td>0.144</td>
<td>2.259e-05</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>147</td>
<td>453</td>
<td>0.141</td>
<td>4.866e-07</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>4</td>
<td>448</td>
<td>0.132</td>
<td>4.345e-07</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>320</td>
<td>18</td>
<td>0.120</td>
<td>3.002e-06</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>17576</td>
<td>42</td>
<td>0.108</td>
<td>6.682e-05</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>72</td>
<td>6</td>
<td>0.103</td>
<td>1.547e-02</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>96</td>
<td>38</td>
<td>0.091</td>
<td>2.882e-06</td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>624</td>
<td>140</td>
<td>0.088</td>
<td>1.126e-06</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>8</td>
<td>9</td>
<td>0.085</td>
<td>8.373e-07</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)

% comm time ≤ buffer size

cumulative values, values
Buffer Size Distribution: Ncalls

Message Buffer Size Distributions: Ncalls

% calls ≤ buffer size

Buffer size (bytes)

cumulative values, values
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
  \[\text{Not your problem!!}\]

• 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 Ranger and Lonestar
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
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