# An Insiders Guide: Janus Optimization and Troubleshooting

Ben Cole, Intel Computational Scientist at SNL
Pat Fay, Intel Computational Scientist at LANL
Greg Henry, Intel Computational Scientist in OR

http://www.sandia.gov/ASCI/Red/usage/optimize.html
http://www.sandia.gov/ASCI/Red/usage/optimize.ps

# Overview

- ***Sigportals***
- The Debugger
- NQS
- OpenMP
- The Profiler
- CTH Profiler Example
- IO Cliff Notes
- Math libraries

# Sigportals - Message Passing Failures

Recv SIGPORTAL: msg drop, Rank:304 Portals:62, Load Control Portal src nid:4624,msg_len:6912,match_bits:0x3200d, offset:0 EXITING!

- Fundamental meaning: Something (the app, the OS, something) has dropped a message. Note: at this level the "other node" may be a service/IO node.
- Rank: mynode()
- Portals: Portal number for which the signal has occurred. (more in a minute)
- src nid: Node number (according to the OS, not the application) from which the message was sent: For user communications (MPI,NX), one can get the app node number corresponding to this, but this process is laborious. Contact one of the comp sci's. (and one of us should write a tool for this.)
- len: length, in bytes, of the message
- match_bits: For user communications, this is the message type.
- Offset: Let us know if it is not zero.

# Sigportal Problems by Number

- Depends on the Portal Number
- MPI Portals
  - 53-56, 62 (note below)
  - Sigportals on these are usually caused by exhausting your MPI environment space. See http://www.cs.sandia.gov/~bright/mpi/Options.html
- NX Portal
  - 5,60 NX_RECV(?)

  Sigportals on these are usually caused by exhausting your NX message buffering space. Try increasing them via the -comm flag to yod.
- IO-related portals:
  - 0-4

  We don't expect you to see these except through memory corruption.
- OS-related portals:
  - 61-63
  - 62 is reused for MPI. Its use by the OS is restricted to program loading.

    (so SIGPORTAL on 62 before your program starts to execute is an OS problem, otherwise MPI).

# More Comments on Sigportals

- Almost any of these can also be caused by memory management errors
- You can (though we don’t recommend it) use both NX and MPI in an application.  In particular, if a library uses message-passing calls, it may be using a different communications framework from what you’re using in the application. Beware!

# Overview - the Debugger

- Sigportals
- ***<u>The Debugger</u>***
- NQS
- OpenMP
- The Profiler
- CTH Profiler Example
- IO Cliff Notes
- Math libraries

# What to do when large production jobs stop making progress and you'd like to figure out why (or gather enough information to beat on your code team, at least.)

- Attach the debugger to the 'running' job.
  - —debug -a <PID of the yod command for this run>
- "spacebar" through the initial messages (once they come up).
- Issue these commands:

  more off

  log <filename>
- All output now logged to file

## Other Useful Debug Commands After Attaching

- Suggested debug commands to run:

  process

  recvqueue

  someday: sendqueue

  where

- Real Sandia Apps where this has been done on over 1024 nodes:
  - CTH
  - Pronto
  - Zapotec
  - Alegra

# Debug's 'where' Command Bug

(When debugging your application)

- on a large number of nodes, "where" can cause the debugger itself to memory fault.

  App. Itself is not impacted.

  Log file is preserved

  workaround: use the context command to get this information in chunks (256, 512) of nodes.

# Debug: Using Watchpoints

- Set a watchpoint on a variable. Debug will interrupt your program when variable is read/written (with an optional conditional statement).

- Useful for memory problems (say you know x(100) is getting trashed but you don't know how).

- Example: checking errno. I want to find where in my program free() is failing. Free() sets errno = EINVAL when it finds a problem. Free() can fail due to memory overruns/underruns, invalid pointers.

- Other useful variables for watchpoints:
  - profile_total_bytes_used: set by malloc/free for current total heap usage. Use profiling to find out max heap used and then use 'stop -w' to break when profile_total_bytes_used is near to the max heap used
  - libdbmalloc.a sets malloc_errno when it finds an error.

# Debug : Using Watchpoints Example

- Example: Find free() failures with watchpoints

```
janus ~/tst 148 > cat ck_errno.c
#include <stdio.h>
#include <errno.h>
#include <nx.h>
#include <malloc.h>
#include <stdlib.h>
#include <sys/types.h>
int *my_errno = &errno;
void ck_errno(void) { /*now my_errno is in scope*/
   if(mynode()== -1 && *my_errno == 0)
   { printf("This will never happen\n");}
}
int main(int argc, char **argv) {
     int *int_ptr;
     ck_errno();
     int_ptr = (int *)malloc(100*sizeof(int));
     int_ptr = int_ptr + 1;
     /* int_ptr != base of the malloc'd area so free fails*/
     free(int_ptr);
     printf("above free() should fail.\n");
     printf("errno= %d\n",errno);
    return 0;
}
```

# Debug: Compiling and Running the Watchpoint Example

- Compiling

```
janus ~/tst 148 > pgcc -cougar -g -O0 ck_errno.c -o tstc
```

- Debugging

```
janus ~/tst 149 > debug -sz 1 tstc
*** Debug (Parallel Debugger), Release 2.4
*** reading symbol table for /Net/usr/home/pfay/tst/tstc...
*** initializing Debug for parallel application...
*** load complete
(0) > stop ck_errno
(0) > run
(0) > where   #now errno and my_errno are in scope
***** (0) *****
ck_errno(void) [ck_errno.c #10]
main(int, char**) [ck_errno.c #16]
cstart() [unknown 0x00026243]
__start() [unknown 0x00020120]
(0) > stop -w errno if *my_errno > 2 #only stop if errno > 2
(0) > cont    #now we get an interrupt when the free() fails
(0) > where
***** (0) *****
__free() [unknown 0x000464bc]
free() [unknown 0x000460c6]
main(int, char**) [ck_errno.c #20]
cstart() [unknown 0x00026243]
__start() [unknown 0x00020120]
(0) > print errno
***** (0) *****
 ** `unknown`__free`errno **
errno = 0x00000016
```

# Overview - NQS

- Sigportals
- The Debugger
- ***NQS***
- OpenMP
- The Profiler
- CTH Profiler Example
- IO Cliff Notes
- Math libraries

# What NQS Topics We Can Cover

- We will not, cannot, speak to NQS policies
  - Concerns should be carried through the proper Sandia management chain
  - There has been a recent change in NQS policies.
- We do not have time to discuss how to use NQS.
- We can talk about some features/tools that might not be widely known about NQS.

