SGI® Altix™
Application Programming

Reiner Vogelsang
SGI GmbH
reiner@sgi.com

February 13th, 2007
SGI ccNuma Balanced System Architecture
Parallel Architectures

Shared Memory (S.M.)

Easy to Program  Difficult to Scale
~ 32p

NUMA

Easy to Program  Scales Well
~ 1024p

Cluster

Difficult to Program  Highly Scalable
~ 4096p

Distributed Memory (D.M.)
SGI Scalable ccNUMA Architecture

Interconnect: section of interface chip, cables and routers
ccNuma: Distributed Shared Memory

• ccNuma:
  – Memory is physically distributed but logically shared
  – Memory is kept coherent automatically by hardware
  – Coherent memory: memory is always valid (caches hold copies)
  – Granularity is L3 cacheline (128 B)

• Directory memory:
  – For each cacheline access information is stored:
    – Who has valid copies
    – Which processor has write access
    – Hardware revokes access rights automatically

• In contrast snoopy bus protocols do not scale well
  – Access requests are broadcasted

• Directory information is stored in main memory
  – Directory entry is 4 byte wide for each 128 byte cache line
SGI Altix 4700 Processor Blade

**Top View**
- **Bandwidth Compute Blade**
  - Itanium2 Socket
  - Shub 2.0
  - DDR2 DIMM

**Front View**
- **Highest Memory BW, Performance: Bandwidth Compute Blade**
  - 667MHz FSB -> 10.7GB/s Local Memory Bandwidth
  - 32 Sockets / S-Rack
  - Memory Sizes: 2G – 24GB per blade

**Top View**
- **Density Compute Blade**
  - Itanium2 Socket
  - Shub 2.0
  - DDR2 DIMM

**Front View**
- **Best $/FLOP, Best Density: Density Compute Blade**
  - 533MHz FSB -> 8.524GB/s Local Memory Bandwidth
  - 64 Sockets / S-Rack
  - Memory Sizes: 2G – 24GB per blade
IRU Blockdiagramm
Standardized Blades, NUMAlink Backbone

Individual Rack Unit (IRU)
(Contains 10 Blades)

Rack
Small Rack = 4 IRUs
# Next Generation Reconfigurable Compute Technology

<table>
<thead>
<tr>
<th><strong>SGI® RASC™ RC100 Blade</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA</strong></td>
</tr>
<tr>
<td><strong>No. of FPGAs</strong></td>
</tr>
<tr>
<td><strong>Host System</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td><strong>I/O</strong></td>
</tr>
<tr>
<td><strong>Max Config</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Dimensions</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>O/S</strong></td>
</tr>
</tbody>
</table>

---

* with available 2 blade slot upgrade chassis
+ rack mounted version only

---

Product plans and information are preliminary and subject to change without notice.
Basic System – Single IRU

79.5 in. H × 25.8 in. W × 45

6 x 6.4 = 38.4 GB/s = 3.84 GB/s/blade
Single Rack

Hypercube topology within rack
128 Compute Blades

28 x 6.4 = 179.2 GB/s = 1.28 GB/s/blade
Montecito, Intel P9000

Itanium Dual Core: Montecito

<table>
<thead>
<tr>
<th>Montecito Feature Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simultaneous Threads</td>
</tr>
<tr>
<td>Process technology</td>
</tr>
<tr>
<td>L1 Cache</td>
</tr>
<tr>
<td>L2 Data</td>
</tr>
<tr>
<td>L2 Instruction Cache</td>
</tr>
<tr>
<td>L3 Cache (Unified)</td>
</tr>
<tr>
<td>Transistors</td>
</tr>
<tr>
<td>Availability Target</td>
</tr>
</tbody>
</table>
Montecito, Intel P9000

**Montecito – 4 contexts, 1 socket**

- **L1I Cache (16KB)**
- **Branch Prediction**
- **Instruction TLB**
- **Register Stack Engine / Re-name**
- **Branch & Predicate Registers**
- **Integer Registers**
- **Floating Point Registers**
- **Branch Unit**
- **Integer Unit**
- **Memory/Integer**
- **Floating Point Unit**
- **L1D Cache (16KB)**
- **ALAT**
- **Data TLB**
- **System Interface**
- **L2D Cache (256KB)**
- **L2I Cache (1MB)**
- **Queues/Control**
- **L3 Cache (12MB)**
- **Arbiter**
- **Synchronizer**

- **L1I Cache (16KB)**
- **Branch Prediction**
- **Instruction TLB**
- **Register Stack Engine / Re-name**
- **Branch & Predicate Registers**
- **Integer Registers**
- **Floating Point Registers**
- **Branch Unit**
- **Integer Unit**
- **Memory/Integer**
- **Floating Point Unit**
- **L1D Cache (16KB)**
- **ALAT**
- **Data TLB**
- **System Interface**
- **L2D Cache (256KB)**
- **L2I Cache (1MB)**
- **Queues/Control**
- **L3 Cache (12MB)**
- **Arbiter**
- **Synchronizer**
Montecito, Intel P9000

Montecito Core Extensions

Slightly extended Itanium® 2 processor core:
- Larger atomic ops: 16-byte ld16/st16/cmp8xchg16
  - Support non-blocking synchronization in database apps
  - Improves performance scalability of database applications on large SMP
- Instruction(s) to support virtualization
- Cash flush extensions (fc and fc.i)
- hint@pause for thread switching
  - Is a NOP on older architecture, will not fault
- Additional integer shifter and popcount
  - Allows scheduling two variable shifts per cycle
  - Enhanced processing performance in cryptographic codes
- Faster chk.a/chk.s resteer
- Support in compiler 9.0 via
  - intrinsics (not documented yet)
  - Introduction of new machine model (KNOBs file for Montecito)
Traditional Architectures: **Limited Parallelism**

