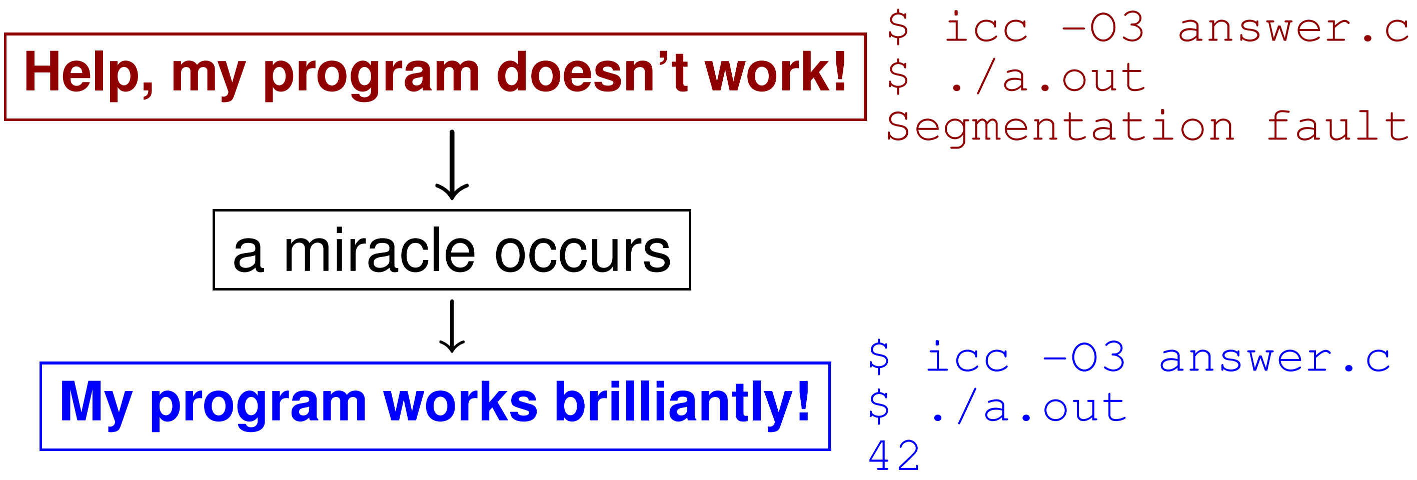
Debuggers and Parallel Debugging at SciNet: gdb, ddd, padb

SNUG TechTalk

SciNet, Toronto

Debugging basics

Debugging basics



Help, my program doesn't work!

```
$ icc -O3 answer.c
$ ./a.out
Segmentation fault
```

↓

a miracle occurs

↓

My program works brilliantly!

```
$ icc -O3 answer.c
$ ./a.out
42
```

- Unfortunately, “miracles” are not yet supported by SciNet.

Debugging:

Methodical process of finding and fixing flaws in software

Debugging basics

Ways to debug

- Don't write buggy code. YEAH, RIGHT.
- Add print statements NO WAY TO DEBUG!
- Command-based, symbolic debuggers
  - GNU debugger: gdb
  - Intel debugger command-line: idbc
- Symbolic debuggers with Graphical User Interface
  - GNU data display debugger: ddd
  - Intel debugger: idb
  - IDEs: Eclipse (not on SciNet), emacs/gdb

What’s wrong with using print statements?

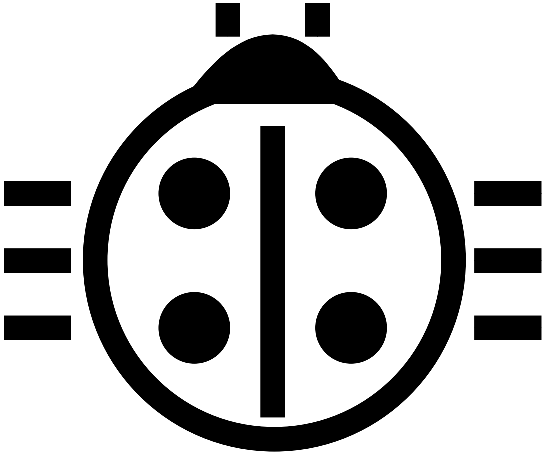
Print debugging

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output *bug not found?*
- Removing the extra code after the bug is fixed
- Repeat for each bug

Problems

- Time consuming
- Error prone
- Changes memory, timing. . . THERE’S A BETTER WAY!