# Grabnodes.ksh -- Your Personal Interactive Partition

- The problems:
  - interactive partition often full, and often many idle NQS nodes.
  - Interactive partition not large enough.
- Known solution
  - Use NQS to reserve a set of nodes
  - Set the NX_DFLT_PART environment variable in a shell to allow me to use these nodes interactively.
- This tool is an automated script for doing what is described in the FAQ. See http://www.sandia.gov/ASCI/Red/usage/faq.html#ques66
- Under development

# Using NQS Warnlimit for an Orderly Shutdown before Time Expires

- NQS qsub 'warnlimit' parameter lets you pick when your nqs job will get the SIGTERM signal
  - *This will replace Qsignal!*
  - By default, NQS sends a SIGTERM signal to every NQS job grace-time-limit seconds before the job's time expires. Grace-time-limit is 300 seconds currently.
  - The default SIGTERM handler in cougar terminates the job.
  - The qsub command lets you pick when you will get the SIGTERM.
  - You can:
    - install a SIGTERM handler to catch the signal and set a flag indicating that a SIGTERM has been received
    - have your program's main loop check the flag and, if the flag is set, do a restart dump and exit cleanly.
  - Be sure to leave plenty of time to do the restart files. In R2.7.1, you can leave <exiting> fyods if your job doesn't finish writing before the time expires. (NQS sends a SIGKILL signal when the time is up to force your job to terminate.) R2.8 (due Nov. 99?) should greatly reduce the <exiting> fyod problem.

# Example: Using NQS Warnlimit, Sample Program

- Program which tests to see if SIGTERM has arrived:

```
janus 120 > cat main.c
#include <stdio.h>
#include <time.h>

#ifdef __PARAGON__
  /*__PARAGON__ is always defined for asci-red*/
  extern int get_sigterm_count(void);
#else
  int get_sigterm_count(void) {return 0;}
#endif

int main() {
  time_t myt;
  myt = time(NULL);
  printf("start time= %s",ctime(&myt));
  while(1) {
     if(get_sigterm_count() > 0) {
        printf("I got a sigterm, go do shutdown now\n");
        break;
     }
  }
  myt = time(NULL);
  printf("endit time= %s",ctime(&myt));
  return 0;
}
```

# Example: Using NQS Warnlimit, Compile Line and qsub Script

- Compiling:

```
janus 121 > pgcc -cougar main.c -o tstc /usr/community/lib/libsigterm.a
main.c:
Linking:
```

- NQS script:

```
janus 122 > cat getpart
#!/bin/csh
#qsub -re -ro -q intel -lT 5:00,2:00 -lP 1 getpart
#Sample qsub cmd above does:
# Submit job to intel queue for max of 5:00 minutes.
# Have nqs send the SIGTERM 2:00 minutes into the job.
cd $QSUB_WORKDIR
date
yod -sz 1 tstc
date
```

- Submit to NQS for max time of 5 minutes. Warnlimit after 2 minutes:

```
janus 123 > qsub -re -ro -q intel -lT 5:00,2:00 -lP 1 getpart
Request 31050.janus submitted to queue: intel.
janus 124 > qstat
===============================================================
NQS Version:2    BATCH  PIPE REQUESTS on janus
===============================================================
 REQUEST        NAME    OWNER      QUEUE      PRI  NICE CPU      MEM      STATE
31050.janus    getpart pfay       intel      50.0   0  300      UNLIM.  RUNNING
```

# Example: Using NQS Warnlimit, Output and Results

- Results (we got the sigterm 2 minutes after the job started!):

```
janus /scratch/tmp_6/pfay/sigterm 125 > cat getpart.o31050
Sun Sep 19 23:07:18 MDT 1999
installing sigterm handler
start time= Sun Sep 19 23:07:21 1999
I got a sigterm, go do shutdown now
endit time= Sun Sep 19 23:09:16 1999
Sun Sep 19 23:09:21 MDT 1999
logout
```

- Notes:
  — Be sure to leave enough time to shutdown your program.

**Qsignal script will be removed since warnlimit is cleaner and more reliable.**

# Overview - OpenMP

- Sigportals
- The Debugger
- NQS
- ***OpenMP***
- The Profiler
- CTH Profiler Example
- IO Cliff Notes
- Math libraries

# OMP Introduction

1) Usage/Compiling/Linking
2) References and Directives
3) Sample OMP Routine
4) Suggestions for using OMP
5) OMP vs Virtual Nodes

# OMP Usage/Compiling/Linking

- OpenMP or OMP is a shared-memory parallelism standard so that one can write one source code to build on many shared memory platforms.
- PGI's compiler (on Janus) implements a subset of the OpenMP standard.
- Use -mp on the link line:
  - `cif77 -mp -o executable object_files.o`
- Use -mp when compiling the code that contains the directives (defined later):
  - `cif77 -mp -c omp_code.f`
- Use -mp and/or -Mreentrant on any code that is called by your OMP subroutine:
  - `cif77 -mp -c called_by_omp_code.f`
- There are no "special messages" unless something goes wrong.