- **Original Source Code**
- **Compile**
- **Sequential Machine Code**
- **Hardware**

- **Parallelized Code**
- **Multiple Functional Units**

Execution Units Available - Used Inefficiently

**Today’s Processors are often 60% Idle**
Intel® Itanium® Architecture: Explicit Parallelism

Original Source Code

Compile

Parallel Machine Code

Compiler Views

Itanium Architecture

Compiler

More efficient use of execution resources

Increases Parallel Execution
IA-64™ Instruction Bundles

1 instruction coded on 41 bits
3 instructions grouped into 1 bundle (128 bits)

Bundle type is specified through 5-bit template:

```
{ .mfi            // template (mem-fp-int)
  (p16) ld fd f39=[r2],16   // load fp, post-increment
  (p19) fnma.d.s0 f49=f42,f6,f45 // multiply Add
  (p16) adds r32=16,r33     }; // integer add immediate

{ .mib            // template (mem-fp-br)
  (p16) ld fd f42=[r33]   // load fp, post-increment
  (p16) adds r40=8,r33
  br.cstop.dptk.few .BB13_mp_ortho2_ ;; }; // counted loop branch
```
Application Porting and Getting Correct Code
Endianess

• The Intel IA64 as well as the rest of the Intel processor family is working with byte-wise little-endian address representation.
  – The number 1025 bit-wise represented and grouped in 4 bytes:

    00000000 00000000 0000100 00000001
    \^MSB \^LSB

  – Big Endian          Little Endian
    00  00000000          00000001
    01  00000000          00000100
    02  00000100          00000000
    03  00000001          00000000

  – In rare cases even bytes can be little-endian.
Endianess (cont.)

• Big endian systems are
  – SGI MIPS/Irix (Origin 3000, 2000,...)
  – HP PA Risc
  – Sun Sparc
  – IBM Power RISC
  – NEC vector systems, Cray vector systems

• To read/write big endian binary data you HAVE to set (Intel 9.x, 8.x and 7.x compilers):
  
  F_UFMTENDIAN=big (applies to all units)
  F_UFMTENDIAN=big:10,20 (applies to unit 10 and 20 only)
  or compile with
  -convert big (Intel 9.x and 8.x compilers)

(See Intel® Fortran Compiler Documentation, Building Applications, Chapter: Data and I/O)
Array Bound Checking

• ifort supports array bound checking and check for temporary argument creation:
  – Compile with -check all -traceback -ftrapuv -g

forrtl: severe (408): fort: (2): Subscript #1 of the array DIST has value 601 which is greater than the upper bound of 600

<table>
<thead>
<tr>
<th>Image</th>
<th>PC</th>
<th>Routine</th>
<th>Line</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>mat_dist</td>
<td>0x040000000000004f9b0</td>
<td>Unknown</td>
<td></td>
<td>Unknown</td>
</tr>
<tr>
<td>mat_dist</td>
<td>0x400000000000009f40</td>
<td>Unknown</td>
<td></td>
<td>Unknown</td>
</tr>
<tr>
<td>mat_dist</td>
<td>0x400000000000009890</td>
<td>Unknown</td>
<td></td>
<td>Unknown</td>
</tr>
<tr>
<td>mat_dist</td>
<td>0x400000000000006a20</td>
<td>dodist_</td>
<td>40</td>
<td>dodist.F</td>
</tr>
<tr>
<td>mat_dist</td>
<td>0x400000000000004880</td>
<td>MAIN__</td>
<td>86</td>
<td>main.F</td>
</tr>
<tr>
<td>mat_dist</td>
<td>0x4000000000000036d0</td>
<td>Unknown</td>
<td></td>
<td>Unknown</td>
</tr>
<tr>
<td>libc.so.6.1</td>
<td>0x200000000001fa890</td>
<td>Unknown</td>
<td></td>
<td>Unknown</td>
</tr>
</tbody>
</table>
Hidden Floating Point Exceptions

• Check with `dmesg` for messages within the system log:
  
  a.out(28282): floating-point assist fault at ip 40000000000001d11, isr 0000020000000008
  mat_dist(28703): floating-point assist fault at ip 40000000000003861, isr 0000020000000004

• Look into “Intel Itanium Architecture Software Developer's Manual” for a description of the ISR.

• Assist faults are managed by the kernel.
  – Can be a killer of performance and scalability if assist faults occur at high rates.

• Reason for assist faults:
  – Wrong precision chosen for the floating point operations
    • Code was run in single precision, should be double precision.
  – Programming errors and/or bad algorithmic design
  – Speculative floating point operations due to high opt. levels
    • Try to compile with `-IPF_fp_speculation [save|off]`

Single Step Trap

Divide by zero
Data Display Debugger -- ddd

• Front end GUI to gdb and other debuggers, written by Dorothea Lütkehaus and Andreas Zeller
• Home page at http://www.gnu.org/software/ddd/
• Features an interactive graphical data display, where data structures are displayed as graphs
• Works best with gdb, but can work with idb in dbx mode
  – ddd -debugger idb -dbx ./a.out
Main Window

• By default displays the Menu Bar, Tool Bar, Source Window, Debugger Console and Status Line.

