

Origin / Onyx2 Quick Reference Single-Processor Tuning

References which are online available (html and postscript) from <http://techpubs.sgi.com>:

- Origin2000 and Onyx2 Performance Tuning and Optimization Guide
- Topics in IRIX Programming
- MIPS Compiling and Performance Tuning Guide

Note that all words set in `typewriter` font have an online man page.

Step One: Get the Right Answers

Select those options that emphasize tracing and porting over performance. Options are described in the man pages of the `f77`, `f90`, `cc`, `CC` compilers and the `DEBUG_group`.

Flag	Usage
-n32	best choice for most programs (cannot mix with -o32!).
-64	Needed instead of -n32 when memory usage exceeds 2 GB.
trap_uninitialized	-DEBUG group option: to trap uninitialized variables.
-fullwarn	Use during development: emits useful warnings and remarks on your code.
-g	Preserves symbols for debugging; disables all optimizations.
-static	Allocate data in heap initialized to zero when foreign code expects this.
-r8 -i8	Default real and integer to 64-bit when porting from Cray only. Caution: explicitly declare external library functions.
-lmalloc_ss	Tracing and error detection for calls to malloc/free. See <code>malloc_ss</code> .

Choose a debugger and learn its command set:

- `dbx`: Standard UNIX debugger, elaborate command-line interface, see file: `/usr/lib/dbx.help`
- `cvd`: Graphical debug/test environment in Workshop package (license required).

Do not proceed until you have:

- A working makefile (see `make` and for parallel makefiles `pmake` and `smake`).
- Basic sanity-test cases that yield the expected answers.

Step Two: Use Existing Tuned Code

Hardware tuned mathematical functions are in:

Collection	man-page	Available functions
fastmath	<code>libfastm</code>	Fastest, but less accurate, versions of sin, cos, tan, exp, log, pow.
SCSL	<code>intro_libscsl</code>	FFT's, convolutions, BLAS, LAPACK, sparse solvers.
vector intrinsics	<code>math</code>	<code>vsin</code> , <code>vcos</code> , <code>vtan</code> , <code>vexp</code> , <code>vlog</code> , <code>vsqrt</code> ; enable automatic vectorization with <code>-O3</code> or <code>-LNO:vintr=on</code> .

For sequential version use `-lscs -lfastm` in the link line of the makefile. For parallel versions of the functions use `-lscs_mp -lfastm`. Note, `complib` is an older version of SCSL.

Step Three: Find Out Where to Tune

Use `timex` for overview, `perfex` to see code behavior and go down to source code level with `ssrun`.

Command	Purpose
<code>timex prog args</code>	Reports real, user, system time. High system time may be due to I/O, FPEs (<code>sigfpe</code>), or system calls (<code>par</code>).
<code>perfex -a [-y -x] prog args</code>	Get overview of program behavior; spot major issues.
<code>perfex -e nn prog args</code>	Reports R12K counter nn, e.g. <code>nn=26</code> for cache misses (list counters with <code>perfex -h</code>). Verify hypotheses, check for changes from <code>perfex -a</code> .
<code>ssrun -fpcsampx prog args</code>	Sample on realtime intervals; show functions using most real time.
<code>ssrun -fcy_hwc prog args</code>	Sample on cycles; show functions using most CPU cycles.
<code>ssrun -gfp_hwc prog args</code>	Sample on FLOPS; show functions doing the most arithmetic.
<code>ssrun -dsc_hwc prog args</code>	Sample on secondary data cache misses.
<code>ssrun -fpe prog args</code>	Trace floating point exceptions.
<code>ssrun -ideal prog args</code>	Profile true counts of basic block use; shows most-used functions. <code>prof -butterfly name.ideal.pid</code> shows the call tree of the program.

`perfex` output goes to `stderr`. `ssrun` output goes to file `prog.name.exptype.pid`. Display the measurements with the `prof -h prog.name.exptype.pid` command. The man page of `speedshop` gives more information of different type of experiments.

Step Four: Let the Compiler Do the Work

Don't rely on default options. General group options for the compilers are described in `OPT`, `IPA`, `LNO`, `ABI`. Turn off debugging flags and try these optimization flags:

Option	Effect and Purpose
<code>-O3</code>	Enable maximum optimizations (includes LNO).
<code>-mips4</code>	For R12K, R10K, R8K, R5K only; use <code>-mips3</code> for R4K (<code>mips_ext</code>).
<code>-OPT:IEEE_arithmetic=3</code> <code>-OPT:roundoff=3</code>	See <code>opt</code> for discussion. Check that answers are still correct after applying these options.
<code>-IPA=on</code>	Enable inter-procedural analysis. Introduces longer link times. <code>ipa</code> .
<code>-OPT:alias=name</code>	Disambiguate pointer references. Use <code>name=disjoint</code> , or <code>restrict</code> , whenever possible. See the <code>opt</code> man page.

Step Five: Tune Cache Performance

- Use Loop Nest Optimizations (via `-O3`) to enable loop fusion, interchange, cache blocking, array padding and prefetching
- Avoid large power-of-2 strides/dimensions that cause cache thrashing
- Use stride 1 accesses whenever possible
- Group together data used at the same time, de-vectorize your code
- Use LNO directives to fine-tune its actions (`lno`)
- Use larger page sizes to reduce TLB misses Command Purpose
- Try to avoid the use of temporary arrays, minimize data copies
- The compilers are good at instruction level optimization, but they often benefit from high level insight.
- Compile with the `-S` flag to create symbolic assembly language output files suffixed with `.s`. In this `.s` file look for `< swp >` steady state lines to see the software pipeline performance.
- Try to achieve overlap of memory and compute instructions (`prefetch`).