# OMP Refs. & Directives

**http://www.openmp.org**
**PGI Users Guide: http://www.pgroup.com/ppro_docs/pgiws_ug/pgi30u.htm**

## Fortran Directives

```
◆ PARALLEL ... END PARALLEL !specify parallel region
◆ CRITICAL ... END CRITICAL !allow only 1 cpu at a time in region
◆ MASTER ... END MASTER     !allow only the 'main' cpu in region
◆ SINGLE ... END SINGLE     !allow one 1 cpu in this region, other skip it
◆ DO ... END DO             !parallel do loop
◆ BARRIER                   !synchronize cpus
◆ DOACROSS                  !not OMP, SGI-style parallel DO
◆ PARALLEL DO               !combines PARALLEL & DO
◆ SECTIONS ... END SECTIONS !split work among cpus by section (non-iterative)
◆ PARALLEL SECTIONS         !combines PARALLEL & SECTIONS
◆ ATOMIC                    !enclose next statement in CRITICAL section
◆ FLUSH                     !flush variables to memory
◆ THREADPRIVATE             !make common blocks private to thread
◆ Run-time Library Routines !omp_get_thread_num(),omp_get_num_threads()
```

# OMP C Pragmas

- `#pragma parallel` //define parallel region
- `#pragma critical` //only 1 cpu at a time in region
- `#pragma one processor` //only cpu 0 allowed in region
- `#pragma pfor` //parallel for loop
- `#pragma synchronize` //wait for all cpus
- Run-time Routines //same as Fortran above

# OMP Clauses

```
!$OMP PARALLEL [Clauses]
< Fortran code executed in body of parallel region >
!$OMP END PARALLEL
```

- PRIVATE(list) make 'list' local to thread
- SHARED(list) make 'list' global to all threads
- DEFAULT(PRIVATE | SHARED | NONE)
  - set default scope for variables
- FIRSTPRIVATE(list)
  - initialize private 'list' variables from existing values
- REDUCTION({operator | intrinsic} : list)
  - perform ‘operator’ on ‘list’ at exit
- COPYIN (list) for threadprivate
- IF (scalar_logical_expression)
  - execute region in PARALLEL only IF .TRUE.

# OMP Sample Program, Compile and Run

```
 PROGRAM MASTER
        INTEGER omp_get_thread_num()
        INTEGER A(1000), B(1000), C(1000)
        DO I=1, 1000
           B(I) = I
           C(I) = 2 * I
        ENDDO
!$OMP PARALLEL PRIVATE(J)
        J = omp_get_thread_num()
!$OMP DO
        DO I=1, 1000
           A(I) = B(I) + C(I) + 10000 *J;
        ENDDO
!$OMP END DO
!$OMP END PARALLEL

Compile with:
cif90 -mp omp_sample.f90 -o tstf
Run with:
yod -sz 1 -proc 2 tstf
```

# OMP Need-to-do's!

•Check your results (**rigorously**)
• Cache reuse important for big OMP gains
  (see forthcoming example!)
• Watch out for variables that must be shared
• Use 'default(none)' clause so that new variables introduced must be added to private() or shared() list
• Use profiler to see if elapsed time is shorter.
• Use CRITICAL to isolate subroutines if you get incorrect results.
• Utilities to get variable lists (under development)
  rd_debug - list local/global variables
  omp_xref - list variables used in the parallel region

# Virtual Nodes

- There is a project underway between SNL and Intel to see if a future OS release will support the concept of "virtual nodes" where each processor is seen as its own node.
- It is too early in the project to determine the outcome of this effort, and therefore too early for us to give performance observations.

# Overview - the Profiler

- Sigportals
- The Debugger
- NQS
- OpenMP
- ***<u>The Profiler</u>***
- CTH Profiler Example
- IO Cliff Notes
- Math libraries

# Profiler Introduction

1. Features
2. Using the profiler
3. Notes
4. Manual Profiling

# Profiler Overview

- Get cpu time/memory usage/hw counters by routine/line/node/job
- Easy to use: compile and link with one of:

```
-Mprof=func
-Mprof=lines
```

- Run the job as usual.
- Easy to get profile results: just run ‘`profile`’

# Profiler Features
# Cpu time/HW counters

- Low overhead: 1-10% overhead depending on features
- Precise: not based on sampling
- Function-level and basic-block-level profiling (R2.7)

      -Mprof=func for function level profiling

      -Mprof=lines profiles at the 'basic block' level
- CPU time and hardware (HW) counters. (R2.7)
  - Always get cpu time for function level-profiling. Elapsed time and self-time (elapsed minus profiled children).
- Optionally specify 2 HW counters to monitor at function level.(R2.7)

      setenv PROFILE_COUNTERS PP_FLOPS,PP_RESOURCE_STALLS
- Get 'self' and elapsed counter values by routine. (R2.7)
- Optionally specify that one of the counters be used for line level profiling instead of using cpu time. (R2.7)

      setenv PROFILE_COUNTERS PP_FLOPS,LINE,PP_RESOURCE_STALLS
- List of HW counters with descriptions is on the web:
  - http://www.sandia.gov/ASCI/Red/usage/perfeva.htm

# Profiler Features
# Memory usage/Env. variables

- Stack and Heap usage (R2.8)

  Always get max stack and heap usage. Profiler reports max used, node and routine that hit the value. You can use the debugger to stop your application when the high water mark is hit.

  – If you want more memory usage data, you can monitor malloc/free by setting the environment variable PROFILE_WATCH_HEAP. See below for more on this feature.

- Environment Variables

  – PROFILE_TURN_OFF: turns off profiling (R2.8)

  – PROFILE_START_WITH_ROUTINE routine_name (R2.7)

    Profile only routine_name and its children. Permits timing of 'solver' while excluding startup/shutdown.

  – PROFILE_COUNTERS hw_counter0[,LINE][,hw_counter1] (R2.7)

    Specify HW counters for function profiling (and line-level profiling optionally). See /usr/include/perfmon.h for list of counters.

# Profiler Features
# More Env. Variables

- Env variables (cont.)
  - PROFILE_WATCH_HEAP (R2.8)

    If this variable is found, the profiler collects total bytes malloc'd and free'd by routine, net bytes malloc'd/free'd by self and children, high heap watermark, and fragments. Can't be used with PROFILE_COUNTERS. Use 'profile -m …'.
  - PROFILE_GSYNC_ROUTINE routine_name (R2.8)

    Insert global sync()s before and after routine_name. Report time each node spends waiting at each gsync(). All nodes must execute routine (due to the gsyncs). This helps determine load balancing by node. Use 'profile -g'
  - PROFILE_COMMENT comment_string (R2.7)

    comment_string will be written into profile data sets.
  - PROFILE_DIR directory_name (R2.7)

    Write profile datasets to directory_name. Default subdirectory is 'pmon.out'.