• The Data Window, when invoked, appears above the Source Window, and an optional Machine Code Window appears below the Source Window.
Debugging MPI with gdb

• In the first window set MPI_ATTACH_DEBUG equal to the rank to be debugged.
• Open a second window.
• Start your MPI code in the first window.

```
reiner@dcm24 75> setenv MPI_SLAVE_DEBUG_ATTACH 0
reiner@dcm24 75> mpirun -np 4 mxm4.mpi.x
```

MPI rank 0 sleeping for 20 seconds while you attach the debugger.
You can use this debugger command
```
gdb /proc/30541/exe 30541
```
or
```
idb -pid 30541 /proc/30541/exe
```
• Mouse the the gdb or idb line into your second window.
Single Node Tuning
## Some General Switches

<table>
<thead>
<tr>
<th>Feature Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable optimization</td>
<td>-O0</td>
</tr>
<tr>
<td>Optimize for speed (no code size increase), no SWP</td>
<td>-O1</td>
</tr>
<tr>
<td>Optimize for speed (default), includes SWP</td>
<td>-O2</td>
</tr>
<tr>
<td><strong>High-level optimizer</strong>, incl. prefetch, unroll, -FTZ</td>
<td>-O3</td>
</tr>
<tr>
<td>Aggressive optimizations ( == -O3 -ipo -static)</td>
<td>-fast</td>
</tr>
<tr>
<td>Create symbols for debugging</td>
<td>-g</td>
</tr>
<tr>
<td>Generate assembly files</td>
<td>-S</td>
</tr>
<tr>
<td>Assume no aliasing</td>
<td>-fno-fnalias</td>
</tr>
<tr>
<td>OpenMP 2.0 support</td>
<td>-openmp</td>
</tr>
<tr>
<td>Automatic parallelization for OpenMP threading</td>
<td>-parallel</td>
</tr>
</tbody>
</table>
New command line switches

Run-time checking (-check)
Floating point exception handling (-fpe)
Non-native I/O conversion (-convert)
Detect FP stack corruption (-fpstkchk)
Display traceback on errors (-traceback)
Link in threaded libraries (-threads)
Compiler Directives

**Fortran:**
- `cdec$ ivdep`  
  - no aliasing
- `cdec$ swp`  
  - try to software-pipeline
- `cdec$ noswp`  
  - disable software-pipelining
- `cdec$ loop count (NN)`  
  - hint for SWP
- `cdec$ distribute point`  
  - split this large loop
- `cdec$ unroll (n)`  
  - unroll $n$ times
- `cdec$ nounroll`  
  - do not unroll
- `cdec$ prefetch a`  
  - prefetch array “a”
- `cdec$ noprefetch c`  
  - do not prefetch array “c”

**C/C++:**
Use `#pragma` instead of `CDEC$`
Profile Guided Optimization: Three Steps

Step 1
Instrumented Compilation
```
ifort -prof_gen prog.c
```
Instrumented executable: prog.exe

Step 2
Instrumented Execution
prog.exe (on a typical dataset)
DYN file containing dynamic info: .dyn

Step 3
Feedback Compilation
```
ifort -prof_use prog.c
```
Merged DYN summary file: .dpi
Delete old dyn files unless you want their info included
Report Generation Options

- **-opt_report**
  - generate an optimization report to stderr (NB: or *file*)
- **-opt_report_file file**
  - specify the filename for the generated report
- **-opt_report_level level**
  - specify the level of report verbosity (min|med|max)
- **-opt_report_phase phase_name**
  - specify the phase that reports are generated against
- **-opt_report_routine name**
  - reports on routines containing the given name
- **-opt_report_help**
  - display the optimization phases available for reporting
Phase_names (from --opt_report_help)

- ipo
- ipo_inl
- ipo_cp
- ipo_modref
- ipo_lpt
- ipo_subst
- ipo_ratt
- ipo_vaddress
- ipo_pdce
- ipo_dp
- ipo_gprel
- ipo_pmerge
- ipo_dstat
- ipo_fps
- ipo_ppi
- ipo_unref
- ipo_wp
- ipo_dl
- ilo
- ilo_lowering
- ilo_strength_reduction
- ilo_reassociation
- ilo_copy_propagation
- ilo_convert_insertion
- ilo_convert_removal
- ecg
- ecg_gra
- **ecg_swap**
- ecg_predication
- ecg_speculation
- ecg_code
- ecg_code_cycles
- ecg_code_size
- ecg_code_size_fsp
- pgo
- hlo
- hlo_fusion
- hlo_distribution
- hlo_scalar Replacement
- hlo_unroll
- hlo_prefetch
- hlo_loadpair
- hlo_linear_trans
- hlo_opt_pred
- hlo_data_trans
- hlo_reroll
- hlo_array_contraction
- hlo_scalar_expansion
- all
Software Pipelining

**Sequential Loop**

**Software-Pipelined Loop**

- Traditional architectures use loop unrolling
  - Results in code expansion and increased cache misses
- Itanium™ Software Pipelining uses rotating registers

*Itanium™ provides direct support for Software Pipelining*
Sample SWP Report
--opt_report --opt_report ecg_swp

Swp report for loop at line 12 in multiply_d in file multiply_d.c

Resource II = 1
Recurrence II = 4 >0 means loop carried dep. Rewrite loop if possible
Minimum II = 4
Scheduled II = 4 Min. II = Sched. II \(\Rightarrow\) loop optimally scheduled
Percent of Resource II needed by arithmetic ops = 100%
Percent of Resource II needed by memory ops = 100%
Percent of Resource II needed by floating point ops = 100%