Symbolic debuggers



Symbolic debuggers

Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

Use a graphical debugger or not?

- Local work station: graphical is convenient
- Remotely (SciNet):
  - Graphical debuggers slow
  - Graphics may not be available
  - Command-based debuggers fast (esp. gdb).
- Graphical debuggers still have command prompt.

Symbolic debuggers

Preparing the executable

- Required: compile with `-g`.
- Optional: switch off optimization `-O0`

Command-based symbolic debuggers

- gdb ⟵ Focus on this one
- idbc ⟵ Has gdb mode

```
$ module load intel
$ icc -g -O0 example.c -o example
$ module load gdb
$ gdb example
...
(gdb)_
```

gdb building blocks

GDB basics - 1 Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

- needs max core size set (`ulimit -c <number>`)
- gdb reads with `gdb <executable> <corefile>`
- it will show you where the program crashed

No core file?

- can start gdb as `gdb <executable>`
- type `run` to start program
- gdb will show you where the program crashed if it does.

GDB basics - 2 Function call stack

Interrupting program

- Press Crtl-C while program is running in gdb
- gdb will show you where the program was.

Stack trace

- From what functions was this line reached?
- What were the arguments of those function calls?

gdb commands

`backtrace`	function call stack
`continue`	continue
`down`	go to called function
`up`	go to caller

GDB basics - 3 Step through code

Stepping through code

- Line-by-line
- Choose to step into or over functions
- Can show surrounding lines or use `-tui`

gdb commands

`list`	list part of code
`next`	continue until next line
`step`	step into function
`finish`	continue until function end
`until`	continue until line/function

GDB basics - 4 Automatic interruption

Breakpoints

- `break [file:]<line>|<function>`
- each breakpoint gets a number
- when run, automatically stops there
- can add conditions, temporarily remote breaks, etc.

related gdb commands

`delete`	unset breakpoint
`condition`	break if condition met
`disable`	disable breakpoint
`enable`	enable breakpoint
`info breakpoints`	list breakpoints
`tbreak`	temporary breakpoint