# Profiler Features
# Call Tree/Ancestor Tree/Load Bal.

- Call tree (R2.7)
  - For each called routine, show how much of the cpu time is due to each caller. Use ‘profile -C …’.
- Ancestor tree (R2.8)
  - Invert the call tree. Still a ‘beta’ feature. Use ‘profile -A …’
- Load balancing (R2.7)
  - Use ‘profile -b …’ get high/low/ave/dev. Cpu time for each routine by node.

# Profiler: Compiling/Running/Getting Results

- How to compile
  - Use '-Mprof=func' for function level profiling.
  - Use '-Mprof=lines' for basic block level profiling.

    If you are compiling with '-O2', then a basic block might be one to hundreds of source code lines. This option will slow performance maybe by 2x.

    If you compile with '-O0', the basic block is 1 line and performance will really suffer overheads, but the relative line number performance should be correct.
- How to run
  - Run the job as usual. Output files are put in subdirectory pmon.out in the current directory by default.
- How to get results
  - Run 'profile' (/usr/bin/profile).
  - Run 'profile -h' for options.

# Profiler example
# exam1.c - the basics on 1 node

- `janus> cat exam1.c`

```
void sub2() { sleep(10); }
void sub1() { sub2();sleep(4); }
int main() {
  sub1();
  sleep(2);
  return 0;
}
```

- `janus> pgcc -cougar exam1.c -Mprof=func -o tstc`

```
Linking:
```

- `janus> yod -sz 1 tstc`

# Profiler example
# exam1.c - Profile output

```
janus> profile -C
      Profile Data Summary
Code:    tstc
Last Mod:  Mon Sep 13 13:15:33 1999   Size: 392216
Run on:    Mon Sep 13 13:27:38 1999   CWD:  /Net/usr/home/pfay
nodes:    1    proc mode: 0   elapsed time:    16.00 wall_sec.
Max stack used:          32   Max stack avail:     2097152
Node/Rtn which hit max stack: 0000,sub2
Max heap used:       112800   Node/Rtn which hit max heap: 0000,sub2
Elapsed time=   16.000068 (node secs)  for start with routine= main

Routine              #calls     Time          Time      Cumu.  File
                                (sec)          (%)      Time
--------------------------------------------------------------------
sub2                      1      10.00002   62.500    10.00 exam1.c
sub1                      1       4.00002   25.000    14.00 exam1.c
main                      1       2.00003   12.500    16.00 exam1.c
```

# Profiler Example
# exam1.c - Call Tree

```
janus> profile -C
         Profile Data Summary
...
                         Call Tree Data

Routine                              #calls    Time      Self        Cumu.
     Called by                               per call    Time        Time
-------------------------------------------------------------------------
sub2
     sub1                                1   10.00002      10.000      10.000

sub1
     main                                1    4.00002       4.000      14.000

main
```

# Profiler Example
# exam1.c - Running on 2 nodes

```
Now on 2 nodes:
janus> yod -sz 2 tstc
janus> profile -C

         Profile Data Summary
Code:    tstc
Last Mod:  Mon Sep 13 13:15:33 1999   Size: 392216
Run on:    Mon Sep 13 13:29:00 1999   CWD:  /home/pfay/ark/pres/prof_sam
nodes:    2    proc mode: 0   elapsed time:    16.00 wall_sec.
Max stack used:          32   Max stack avail:     2097152
Node/Rtn which hit max stack: 0001,sub2
Max heap used:       902912   Node/Rtn which hit max heap: 0000,sub2
Elapsed time=   32.000122 (node secs)  for start with routine= main

Routine             #calls     Time         Time      Cumu.  File
                               (sec)         (%)      Time
-----------------------------------------------------------------------------
sub2                      2      20.00002  62.500    20.00 exam1.c
sub1                      2       8.00005  25.000    28.00 exam1.c
main                      2       4.00005  12.500    32.00 exam1.c
```

# Profiler Example
# exam1.c - Call Tree for 2 nodes

And the call tree:

```
 Call Tree Data

Routine                             #calls    Time      Self        Cumu.
     Called by                                per call    Time        Time
-------------------------------------------------------------------------
sub2
     sub1                                2   10.00001      20.000      20.000

sub1
     main                                2    4.00003       8.000      28.000

main
```

# Profiler Example exam2.c - Using the HW Counters to Gets Flops by Routine

Now for a hw counter example:

```
janus> setenv PROFILE_COUNTERS PP_FLOPS
janus> cat exam2.c
double x=0.0;
void add_to_x(){ int i; for(i=0;i<100;i++){x +=1.0;}}
void sub2() { add_to_x(); }
void sub1() { sub2();add_to_x(); }
int main() {
  sub1();
  add_to_x();
  return 0;
}
janus> pgcc -cougar exam2.c -Mprof=func -o tstc
```

# Profiler Example
# exam2.c - Display the flops

```
janus> yod -sz 1 tstc
janus> profile -p
         Profile Data Summary
Code:    tstc
Last Mod:Mon Sep 13 13:4  Size:           392487
Run on:  Mon Sep 13 13:4  cwd:  /Net/user/home/pfay
nodes:    1    proc mode: 0   elapsed time:    0.000 wall_sec.
Max stack used:          32   Max stack avail:     2097152
Node/Rtn which hit max stack: 0000,add_to_x
Max heap used:       112800   Node/Rtn which hit max heap: 0000,add_to_x
Elapsed time=       5e-05 (node secs)  for start with routine= main

Routine             #calls          Time    Time   Cumu.       PP_FLOPS
                                   (sec)     (%)   Time          (self)(self+chld)
-------------------------------------------------------------------------------------
sub2                     1       0.00002  32.000     0.00             2      103
main                     1       0.00002  30.000     0.00             3      311
add_to_x                 3       0.00001  26.000     0.00           303      303
sub1                     1       0.00001  12.000     0.00             3      207
```

# Profiler & Load Balance Sample Code

```
#include "mpi.h"
void wait1(int nsec)
{
       sleep(nsec);
}
void wait2(int nsec,int max)
{
        int shortwait, longwait;
        shortwait = 10;
        longwait = 45;
        if (nsec == 0) {
                sleep(0);
        }else if (nsec == (max-1)) {
                sleep(longwait);
        } else {
                sleep(shortwait);
        }
}
void dummy1()
{
         MPI_Barrier(MPI_COMM_WORLD);
}
void dummy2()
{
         MPI_Barrier(MPI_COMM_WORLD);
}
```

```
void main(int argc, char **argv)
{
        int rank,size;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank( MPI_COMM_WORLD, &rank );
        MPI_Comm_size( MPI_COMM_WORLD, &size );
        wait1(rank);
        dummy1();
        wait2(rank,size);
        dummy2();
}
```