Number of stages in the software pipeline = 3

Following are the loop-carried memory dependency edges:
Store at line 12 \(\Rightarrow\) Load at line 12
Store at line 12 \(\Rightarrow\) Load at line 12

• What to look for:
  – Was loop software pipelined or “loop not pipelined”?
  – Scheduled II – most important info
Software Pipelining Terms

- **Initiation Interval (II)**
  the number of cycles between the start of successive iterations in the loop; If the II is n cycles, a new loop iteration will be completed every n cycles at steady state

- **Resource II**
  the smallest II that is feasible for pipelined loop according to the compiler’s knowledge of the available resources and architecture limitations (execution units, cache latencies etc)

- **Recurrence II**
  the smallest II fullfilling the (loop-carried) dependencies

- **Minimum II**
  Max (Resource II, Recurrence II)

- **Scheduled II**
  the II (cycles per iteration) of the pipelined loop as finally being created by compiler

- **GCS (Global Code Scheduler) II**
  number of cycles needed without Software Pipelining
  Now (since 8.1) printed too in SWP reports
Tips on how to read SWP report

- If Recurrence II > 0, means compiler detected loop carried dependencies
- If Minimum II = Scheduled II, means loop is optimally scheduled according to the compiler
- Percent of Resource II used by memory ops, floating point ops and integer ops shows the utilization of the corresponding execution units throughout the loop kernel
- If your floating point Resource II utilization is less than memory – not optimal situation for number crunching algorithm. Consider loop balancing.
lipfpm: Linux IPF Performance Monitor

- Reports counts of desired events for entire run of a program
- lipfpm [ options] command [ arguments]

- Options
  - `-c` request named collection of events
  - `-e cnt0 [-e cnt1 ...]` specific counts (up to four)
  - `-i` interactive selection of events
  - `-f` follow forks
  - `-h` display a help message
  - `-k` include counts at kernel level
  - `-l` lists available performance counter
  - `-o path` send output to path.command.PID

Example: `lipfpm -f -o mflops -e FP_OPS RETIRED \ dplace -x2 -c 4-7 ./magic3_16.exe`

`mpirun -np 64 lipfpm -f -o mflops -e \ FP_OPS RETIRED ./cpmd.x`
histx: Histogram Execution

- Profiling tool, it can sample either the IP (instruction pointer, aka program counter) or the call stack
- histx [ options] -o path [-s type] command

Options:

- -b specify bin bits when using ip sampling: 16, 32 or 64 (default: 16)
- -e specify event source (default: timer@1 )
- -f follow fork
- -h this message (command not run)
- -k also count kernel events for PM source
- -l include line level counts in IP sampling report
- -d sampling is off on launch, -t <signal> to toggle, -q <signal> to quit
- -o send output to file path. command. PID
- -s type of sampling
histx: Histogram Execution

- Event sources:
  - timer@n profiling timer events. A sample is recorded every n ticks. (One tick is about 0.977 ms.)
  - pm:<event>@n performance monitor events. A sample is recorded when the counter associated with <event> increases by n or more.
  - dlatM@N A sample is recorded whenever the number of loads whose latency is greater than or equal to M cycles is N larger than the number at the time of the previous sample. M must be a power of 2 between 4 and 4096.
  - numaM@N A sample (consisting of the cpu number and node number of the memory being accessed) is recorded whenever the number of loads whose latency is greater than or equal to M is N larger than the number at the time of the previous sample. M must be a power of 2 between 4 and 4096.
  - unaligned A sample is recorded for every unaligned load or store executed by the application.
histx: Histogramm Execution

- Types of sampling:
  - ip: Sample instruction pointer
  - callstack[N]: Sample the call stack. $N$, if given, specifies the maximum number of frames.
iprep: IP Sampling Report

• Generates a report from one or more raw IP sampling reports produced by histx

% iprep ip.prog.* > report.all
% more report.all

<table>
<thead>
<tr>
<th>Count</th>
<th>Excl. %</th>
<th>Incl. %</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12362</td>
<td>29.730</td>
<td>29.730</td>
<td>libm.so.6.1:cos</td>
</tr>
<tr>
<td>7716</td>
<td>18.557</td>
<td>48.286</td>
<td>libm.so.6.1:acos</td>
</tr>
<tr>
<td>6533</td>
<td>15.712</td>
<td>63.998</td>
<td>libm.so.6.1:asin</td>
</tr>
<tr>
<td>6338</td>
<td>15.243</td>
<td>79.241</td>
<td>a.out:f1 [prog.c:19]</td>
</tr>
<tr>
<td>5655</td>
<td>13.600</td>
<td>92.840</td>
<td>a.out:f2 [prog.c:29]</td>
</tr>
<tr>
<td>1401</td>
<td>3.369</td>
<td>96.210</td>
<td>a.out:_init</td>
</tr>
<tr>
<td>625</td>
<td>1.503</td>
<td>97.713</td>
<td>libm.so.6.1:sin</td>
</tr>
</tbody>
</table>

• iprep -c <n> cuts off the listing when the exclusive percentage goes below n; the default cutoff is 0.5%. 
pfmon / profile.pl

• Profile.pl

• Profile.pl is a Perl script that provides a simple way to do procedure-level profiling of an unmodified binary.