GDB basics - 5 Variables

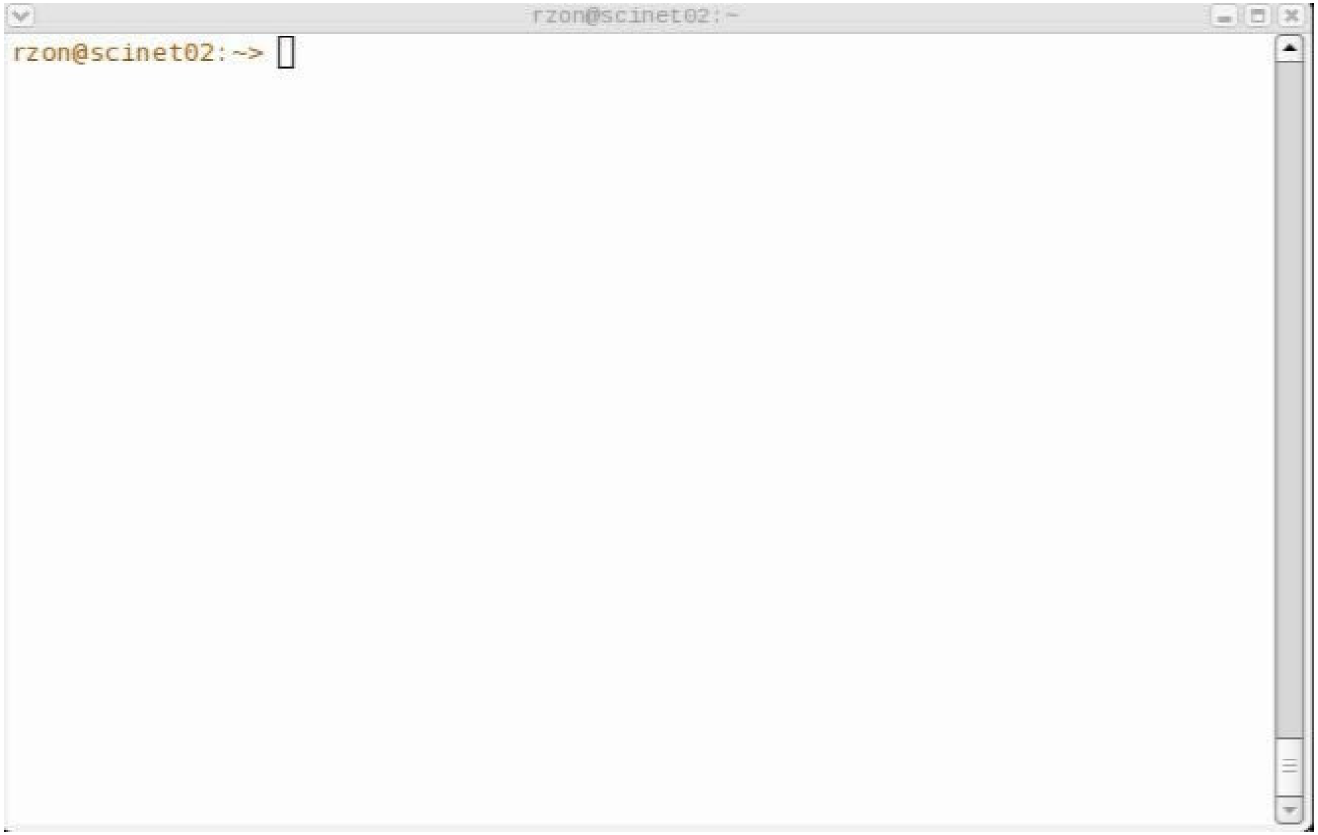
Checking a variable

- Can print the value of a variable
- Can keep track of variable (print at prompt)
- Can stop the program when variable changes
- Can change a variable (“what if . . . ”)

gdb commands

`print`	print variable
`display`	print at every prompt
`set variable`	change variable
`watch`	stop if variable changes

Demonstration gdb



```
rzon@scinet02:~> 
```

Graphical symbolic debuggers

Graphical symbolic debuggers

Features

- Nice, more intuitive graphical user interface
- Front to command-based tools: Same concepts
- Need graphics support: Problematic on compute nodes

Available on SciNet

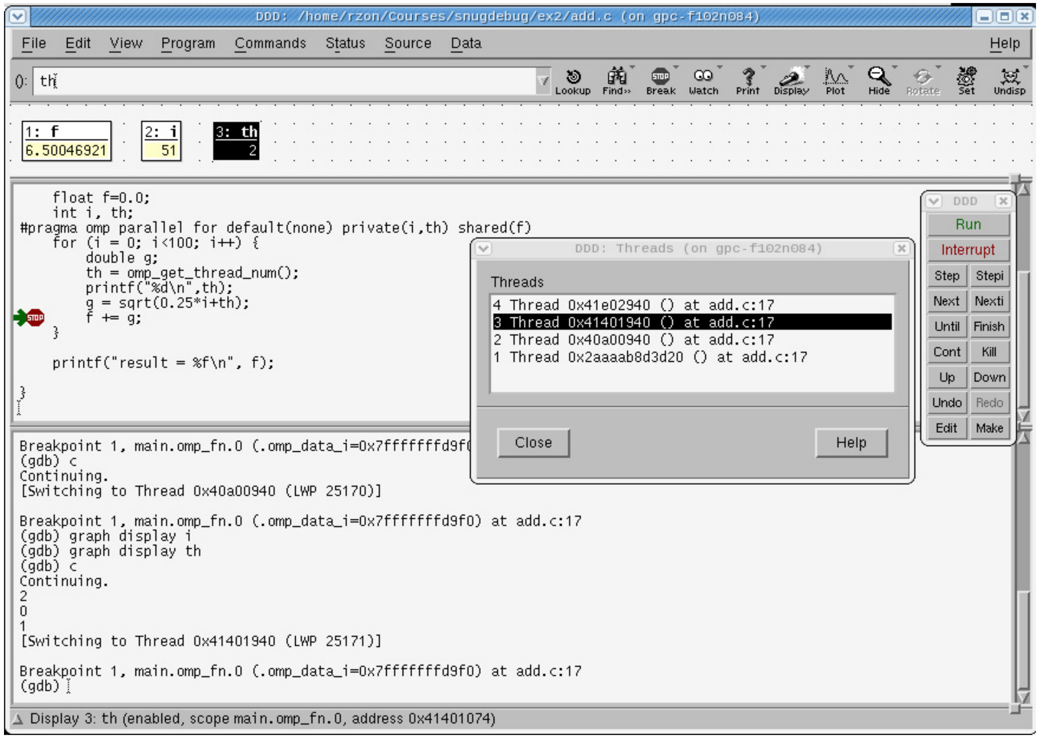
- ddd

```
$ module load gcc ddd
$ ddd <executable compiled with -g flag>
```