- **wait1() takes 'rank' sec per node**
- **dummy1 is just a sync**
- **wait2() is more complicated**
  - node 0 takes essentially zero time
  - node "size-1" takes 45 sec.
  - Others take 10

# First Look with Profiler,
## Basic and PROFILE_GSYNC_ROUTINE

- Profile on 8 nodes

| Routine | #calls | Time (sec) | Time (%) | Cumu. Time | File |
|---|---|---|---|---|---|
| dummy2 | 8 | 255.00167 | 61.126 | 255.00 | waittest.c |
| wait2 | 8 | 105.00035 | 25.169 | 105.00 | waittest.c |
| dummy1 | 8 | 28.00116 | 6.712 | 28.00 | waittest.c |
| wait1 | 8 | 28.00026 | 6.712 | 28.00 | waittest.c |
| main | 8 | 1.17210 | 0.281 | 417.18 | waittest.c |

- So it looks like dummy1 and dummy2 are significant, from this
- use PROFILE_GSYNC_ROUTINE dummy2

| Routine | #calls | Time (sec) | Time (%) | Cumu. Time | File |
|---|---|---|---|---|---|
| main | 8 | 256.18515 | 61.407 | 417.19 | waittest.c |
| wait2 | 8 | 105.00037 | 25.169 | 105.00 | waittest.c |
| dummy1 | 8 | 28.00110 | 6.712 | 28.00 | waittest.c |
| wait1 | 8 | 28.00026 | 6.712 | 28.00 | waittest.c |
| dummy2 | 8 | 0.00231 | 0.001 | 0.00 | waittest.c |

# Looking at wait1() & wait2()

- No PROFILE_GSYNC_ROUTINE
- run on 64 nodes; profile -b

```
                nodes   Time                          Calls
--------------------------------------------------------------------------------
Routine                |  Min       Max       Avg.      |  Min.     Max.  Avg.
                       |  node      node      sigma     |  node     node  sigma
                       |                      frac      |                 frac
--------------------------------------------------------------------------------
dummy2             64     0.000    45.000    34.610         1        1        1
                             63         0     4.533        63       63      0.0
                                              0.969                       0.000

dummy1             64     0.000    63.000    31.500         1        1        1
                             63         0    18.473        63       63      0.0
                                              0.563                       0.000

wait1              64     0.000    63.000    31.500         1        1        1
                              0        63    18.473        63       63      0.0
                                              0.563                       0.000

wait2              64     0.000    45.000    10.391         1        1        1
```

# Interpreting the Output

- All routines are called once per node.
- Wait1()
  - — consumes 0 seconds on node 0, and 63 seconds on node 63
  - — avg time spent in wait1 over all nodes is 31.5 sec
  - — width (rms) of distribution of times is 18.47
  - — only about half of the nodes are within one rms of the avg

  *so the times for wait1 are all over the place*
- wait2()
  - — consumes 0 seconds on node 0, and 45 seconds on node 63
  - — avg time spent in wait2 over all nodes is 10.391 sec
  - — width (rms) of distribution of times is 4.53
  - — 97% of nodes are within one rms of the avg.

  *so the times for wait2 are clustered around 10 sec, with a couple of outliers*

# Profiler Notes - 2nd CPU and Useful HW Counters

- The 2nd CPU is 'handled' by not allowing the 2nd cpu to enter the profiler 'hook' routines. If you've compiled with OMP (pgcc -cougar -mp …) and you've used proc 2 mode (yod -proc 2 -sz …), the profiler doesn't collect any data for the 2nd CPU. You will have to infer that the change in cpu time for your OMP routine is due to the 2nd processor. The profiler is thread safe.
- Useful HW counters:
  - PP_FLOPS the number of flops (more or less)
  - PP_FP_ASSIST Usually corresponds to number of NANs
  - PP_CYCLES_DIV_BUSY cpu cycles that the divide unit was busy. Divides are slooow. Try to replace with inverse operation.
  - PP_MISALIGN_MEM_REF Misaligned memory references can really slow down code. Unfortunately, many cougar message passing routines generate misaligned references.
  - PP_RESOURCE_STALLS Number of cycles the CPU was blocked waiting on some resource, usually memory. This is frequently 80-90% of total cpu time but if a certain loop has a very high stall count,then perhaps the loop indices can be swapped to access memory in a more efficient manner.
  - PP_BUS_SNOOP_STALL Number of CPU clock cycles that the CPU was stalled fetching data out of the other CPU's cache. Useful for OMP optimization.
  - PP_DATA_MEM_REFS The number of loads and stores to registers

# Manual Profiling - genperf and perfrep

- Beginmflops, etc.. These routines don't handle the hw counter overflow properly so if you get more than 2^30 events, the counters will wrap.
- There are a set of routines similar to the begin/end/print series that handle the wrap around properly. These are the genperf routines.
- Genperf can be simpler if you just want to collect something like flops for a loop or a subroutine.
- The list of genperf routines are on the web:

  http://www.sandia.gov/ASCI/Red/usage/genperf.html
- Unlike using the profiler with -Mprof=func, you need to link in libperfmon.a (so add '-lperfmon' to the link command)
- We have a routine to globally block and return the number of flops on all nodes and processors: beginflopmon_(), endflopmon_(), printflopmon_()
- Genperf has cache info routines, flops, branch prediction and much more.
- Perfrep: These routines are designed to iterate over all hw counters, then generate a table of values. Useful for examining kernels in detail, or looking at sections of code within an application that is visited many times with similar data.  It does handle overflow and it prints out some additional computed quantities to assist optimization.