• The simplest way to use these scripts is as follows:

  profile.pl -c0-3 –x2 test_program.

• The 4 processes will be bound to processors 0-3 (via dplace) and the program will profiled under control of pfmon. The profile event will be CPU_CYCLES.

• The profile.pl script will create a map file (using makemap.pl) for test_program and put it into test_program.map.

• The profile samples themselves will go into sample.out. The analyzed profile will go into profile.out.
pfmon / profile.pl: Example with OpenMP

[root@palais fine_grain.omp]# profile.pl -c0-3 -x ./blast_waves < input
profile.pl: Parsing arguments and setting defaults.
profile.pl: Samples/tick defaults to: 9958070 for event CPU_CYCLES.
profile.pl: Program to profile is: ./blast_waves.
profile.pl: Running the program under pfmon control:
profile.pl: pfmon --system-wide --smpl-outfile=sample.out --smpl-entries=100000
-u -k --short-smpl-periods=9958070 --smpl-output-format=compact --events=CPU_CYCLES --cpu-mask=F ./blast_waves

BI-CGSTAB & symmetric difference scheme
3D Laminar shock wave propagation
Re, Pr
Re: 100000.0 Pr: 0.720000
(nx, ny, nz)?
grid size is: 33 33 32
(CFL, nuim, nuex2, nuex4)?
CFL: 2.000000 nuim: 0.1000000 nuex2: 0.1000000 nuex4: 5.000000E-02
What scheme you will use - explicit(0) or implicit(1)?
Implicit scheme is working
What initial configuration do you want- cubic(0) or spheric(1)?
Cubic initial configuration
Number of Time Steps?
Number of Time Steps: 50
Time step: 1 dt: 1.530094E-03
9.407178E-03 1.000000E-03 convergence after 1 iterations.
Timing for time step: 0.1835938
pfmon / profile.pl: Example

profile.pl: Program has completed.
profile.pl: Checking the profile results.
profile.pl: cpu 0: 994 samples.
profile.pl: cpu 1: 993 samples.
profile.pl: cpu 2: 993 samples.
profile.pl: cpu 3: 993 samples.
profile.pl: Merging sample files into a single file.
profile.pl: cat sample.out.cpu0 sample.out.cpu1 sample.out.cpu2 sample.out.cpu3 > sample.out
profile.pl: Removing the per processor sample files.
profile.pl: rm -f sample.out.cpu0 sample.out.cpu1 sample.out.cpu2 sample.out.cpu3
profile.pl: Creating a program map file.
makemap.pl: Read 4529 symbols from ./blast_waves.
makemap.pl: Read 590 symbols from /lib/libm.so.6.1.
makemap.pl: Read 1374 symbols from /opt/intel/compiler70/ia64/lib/libcxa.so.2.
makemap.pl: Read 251 symbols from /lib/libpthread.so.0.
makemap.pl: Read 2577 symbols from /lib/libc.so.6.1.
makemap.pl: Read 218 symbols from /lib/ld-linux-ia64.so.2.
makemap.pl: Read 884 symbols from /opt/intel/compiler70/ia64/lib/libunwind.so.2.
makemap.pl: Sorting symbols.
makemap.pl: Wrote 10428 symbols to blast_waves.map
profile.pl: Running the profile analyzer.
profile.pl: analyze.pl blast_waves.map sample.out > profile.out
analyze.pl: Read 10429 symbols from blast_waves.map.
analyze.pl: No System.map file found: kernel analysis will be skipped.
analyze.pl: total observations: 3828
analyze.pl: Sorting the user observations
profile.pl: Profile results are in file: profile.out.
[root@palais fine_grain.omp]#
Total observations: 3843
user ticks: 3843 100%

<table>
<thead>
<tr>
<th>Ticks</th>
<th>Percent</th>
<th>Cumulative Percent</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1922</td>
<td>50.01</td>
<td>50.01</td>
<td>_mat_times_vec__174__par_loop20</td>
</tr>
<tr>
<td>773</td>
<td>20.11</td>
<td>70.13</td>
<td>_shell__207__par_loop5</td>
</tr>
<tr>
<td>215</td>
<td>5.59</td>
<td>75.72</td>
<td>_jacobian__30__par_loop8</td>
</tr>
<tr>
<td>97</td>
<td>2.52</td>
<td>78.25</td>
<td>POW_COMMON</td>
</tr>
<tr>
<td>74</td>
<td>1.93</td>
<td>80.17</td>
<td>_shell__258__par_loop6</td>
</tr>
<tr>
<td>72</td>
<td>1.87</td>
<td>82.05</td>
<td>_bi_cgstab_block__87__par_loop15</td>
</tr>
<tr>
<td>68</td>
<td>1.77</td>
<td>83.81</td>
<td>_bi_cgstab_block__60__par_loop13</td>
</tr>
<tr>
<td>67</td>
<td>1.74</td>
<td>85.56</td>
<td>_bi_cgstab_block__99__par_loop16</td>
</tr>
<tr>
<td>62</td>
<td>1.61</td>
<td>87.17</td>
<td>_bi_cgstab_block__114__par_loop17</td>
</tr>
<tr>
<td>59</td>
<td>1.54</td>
<td>88.71</td>
<td>__kmp_wait</td>
</tr>
<tr>
<td>59</td>
<td>1.54</td>
<td>90.24</td>
<td>_bi_cgstab_block__72__par_loop14</td>
</tr>
<tr>
<td>49</td>
<td>1.28</td>
<td>91.52</td>
<td>_shell__175__par_loop4</td>
</tr>
<tr>
<td>49</td>
<td>1.28</td>
<td>92.79</td>
<td>pow</td>
</tr>
<tr>
<td>45</td>
<td>1.17</td>
<td>93.96</td>
<td>_bi_cgstab_block__127__par_loop18</td>
</tr>
<tr>
<td>44</td>
<td>1.14</td>
<td>95.11</td>
<td>_flux__58__par_loop10</td>
</tr>
<tr>
<td>40</td>
<td>1.04</td>
<td>96.15</td>
<td>_bi_cgstab_block__143__par_loop19</td>
</tr>
<tr>
<td>33</td>
<td>0.86</td>
<td>97.01</td>
<td>_flux__21__par_loop9</td>
</tr>
<tr>
<td>30</td>
<td>0.78</td>
<td>97.79</td>
<td>_bi_cgstab_block__35__par_loop12</td>
</tr>
<tr>
<td>16</td>
<td>0.42</td>
<td>98.20</td>
<td>_shell__312__par_loop7</td>
</tr>
<tr>
<td>16</td>
<td>0.42</td>
<td>98.62</td>
<td>__kmp_yield</td>
</tr>
</tbody>
</table>