- idb

```
$ module load intel java
$ idb <executable compiled with -g flag>
```

Graphical symbolic debuggers - ddd



```
th
```

```
1: f        2: i     3: th
6.50046921    51         2
```

```
    float f=0.0;
    int i, th;
#pragma omp parallel for default(none) private(i,th) shared(f)
    for (i = 0; i<100; i++) {
        double g;
        th = omp_get_thread_num();
        printf("%d\n",th);
        g = sqrt(0.25*i+th);
        f += g;
    }

    printf("result = %f\n", f);

}
```

Threads

```
4 Thread 0x41e02940 () at add.c:17
3 Thread 0x41401940 () at add.c:17
2 Thread 0x40a00940 () at add.c:17
1 Thread 0x2aaaab8d3d20 () at add.c:17
```

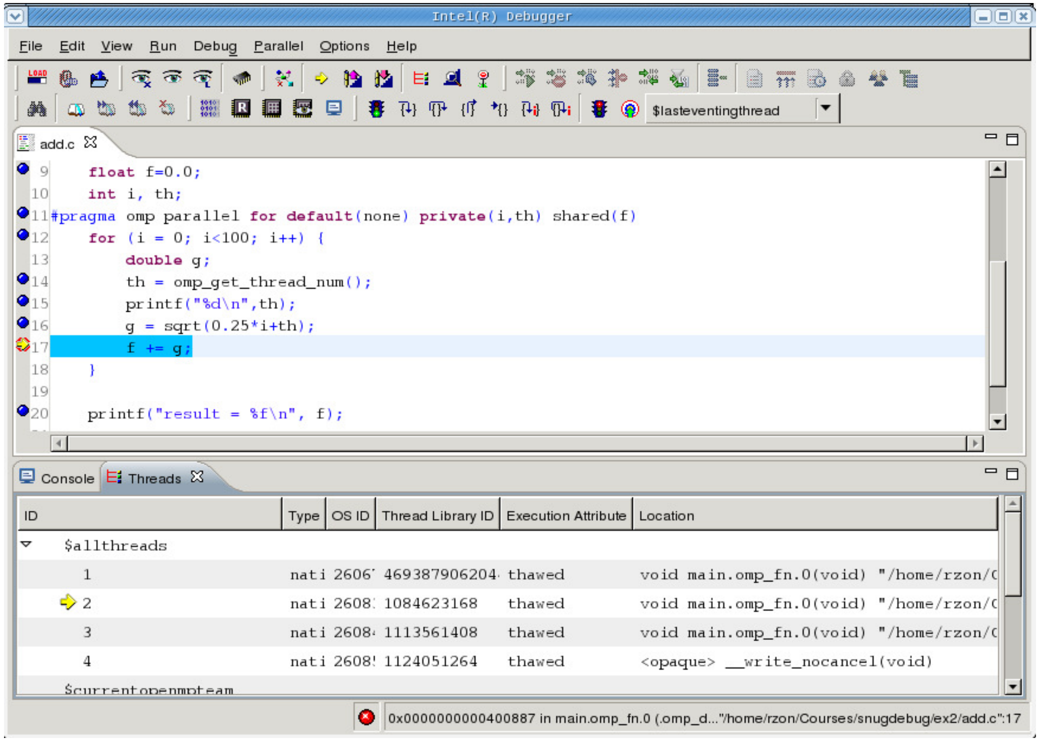
```
Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0
(gdb) c
Continuing.
[Switching to Thread 0x40a00940 (LWP 25170)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0) at add.c:17
(gdb) graph display i
(gdb) graph display th
(gdb) c
Continuing.
2
0
1
[Switching to Thread 0x41401940 (LWP 25171)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0) at add.c:17
(gdb)
```