# Overview - CTH Profiler Example

- Sigportals
- The Debugger
- NQS
- OpenMP
- The Profiler
- ***<u>CTH Profiler Example</u>***
- IO Cliff Notes
- Math libraries

# Another Profile Example -- CTH

- Where to start? (Pretend we don’t know the answer)
  - Pick representative dataset (efp3d.in, 8 nodes)
  - compile with -Mprof=func
  - run it.

```
 Profile Data Summary
Code:    /scratch/tmp_4/bhcole/cth/bin/tf/mpicth
Last Mod:  Mon Sep 13 09:01:56 1999   Size: 5683742
Run on:    Mon Sep 13 09:36:33 1999   CWD:  /scratch/tmp_4/bhcole/cth/pr
nodes:    8  proc mode: 0   elapsed time:    233.6 sec.

Routine             #calls     Time        Time      Cumu.  File
                               (sec)        (%)      Time
----------------------------------------------------------------------
cthsnd                 2412     115.57845   6.893   115.58 mpmpi.F
erpy                   4800     100.93678   6.020   234.01 erpy.F
elsg                   4880      97.40255   5.809   101.36 elsg.F
ermz                   4880      94.42748   5.631   160.52 ermz.F
cthrcv                 2412      91.38929   5.450    91.39 mpmpi.F
bdsnde               143664      74.62850   4.451   257.35 bdsnde.F
```

# What do we learn from this?

- Messaging calls
- erpy, elsg.  -- know there's a lot of meat in there. OMP candidate if cache-friendly.
- Performance counters:
  - PP_DATA_MEM_REFS: all memory references (memory or cache)
  - PP_L2_LINES_IN:  Number of cache lines (4 doubles or 8 floats each) that had to come in from memory.
- Set env. variable PROFILE_COUNTERS PP_DATA_MEM_REFS,PP_L2_LINES_IN

```
nodes:    8  proc mode: 0   elapsed time:    228.6 sec.
```

| Routine | #calls | Time (sec) | Time (%) | Cumu. Time | PP_L2_LINES_I (self) | PP_L2_LINES_I (self+chld) | PP_DATA_MEM_REF (self) | PP_DATA_MEM_REF (self+chld) |
|---|---|---|---|---|---|---|---|---|
| cthsnd | 2412 | 116.20015 | 6.859 | 116.20 | 698677 | 698677 | 11392211209 | 11392211209 |
| erpy | 4800 | 104.25719 | 6.154 | 238.87 | 113242733 | 196167754 | 13888374735 | 43360398351 |
| elsg | 4880 | 97.86985 | 5.777 | 101.92 | 99701007 | 100295819 | 14885584301 | 15550245866 |
| ermz | 4880 | 96.59638 | 5.702 | 163.15 | 181399829 | 274772956 | 14452980462 | 23422777400 |
| cthrcv | 2412 | 91.05389 | 5.374 | 91.05 | 272221 | 272221 | 9424437784 | 9424437784 |
| bdsnde | 143664 | 74.73297 | 4.411 | 257.60 | 194261860 | 495293743 | 3584845611 | 18637630975 |

- Compute fraction of data from memory:

  (4*PP_L2_LINES_IN)/(PP_DATA_MEM_REFS)
- ERPY: 1.8% from memory -- good candidate!!
- Go through, put the OMP directives in.

  <erpy is rezone calculation in ydirection; one big main loop over columns. There's a lot of work in each iteration of the main loop.>

# Taking the next step… More OMP

- Add OMP Directives to ERPY, debug it, etc.
- Run it w/ –proc 2 and profile again:

```
Profile Data Summary
Code:    /scratch/tmp_4/bhcole/cth/bin/tf/mpicth
Last Mod:  Mon Sep 13 09:54:45 1999   Size: 5686754
Run on:    Mon Sep 13 09:59:12 1999   CWD:  /scratch/tmp_4/bhcole/cth/pr
nodes:    8  proc mode: 2   elapsed time:    215.1 sec.

Routine              #calls     Time         Time     Cumu.  File
                                (sec)         (%)     Time
----------------------------------------------------------------------------
cthsnd                 2412     111.39164   7.117   111.39 mpmpi.F
elsg                   4880      97.56098   6.233   101.56 elsg.F
ermz                   4880      94.81277   6.057   161.03 ermz.F
cthrcv                 2412      86.53987   5.529    86.54 mpmpi.F
bdsnde               143664      74.76909   4.777   253.32 bdsnde.F
erpx                   4800      65.66172   4.195   172.18 erpx.F
erpy                   4800      58.99054   3.769   128.30 erpy.F
```

- Time spent in ERPY is down from 234.01 node-sec to 128.3 node-sec: a parallel efficiency of 91% for this routine.
- Overall elapsed time is not that much impacted from this single routine (233 sec to 215 sec.)
- *BUT* if you press on….

# CTH w/ "full" OMP utilization

```
Profile Data Summary
Code:    /scratch/tmp_4/bhcole/cth/bin/tf/mpicth
Last Mod:  Mon Sep 13 10:24:21 1999   Size: 5557879
Run on:    Mon Sep 13 10:34:21 1999   CWD:
   /scratch/tmp_4/bhcole/cth/pr
nodes:    8  proc mode: 2   elapsed time:    160.1 sec.
```

| Routine | #calls | Time (sec) | Time (%) | Cumu. Time | File |
|---|---|---|---|---|---|
| bdsnde | 143664 | 74.48228 | 6.664 | 203.23 | bdsnde.F |
| cthrcv | 2412 | 70.36390 | 6.296 | 70.36 | mpmpi.F |
| ermz | 4880 | 61.95159 | 5.543 | 98.43 | ermz.F |
| cthsnd | 2412 | 61.16924 | 5.473 | 61.17 | mpmpi.F |
| elsg | 4880 | 59.70557 | 5.342 | 61.87 | elsg.F |
| erpy | 4800 | 58.56535 | 5.240 | 127.19 | erpy.F |

- Run time is now down from 233 to 160 seconds!!!
- 1.45x speed-up

# Overview - IO Cliff Notes

- Sigportals
- The Debugger
- NQS
- OpenMP
- The Profiler
- CTH Profiler Example
- ***<u>IO Cliff Notes</u>***
- Math libraries

# Optimizing IO

1. Rules for best IO performance
2. File systems
3. Synchronous IO APIs to async IO
4. Some results
5. Sample sync IO -> async IO

# Rules for best IO performance

- Don't do more IO than necessary
- Use a good IO library if available
- Don’t read/write files from NFS (your home dir)
- Know your IO parameters:
  - Reading or writing < 256Kbytes per IO, or using ASCII format? Use UFS (and get a few MB/s). /ufs…, /scratch/…
  - If you can use a request size >= 256 Kbytes and binary file format, then use PFS (/pfs…).
- Only PFS supports file size >= 2GB
- Open/closes are expensive; try to keep files open if possible.