"profile.out" 39L, 2156C
pfmon / profile.pl (MPI)

To use MPI with profile.pl

```
mpirun -np 4 /usr/bin/profile.pl -c0-3 -s1 ./blast_waves < input
```

![Image of terminal output]

```
[root@palais blastwave-mpi]# mpirun -np 4 /usr/bin/profile.pl -c0-3 -s1 ./blast_waves < input
profile.pl: Parsing arguments and setting defaults.
profile.pl: Samples/tick defaults to: 9958070 for event CPU_CYCLES.
profile.pl: Program to profile is: ./blast_waves.
profile.pl: Running the program under pfmon control:
profile.pl: pfmon --system-wide --smpl-outfile=sample.out --smpl-entries=100000 -u -k --short-sm
profile.pl: periods=9958070 --smpl

BI-CGSTAB & symmetric differencing
3D Laminar shock wave
Re, Pr
Re: 100000.0
(nx,ny,nz)?
grid size is: 100x100x100
(CFL, nuim, nuex, nuex2, nuex3)
CFL: 2.000000

profile.pl: Program has completed.
profile.pl: Checking the profile results.
profile.pl: cpu 0: 324 samples.
profile.pl: cpu 1: 324 samples.
profile.pl: cpu 2: 324 samples.
profile.pl: cpu 3: 324 samples.
profile.pl: Merging sample files into a single file.
profile.pl: cat sample.out.cpu0 sample.out.cpu1 sample.out.cpu2 sample.out.cpu3 > sample.out
profile.pl: Removing the per processor sample files.
profile.pl: Existing (and current) map file found: blast_waves.map.
profile.pl: Running the profile analyzer.
profile.pl: analyze.pl blast_waves.map sample.out > profile.out
analyze.pl: Read 9473 symbols from blast_waves.map.
analyze.pl: Read 9374 symbols from System.map.
analyze.pl: total observations: 1296
analyze.pl: Sorting the kernel observations
analyze.pl: Sorting the user observations
profile.pl: Profile results are in file: profile.out.
```
Documentation on Performance Counter

• Intel Itanium 2 Processor Reference Manual for Software Development and Optimization
  – http://www.intel.com/design/itanium2/documentation.htm

• Excellent report on performance analysis via Hardware performance counters by David Levinthal
Numatools
Coding to Get Good Data Placement

• Initialization with the “first-touch” policy on a single processor
  – All data on a single node
  – Bottleneck in access to that node

• Initialization with the “first-touch” policy with multiple processors in a parallel loop
  – Data is distributed naturally
  – Each processor has local data
  – Minimal data exchange between nodes
  – Page edge effects
Running on Specific CPUs -- taskset

• Known as 'runon' under Irix and Redhat ProPack 3.
  – Syntax slightly different. Consult the man page.
• The `taskset` command executes a command on a CPU or list of CPUs:
  ```
  taskset -c 1,3-5 ./a.out
  ```
• Find idle CPUs using `top`
• `taskset` restricts the process and its children to run on the set of listed CPUs, but does not prevent them from moving around within it.
• Numbering scheme of logical CPUs is relative to the corresponding `cpuset`. 
Influencing NUMA Behaviour -- *numactl*

```
numactl -membind=<nodes> --cpubind=<nodes> \ 
<command>
```

- **membind**: Only allocate memory from nodes. Allocation will fail when there is not enough memory available on these nodes.
- **cpubind**: Only execute process on the CPUs of nodes.
- **interleave**: Memory allocation round-robin across nodes.
Running on Specific CPUs -- cpuset

• `cpuset(1)` allows you to run your programs on a restricted subset of processors and associated memory, called a cpumemset.
• It requires the prior creation of a cpumemset to run the program in.
• Cpusets may be created:
  – Manually by root: `cpuset -c <name> -f <file name>`
  – Automatically by batch schedulers such as Platform's LSF or Altair Engineering's PBSPro.
• Run something in a cpuset: `cpuset -i my_cpuset -I a.out`.
• Determine existing cpumemsets with `ls -d /dev/cpuset`
  – You get a list of created cpusets.
  – `cat /dev/cpuset/<name>/cpus` lists CPUs belonging to the set.
• As usual, read the `man` pages!
Binding Threads on CPUs -- dplace