Display 3: th (enabled, scope main.omp_fn.0, address 0x41401074)

compute • calcul
CANADA

SciNet

Graphical symbolic debuggers - idb

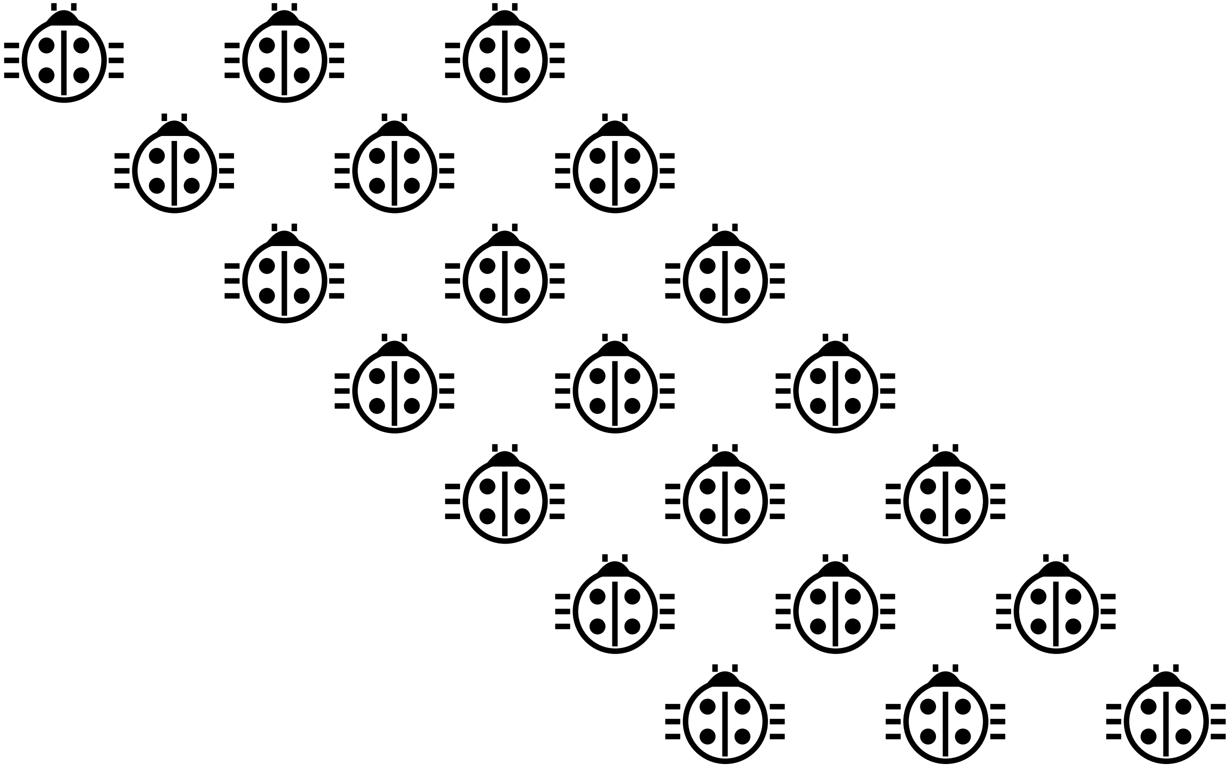


```
    float f=0.0;
    int i, th;
#pragma omp parallel for default(none) private(i,th) shared(f)
    for (i = 0; i<100; i++) {
        double g;
        th = omp_get_thread_num();
        printf("%d\n",th);
        g = sqrt(0.25*i+th);
        f += g;
    }

    printf("result = %f\n", f);
```

ID	Type	OS ID	Thread Library ID	Execution Attribute	Location
$allthreads					
1	nati	2606'	469387906204	thawed	void main.omp_fn.0(void) "/home/rzon/C
2	nati	2608:	1084623168	thawed	void main.omp_fn.0(void) "/home/rzon/C
3	nati	2608	1113561408	thawed	void main.omp_fn.0(void) "/home/rzon/C
4	nati	2608!	1124051264	thawed	<opaque> __write_nocancel(void)
$currentopenmpteam					

```
0x0000000000400887 in main.omp_fn.0 (.omp_d..."/home/rzon/Courses/snugdebug/ex2/add.c":17
```