# IO Bottleneck - fyods

- Major bottleneck: the fyods.
  - All cougar IO goes to the TOS fyod (file yod) which actually does the IO. The fyod is the link between cougar and the disk.
  - Default is 128 nodes/fyod
  - sync IO: only 1 sync IO per fyod at a time. Fyod waits till request is done before starting another request. Most IO requests are synchronous. Open/close/seek/fstat/read/write/fopen/fclose/fcntl/fread.
  - If you can use PFS, then use the asynchronous IO API's. See the examples below. Async calls fwrite/cread/cwrite/iread/iwrite/_iread/_iwrite/ ireadoff/iwriteoff/iodone/iowait/_cread/_cwrite
  - Async IO: fyod posts each request and looks for work

# Rules for Stream IO and Shared File IO

- If you are using streams (fread/fwrite)
  - consider first using cread/cwrite
  - use setvbuf() and get a 1MB buffer.
  - fread is sync (and slooow),
  - fwrite async. Fast if request size >= 256K or you are using a big buffer (and not flushing it till it is full)
  - If the request size >= buffer size, then bypass the buffer.
  - Doing a seek flushes buffer.
- If multiple nodes are writing to the same file
  - use 'yod -masync ...', ireadoff() and iwriteoff().
- Much more detail available in IO report:
  - http://www.sandia.gov/ASCI/Red/usage/ioreport.ps
  - some parts out of date

# File systems -
# NFS (slowest) & UFS (still slow)

- NFS is very, very slow. This is your home directory and anything starting with /Net/... Avoid reading/writing to it.
- UFS is better.
  - Use UFS for small read/write request sizes (< 256Kbytes).
  - UFS files are buffered in TOS so if you have small files that you read/write over and over again, UFS may be better than PFS.
  - UFS files must be < 2GBytes.

# File systems
# PFS file system

- PFS (Parallel File System) is optimized for:
  - Large amounts of data (files can be > 2 Gbytes)
  - Big read/write request size
    - Need at least 256Kbytes
    - Requests size of 2MByte is near optimal
  - Many nodes doing IO simultaneously (if you are using the async IO API)
  - PFS is /pfs_grande/
- Much more detail available in IO report:
  - http://www.sandia.gov/ASCI/Red/usage/ioreport.ps

# C and Fortran IO APIs

- C/C++
  - read()/write() are synchronous. Use cread,cwrite instead.
  - fread is synchronous, fwrite is asynchronous. Use setvbuf and avoid implicit buffer flushes (due to seek).
- Fortran
  - All F77 IO maps to streams with an 8K buffer and is terrible for PFS
    - F77 read maps to fread and is synchronous
    - F77 write maps to fwrite and is asynchronous
  - F90 read/write maps to streams (same as F77) and is terrible for PFS.
  - Use cread/cwrite/ireadoff/iwriteoff instead (only available for f90). All the 'C' async IO routines are available to f90. Use the f90 open unit number as the file descriptor. See 'man 3f cwrite'

# IO Results Impacted by Many Factors

- Your results WILL vary (a lot).
  - The results below are for a janus with lots of other jobs running. Many factors impact IO performance including:
    - what else is running in the service partition
    - how busy the disks are due to other users
    - number of fyods
    - number of service nodes
    - the API used
    - the request size used

# IO Results - C Streams

- C streams

  64 nodes/36 raids (30 MB/s max each raid)/ 2MB request/non-dedicated system/1 file per node/pfs_grande

  fwrite() without setvbuf: 241 MB/sec (low=70,hi=417)

  - Note that since fwrite req_size=2MB, the buffer is bypassed. Performance is the same as cwrite()

  fread() without setvbuf: 2.15 MB/sec (low=1,hi=4.3)

  fwrite() with setvbuf: 311MB/sec (low=25, hi=557)

  fread() with setvbuf: 54 MB/sec (low=25, hi=104)

  Default buffer is 8K (terrible for PFS). Setvbuf buffer is 2MB.

# IO Results - C
# write/read/cwrite/cread

- C stdio

  64 nodes/36 raids (30 MB/s max each raid)/ 2MB request/non-dedicated janus/1 file per node/ pfs_grande/multi

  write(): 17.0 MB/s

  read(): 33 MB/s

  cwrite(): 388 MB/s (low=121,hi=611)

  cread(): 523 MB/s (low=366,hi=782)

# IO Results - C Shared File - Without 'yod -masync'

- C and shared file without 'yod -masync '
  - 128 nodes/6 raids (30 MB/s max each raid)/ 1MB request/dedicated system/1 shared file
  - without the yod -masync parameter: You get about 17 MB/s independent of API.

# IO Results: C Shared File with 'yod -masync' and Best API

- C and shared file with 'yod -masync'
  - 128 nodes/36 raids (30 MB/s max each raid)/ 2MB request/non-dedicated janus/1 shared file
  - with the yod -masync parameter:
    - eseek/write()/1 fyod: 16.8 MB/s, with 8 fyods: 105 MB/s
    - eseek/read()/ 1 fyod: 17.2 MB/s, with 8 fyods: 135 MB/s
    - eseek/cwrite()/1 fyod: 70.2 MB/s, with 8 fyods: 390 MB/s
    - eseek/cread()/1 fyod: 17.2 MB/s, with 8 fyods: 320 MB/s
    - iwriteoff()/1 fyod: 71.1 MB/s, with 8 fyods: 550 MB/s
    - ireadoff()/1 fyod: 125.8 MB/s, with 8 fyods: 720 MB/s

# IO Results: F77 read/write

- F77
  - 64 nodes/pfs_grande/ 1MB request/non-dedicated system/1 file per node
  - All F77 IO maps to streams (and fread is synchronous) and uses an 8Kbyte buffer (which is a disaster for PFS).