• Use of profiling tools may require modification of placement flags. E.g., OpenMP program on ProPack 4.0:

```bash
dplace -x5 -c0-15 histx -o prof a.out
```

```
<table>
<thead>
<tr>
<th>histx</th>
<th>skip</th>
<th>(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.out master</td>
<td>place</td>
<td>(0)</td>
</tr>
<tr>
<td>OpenMP monitor</td>
<td>skip</td>
<td>(1)</td>
</tr>
<tr>
<td>a.out slave1</td>
<td>place</td>
<td>(0)</td>
</tr>
<tr>
<td>a.out slave2</td>
<td>place</td>
<td>(0)</td>
</tr>
<tr>
<td>...</td>
<td>place</td>
<td></td>
</tr>
</tbody>
</table>
```

101₂ = 5₁₀

Or generally `dplace -x2 -c0-15 a.out`

Or use explicit placement:
```bash
dplace -e -c x,0,x,1-15 histx ...
```

• Always use dplace in conjunction with OpenMP programs!

• For MPI jobs use
  ```bash
  -mpirun -np 8 dplace -s1 -c 0-32 hpcc.exe
  ```
Determining Data Access Patterns -- dlook (1)

dlook ls
anaconda-ks.cfg install.log install.log.syslog

Exit:  ls
Pid: 12905       Thu Aug 22 10:45:34 2002

Process memory map:

[...] 4000000000000000-40000000000024000 r-xp 0000000000000000 04:13 58723137   /bin/ls  
[4000000000000000-4000000000004000] 1 page  on node  2  MEMORY|SHARED
[4000000000000000-4000000000008000] 1 page  on node  0  MEMORY|SHARED
[4000000000000000-4000000000008000] 1 page  on node  1  MEMORY|SHARED
[4000000000000000-400000000000c000] 1 page  on node  2  MEMORY|SHARED
[4000000000000000-4000000000010000] 1 page  on node  1  MEMORY|SHARED
[4000000000000000-4000000000018000] 1 page  on node  2  MEMORY|SHARED
[4000000000000000-400000000001c000] 1 page  on node  3  MEMORY|SHARED
[4000000000000000-4000000000020000] 1 page  on node  3  MEMORY|SHARED
[...]
OpenMP
Programming for the Default First-Touch Policy (continued)

integer i, j, n, niters
parameter (n = 8*1024*1024, niters = 1000)
real a(n), b(n), q

c initialization
`omp parallel do private(i) shared(a,b)
do i = 1, n
  a(i) = 1.0 - 0.5*i
  b(i) = -10.0 + 0.01*i*i
enddo

c real work

do it = 1, niters
  q = 0.01*it
`omp parallel do private(i) shared(a,b,q)
do i = 1, n
  a(i) = a(i) + q*b(i)
enddo
enddo
### Programming for the Default First-Touch Policy (continued)

- **Check placement with dlook:**

  ```
  reiner@dcm33 98>dlook -o out.dlook dplace -x 2 -c8-15 ./matd
  reiner@dcm33 99>cat out.dlook | awk -f r.awk
  
  # # # # # # Summary # # # # # #
  
<table>
<thead>
<tr>
<th>Node</th>
<th>pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3347</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
  
  Bad placement! Everything is located in one node! Parallelize data initialization!
Programming for the Default First-Touch Policy (continued)

After parallelization of the data initialization:

#### Summary

<table>
<thead>
<tr>
<th>Node</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>878</td>
</tr>
<tr>
<td>5</td>
<td>828</td>
</tr>
<tr>
<td>6</td>
<td>824</td>
</tr>
<tr>
<td>7</td>
<td>823</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

BEGIN {
    n=16;
    for (i=0;i < n;i++) a[i]=0;
}

# $5 == "node" { print "$6" "$2"; a[6]=a[6]+$2; }
# $1 !~ "-400" && $5 == "node" {print "$1" "$6" "$2";
a[6]=a[6]+$2; }
$1 !~ "-400" && $5 == "node" { a[6]=a[6]+$2; }

END {
    print "##### Summary ######"
    print "Node pages"
    for (i=0;i < n;i++) print i" "a[i];
}

Perfect
False Sharing Fixed

integer m, n, i, j
real a(m,n), s(32,m)
c$omp parallel do private(i,j) shared(s,a)
do i = 1, m
  s(1,i) = 0.0
  do j = 1, n
    s(1,i) = s(1,i) + a(i,j)
  enddo
enddo
Environment Variables For Tuning

- **OMP_SCHEDULE** – static, dynamic, guided
- **OMP_CHUNK_SIZE**
- **KMP_LIBRARY** - Sets runtime execution mode
  - serial – single-processor execution
  - throughput – for multiuser environment, yields cpus to other processes when waiting for work (default). Analogous to IRIX _DSM_WAIT=YIELD
  - turnaround – worker threads do not yield while waiting for work. Analogous to IRIX _DSM_WAIT=SPIN
- **KMP_BLOCKTIME** – Sets spin time in ms before thread is released for re-scheduling
- **KMP_STACKSIZE** – Stacksize of the OpenMP slaves
- **KMP_MONITOR_STACKSIZE** – Stacksize of the monitor thread.
Compiling MPI Programs