Parallel debugging



Parallel debugging

- Challenge: Simultaneous execution
- Shared memory:
  OpenMP (Open Multi-Processing)
  pthreads (POSIX threads)
  - Private/shared variables

    USE GNU COMPILERS FOR DEBUGGING OPENMP!
  - Race conditions
- Distributed memory:
  MPI (Message Passing Interface)
  - Communication
  - Deadlock

Parallel debugging - 1 Shared memory

gdb and idbc

- Track each thread’s execution and variables
- OpenMP serialization: `p omp_set_num_threads(1)`
- Step into OpenMP block: `break` at first line!
- Thread-specific breakpoint: `b <line> thread <n>`

idbc only

- Freezing/thawing thread
- Native OpenMP serialization (requires Intel compiler)
- Graphical: `ddd --debugger idbc`

`info threads`	where is each thread?
`thread`	change thread context
`idb freeze/thaw t:[]`	suspend thread(s)

Parallel debugging - Race conditions

helgrind

To find race conditions:

```
$ module load valgrind
$ valgrind --tool=helgrind <exe> &> out
$ grep <source> out
```

where `<source>` is the name of the source file where you suspect race conditions (valgrind reports a lot more)

Parallel debugging - 2 Distributed memory

Multiple MPI processes

- Your code is running on different cores!
- Where to run debugger?
- Where to send debugger output?
- No universal (free) solution.

Good approach

1. Write your code so it can run in serial: perfect that first.
2. Deal with communication, synchronization and deadlock on smaller number of MPI processes.
3. Only then try full size.

Parallel debugging - 2 Distributed memory

padb

- Tool for debugging parallel mpi programs
- Requires openmpi and gdb:

```
module load gdb openmpi padb
```

Features

- Stack trace generation
- MPI Message queue display
- Deadlock detection and collective state reporting
- Process interrogation
- Signal forwarding/delivery
- MPI collective reporting
- Job monitoring

Parallel debugging - 2 Distributed memory

```
$ qsub -l nodes=1:ppn=8,walltime=1:00:00 -q debug -I
$ cd  /scratch/where_ever
$ mpirun -np 16 whatever
$ padb --all --stack-trace --tree
Stack trace(s) for thread: 1
-----------------
[0-15] (16 processes)
-----------------
main() at ?:?
  system_run() at ?:?
    compute_forces() at ?:?
      -----------------
      [8-15] (8 processes)
      -----------------
      IdVector_exchange() at ?:?
        PMPI_Sendrecv() at ?:?
          -----------------
          [8,10] (2 processes)
          -----------------
          ompi_request_default_wait() at ?:?
            opal_progress() at ?:?
          -----------------
          [9,11-15] (6 processes)
          -----------------
          mca_pml_ob1_send() at ?:?
            opal_progress() at ?:?
```

Advanced tricks

- You want #proc terminals with gdb for each process?
- Possible, but brace yourself!
- Small number of procs:
  - Start terminals: no x forwarding from compute nodes
  - Submit your job on scinet
  - Make sure its runs: checkjob -v
  - From each terminal, ssh into the appropiate nodes
  - Do `top` or `ps -C <exe>` to find process id (pid)
  - Attach debugger with `gdb -pid <pid>`.
  - This will interrupt the process (not for idbc).

Parallel debugging - 2 Distributed memory

Advanced tricks

Wait, so the program started already?

- Yes, and that's probably not what you want.
- Instead, put infinite loop into your code:

  ```
  int j=1;
  while(j) sleep(5);
  ```
- Once attached, go "up" until at while loop.
- do "set var j=0"
- now you can step, continue, etc.

Note: You can use padb to find ranks of process etc.

Useful references

- G Wilson Software Carpentry
  http://software-carpentry.org/3_0/debugging.html
- N Matloff and PJ Salzman
  The Art of Debugging with GDB, DDD and Eclipse
- Padb: http://padb.pittman.org.uk
- Wiki: https://support.scinet.utoronto.ca/wiki
- Email: support@scinet.utoronto.ca