    F77 read/write: 1-5 MB on pfs_grande

# IO Results: F90 using Async IO

- F90
  - 64 nodes/pfs_grande/ 1MB request/non-dedicated system/1 file per node
  - F90 can access the async IO library. Use the F90 unit number as the file descriptor
    - cwrite ran at 200-500 MB/sec (2 runs)
    - cread ran at 250-770 MB/sec (2 runs)

# Sample programs using sync IO and async IO

- The previous results were generated by the sample programs below.
  - streams.c
  - stdio.c
  - shared.c
  - 1file_per_node.f90
  - shared.f90
- Also found on the web at
  - http://www.sandia.gov/ASCI/Red/usage/io.html

# Overview - Math Libs

- Sigportals
- The Debugger
- NQS
- OpenMP
- The Profiler
- CTH Profiler Example
- IO Cliff Notes
- ***Math Libraries***

# ASCI Red Math Libraries

What Libraries Exist

Libcsmath

Libwc

Others

# What Math Libraries Exist

- Libcsmath ("Comp-Sci MATH")
  - Level 1, 2, and 3 BLAS
  - 1D FFTs
  - Partial Man pages in R2.8!
- LAPACK
- BLACS
  - NX (An integer sum bug has been fixed in R2.8)
  - MPI
- libwc (write-combine Cougar libraries)
- ScaLAPACK (Parallel Linear Alegra Package)
- PBLAS (Parallel BLAS)

# LIBCSMATH

- Level 1, 2, and 3 BLAS.  1D FFTs
- /usr/lib/libcsmath_r.a, /usr/lib/libcsmath_cop.a, /usr/lib/libcsmath.a
  - _r: Tries to split the BLAS/FFTs for you using the compiler
  - _cop: Tries to split the BLAS/FFTs for you using “cop”
  - All versions reentrant except on some level-2,level-3 complex BLAS
  - If you do your own parallelism, you will want to explicitly use libcsmath.a.
  - The official C interface to the BLAS is included.
- Linking with -mp or -Mconcur automatically gives you _r.a.
- Dual processor versions are enabled with -proc 2 on the yod line
- $TFLOP_XDEV/tflops/lib
- See the release notes and upcoming man pages (R2.8)
- Works with OSF and Cougar
- http://www.cs.utk.edu/~ghenry/distrib
  - Can find a linux version here (around 16000 licenses.)
- Other interesting kernels:
  - copsync()  ; xdgemm() ; transposition routines

# Libcsmath R2.8 enhancements

- C = A*B case where the number of columns of C (n) is small.
- All DGEMM cases where K is small
- K=24 GEMM cases
- K>64 GEMM cases
- GEMM cases where the number of rows (M) is 2.
- More prefetching done on columns of B
- Enhancements to other level-3 kernels
- Faster handling of smaller BLAS

# LIBWC

- Write Combine Library:
  - Using the “write combine” method of accessing memory as opposed to “write back”. Write combine buffers a single cache line and then writes it directly to memory, instead of loading it first into cache or keeping it in cache awhile like write back.
- Write Combine library for Cougar
  - Takes advantage of new Xeon core features
  - Applicable to any memory-write bound kernel.
  - Please contact us if you have a use for a tuned kernel of this nature.
- Three versions (like libcsmath), but only available on Cougar
  - The compiler does not automatically bring in one or another unlike libcsmath
- Link with -lwc and you must use -wc on the yod line
- libwc versions of: memcpy, dcopy, dzero, memset, memmove, bcopy
- Designed for large (1 Mbyte) memory writes.
- Other interesting kernels: (these can be used by anything!)
  - flush_caches() (flushes the caches on one or both processors)
  - use_write_combine() (returns 1 if it is safe to use write combine)
  - touch1( array, size_of_array_in_bytes) (C and Fortran versions)

# LAPACK, PBLAS, ScaLAPACK

- /usr/lib/scalapack or $TFLOP_XDEV/tflops/lib/scalapack
  - liblapack.a, libtmglib.a, libpblas.a, libscalapack.a, libtools.a, libredist.a
- ScaLAPACK and PBLAS depend on BLACS or BLACS_MPI
- Sample link lines:
  - L/usr/lib/scalapack -ltmglib -llapack
  - L/usr/lib/scalapack -lscalapack -lpblas -ltools -lredist
  - L/usr/lib/scalapack -lblacsF77init_MPI -lblacs_MPI -lblacsF77init_MPI -lmpi
  - L/usr/lib/scalapack -lblacsCinit_MPI -lblacs_MPI -lblacsCinit_MPI -lmpi
  - L/usr/lib/scalapack -lblacs -lnx
- Recent BLACS Integer sum bug fix found in release R2.8!

# A new Optimization Tool

- Not yet available: only in Alpha on Janus right now.
- Optimizes your (F77) subroutine by trying different optimization strategies and returning to you the assembly code corresponding to the optimal one.
- You must provide a "greg_timer()" and "greg_initialize()" routines and link against the library.
- The routine greg_timer() calls "opcode_routine()" instead of the target routine to be optimized.
- The application runs on Janus for anywhere from a minute to a day or more depending on input options.

# Example

```
    subroutine target_routine_to_optimize(A, B, C, M, N)
     double precision A(*), B(*), C(*), SUM1
     integer M, N, I, J
     sum1 = 0.d0
     do I = 1, M
            do J = 1, N
                   sum1 = sum1 + A((I-1)*M+J) + B((I-1)*M+J)
            enddo
            C(I) = sum1
     enddo
     return
     end
/* New auxillary routines */
#define ARRAY_SIZE 1024
int N=ARRAY_SIZE, M=ARRAY_SIZE;
double A[ARRAY_SIZE*ARRAY_SIZE];
double B[ARRAY_SIZE*ARRAY_SIZE];
double C[ARRAY_SIZE];
greg_initialize()
{
   int I;
   for ( I = 0 ; I < ARRAY_SIZE*ARRAY_SIZE ; I++ ) { A[I]=0.0; B[I]=0.0; }
   for ( I= 0 ; I< ARRAY_SIZE ; I++ ) C[I] = (double) I;
   flush_cache();
}
greg_timer()
{
   extern void (*opcode_routine)();
   (*opcode_routine)(A, B, C, &N, &M);
}
```