```bash
icc prog.c -lm mpi

icc prog.C -lm mpi

ifort prog.f -lm mpi
```
Launching MPI Programs

• On most machines, the `mpirun` command launches MPI applications:
  
  \[ mpirun -np \text{ num\_Procs } \text{ user\_executable [ user\_args]} \]

• Launching a program to run with 5 processes on one computer
  
  \% mpirun -np 5 ./a.out

• Example: Launching a program to run with 64 processes on each of two systems
  
  \% mpirun host1,host2 64 ./a.out
MPI Optimization Hints

• Do not use wildcards, except when necessary
• Do not oversubscribe number of processors
• Collective operations are not all optimized
  – Use SHMEM to optimize bottlenecks
• Minimize use of MPI_barrier calls
• Optimized paths
  – MPI_Send() / MPI_Recv()
  – MPI_Isend() / MPI_Irecv()
• Less optimized:
  – ssend, rsend, bsend, send_init
• When using MPI_Isend()/MPI_Irecv(), be sure to free your request by either calling MPI_Wait() or MPI_Request_free()
Tunable Optimizations

• Eliminate Retries (Use MPI statistics) by increasing
  MPI_BUFS_PER_PROC

  setenv MPI_STATS
  or
  mpirun -stats -prefix "%g:" -np 8 a.out

  3: *** Dumping MPI internal resource statistics...
  3:
  3: 0 retries allocating mpi PER_PROC headers for collective calls
  3: 0 retries allocating mpi PER_HOST headers for collective calls
  3: 0 retries allocating mpi PER_PROC headers for point-to-point calls
  3: 0 retries allocating mpi PER_HOST headers for point-to-point calls
  3: 0 retries allocating mpi PER_PROC buffers for collective calls
  3: 0 retries allocating mpi PER_HOST buffers for collective calls
  3: 0 retries allocating mpi PER_PROC buffers for point-to-point calls
  3: 0 retries allocating mpi PER_HOST buffers for point-to-point calls
  3: 0 send requests using shared memory for collective calls
  3: 6357 send requests using shared memory for point-to-point calls
  3: 0 data buffers sent via shared memory for collective calls
  3: 2304 data buffers sent via shared memory for point-to-point calls
  3: 0 bytes sent using single copy for collective calls
  3: 0 bytes sent using single copy for point-to-point calls
  3: 0 message headers sent via shared memory for collective calls
  3: 6357 message headers sent via shared memory for point-to-point calls
  3: 0 bytes sent via shared memory for collective calls
  3: 15756000 bytes sent via shared memory for point-to-point calls
Using direct copy send/recv

• Set MPI_BUFFER_MAX to N
  – any message with size > N bytes will be transferred by direct copy if
    • MPI semantics allow it
    • the memory region it is allocated in is a globally accessible location
  – N=2000 seems to work well
    • shorter messages don’t benefit from direct copy transfer method
  – Look at stats to verify that direct copy was used.
Typical MPI-Env Variables Set

- **Always:**
  - `MPI_DSM_DISTRIBUTE=1`
  - `MPI_DSM_VERBOSE=1`

- **Try for performance enhancement**
  - `MPI_BUFFER_MAX = 2000`

- **Occasional:**
  - `MPI_BUFS_PER_PROC=32` or larger
  - `MPI_DSM_CPULIST=0-xx`
  - `MPI_STATS = 1`
  - `MPI_OPENMP_INTEROP=1`
Message Passing References

• Man pages
  – mpi
  – mpirun
  – shm

• Release notes
  – rpm -ql sgi-mpt | grep relnotes

  – http://techpubs.sgi.com
  – rpm -ql sgi-mpt | grep MPT_MPI_PM.pdf

• MPI Standard
  – http://www.mpi-forum.org/docs/docs.html
Organizer:

- Microarchitecture elements of Cray T3E
  - Enhanced hardware support synchronization primitives
- 8 bidirectional ports
- 3.2 GB/s per direction per port
- Low latency about 50 nsec per router
- Dual plane configuration:
  - 2 x 6.4GB/sec total bandwidth between C-bricks
256 Processor blade system.

Fat-Tree Topology for multiple racks

$64 \times 6.4 = 409.6 \text{ GB/s} = 1.28 \text{ GB/s/blade}$

2 Cables per line
Intel® Itanium® 2 - Why it is important?

High Bandwidth

Many functional units

Large on-chip caches

Large physical address space

System Bus
128 bits wide
200 MHz/400 MT/sec
6.4GB/sec

Width
2 bundles per clock
6 integer units
2 loads and 2 stores per clock
11 issue ports
4 FPMultiply Adds per Clock

Caches
L1: 2X16KB—1 clock latency
L2: 256K—5 clock latency
L3: 3-9MB—12 clk
32GB/sec bandwidth

Addressing
50-bit physical addressing
64-bit virtual addressing
Maximum page size of 4GB
pfmon: Getting Help on Events

To get a list of the supported events use the command

```
pfmon -l
```

`pfmon -l[regex]` to show events that match a regular expression

To find the meaning of a particular event, use the command

```
pfmon --event info=BACK_END_BUBBLE_ALL
```

Name: BACK_END_BUBBLE_ALL
Vcode: 0x0
Code: 0x0
PMD/PMC: [4, 5, 6, 7]
Umask: 0000
EAR: No (N/A)
BTB: No
MaxInc: 1 (Threshold 0)
Qual: None
Group: None
Set: None
Desc: Full Pipe Bubbles in Main Pipe -- Front-end, RSE, EXE, FPU/L1D so tall or a pipeline flush due to an exception/branch misprediction

```
[root@palais raefsky]# pfmon --event info=BACK_END_BUBBLE_ALL
```