If waiting for IO is a problem try asynchronous IO (`aiio_sgi_init`) to let the IO overlap with compute tasks. Other IO improvements can be made with direct IO (`open O_DIRECT` option) or using striped disks (`xlv`) and RAID 3. IO transfer can be monitored by `sar -b` and the the use of the system buffer cache by `bufview`.

Origin/Onyx2 Quick Reference Multi-Processor Tuning

References which are online available (html and postscript) from <http://techpubs.sgi.com>:

- Origin2000 and Onyx2 Performance Tuning and Optimization Guide
- MIPSpro Auto-Parallelizing Option Programmer's Guide
- MIPSpro 7 Fortran 90 Commands and Directives Reference Manual
- C Language Reference Manual

For programming with OpenMP see also <http://www.openmp.org>

Step One: Tune Single Processor Performance

See other side.

Step Two: Parallelize Code

Choose parallelization methodology:

- Automatic parallelization option: -apo
- OpenMP library directives: -mp (mp, pe_environ, dsm, omp_threads)
- Other libraries: MPI, shmem, PVM, pthreads (mpi, shmem, pvm)

The OpenMP shared memory parallelization strategy (the first 2 items) are for most programs the best approach. If message passing is needed, shmem or MPI are preferred over PVM. Note that only OpenMP (shared memory) parallelization is discussed in this document.

First start to parallelize small parts of the program and test every part after completion. Try to combine different parts, to reduce the overhead of creation of parallel regions, and move the parallelization to a higher level.

After these first steps profile the code to monitor degree of parallelization:

- Vary number of threads (for OpenMP library, use `setenv OMP_NUM_THREADS N`)
- Measure wall clock time to determine speedup (e.g., use `time`).
- `perfex -a -mp` prints all counts for each thread
- SpeedShop generates an output file for each thread
- Calculate the fraction S of the code which is parallelized: $S = \frac{N(T_1 - T_N)}{T_1(N-1)}$, where T_1 is the time of the job running on a single CPU and T_N the time of the job running on N CPU's.
- Stop using more CPU's and try to get better scalability, if the speedup/ N of the program becomes less than 50% ($N < \frac{2-S}{1-S}$).

CPU activity may be displayed with `gr_osview` and `top`, memory usage with `gmemusage`, hardware configuration with `hinv` and `topology`. `osview` provides a dynamic display of Origin topology and performance (requires Performance Co- Pilot).

Step Three: Identify and Solve Bottlenecks

- Is load balance OK? To check balance of floating point operations `perfex -a -mp prog args —& grep "floating point"`
- Check parallel overhead by looking for the functions: `_mp_slave_wait_for_work`, `_mp_barrier_nthreads`, `_mp_barrier_master` in a `ssrun -fpcsamx` profile.
- Reduce overhead by parallelizing the outermost section(s) of the program.
- Use OpenMP's `nowait` options where possible and don't parallelize small loops.
- Try dynamic or guided loop scheduling (`schedule(dynamic/guided, chunk_size)`).
- Use `setenv OMP_DYNAMIC TRUE` to free system resources to other users.

- If the answers obtained with multiple CPU's differ from the single CPU answers check for rounding errors (usually not a problem) and race conditions (which is a problem).
- Avoid frequent updated of shared variables.
- Are there a lot of secondary cache misses? `perfex -a -y prog args` If so, false sharing or data placement may be a problem.

Step Four: Fix False Sharing

Is there false sharing? Check `perfex` output to determine if interventions and/ or invalidations are a large fraction of secondary cache misses.

If false sharing is a problem,

- use SpeedShop to monitor stores to shared cache lines:
`setenv _SPEEDSHOP_HWC_COUNTER_NUMBER 31, 28 or 30`
`ssrun -prof_hwc prog args`
- Revise data structures or algorithms to remedy the problem.
- Check shared data and static variables, common blocks, private and public variables in shared objects.
- Use critical regions to indentify which part of the code goes wrong.

Step Five: Tune For Data Placement

For well- parallelized OpenMP programs that generate a lot of secondary cache misses but do not scale well, determine sensitivity to data placement. Try as few steps as needed to achieve satisfactory performance:

- Try round- robin page allocation (`setenv _DSM_ROUND_ROBIN on`) If this don't solve scaling problems, the program needs to be modified to make sure the data are properly distributed.
- Make sure data initializations are parallelized. This relies on "first touch": data are stored in memory of processor which first accesses a page of data. Note that `malloc` does not touch the data allocated.
- Ensure proper data placement via directives and pragmas.

The MIPSpro Fortran 77 Programmer's Guide and C Language Reference Manual describe the data distribution directives and the following environment variables:

Variable	Use and Possible Values	Default
<code>_DSM_VERBOSE</code>	Set (any value) for DSM options in effect.	not set
<code>_DSM_ROUND_ROBIN</code>	Allocate pages cyclically from all nodes used by job.	not set
<code>_DSM_PPM</code>	Processes per memory, set to 1 to use only one CPU per memory.	2
<code>_DSM_BARRIER</code>	Set to FOP to enable inter-thread barrier synchronization using <code>fetchop</code> instructions.	SHM
<code>_DSM_MUSTRUN</code>	Set to lock threads to processors. Not recommended in time-sharing environments.	not set
<code>_DSM_WAIT</code>	Set to SPIN Specifies that a thread wait in a loop until the synchronization event succeeds.	YIELD
<code>_DSM_OFF</code>	When set to OFF disables non uniform memory access calls.	not set
<code>PAGESIZE_STACK</code> , <code>_DATA</code> , and <code>_TEXT</code>	Set virtual page sizes in KB. May reduce number of TLB faults.	16
<code>dplace</code>	NUMA memory placement tool	by IRIX

© SGI, November 11, 1999