Per-Core Performance Optimization

Steve Lantz
Senior Research Associate
Cornell CAC

Workshop: Data Analysis on Ranger, Oct. 13, 2009
Putting Performance into Design and Development

Designing for parallelism and scalability is a topic in itself...

...this talk is about principles and practices during the later stages of development that lead to better performance on a per-core basis
What Matters Most in Per-Core Performance

*Good memory locality!*

- Code accesses contiguous memory addresses
  - Reason: Data arrives in cache lines, including neighbors
  - Reason: Loops can be vectorized for SSE (explained in a moment)
- Multiple operations involving the same data item are collected together
  - Reason: Access to cache is much faster than to RAM
- Data are aligned on doubleword boundaries
  - Reason: More efficient to have data items not straddling cache lines
- Goal: to have the data stay longer in cache, so that deeper levels of the memory hierarchy are accessed as infrequently as possible
Understanding The Memory Hierarchy

Relative Memory Bandwidths

- Functional Units
  - Registers
  - L1 Cache: ~50 GB/s, ~5 CP
  - L2 Cache: ~25 GB/s, ~15 CP

- Latency
  - Processor: ~12 GB/s, ~300 CP
  - L3 Cache Off Die: ~8 GB/s
  - Local Memory

Relative Memory Sizes

- L1 Cache: 16/32 KB
- L2 Cache: 1 MB
- Memory: 1 GB
What’s the Target Architecture?

• AMD initiated the x86-64 or x64 instruction set
  – Extends Intel’s 32-bit x86 instruction set to handle 64-bit addressing
  – Encompasses both AMD64 and EM64T = “Intel 64”
  – Differs from IA-64 (now called “Intel Itanium Architecture”)

• Additional SSE instructions access special registers & operations
  – 128-bit registers can hold 4 floats/integers or 2 doubles simultaneously
  – Within an SSE register, “vector” operations can be applied
  – Operations are also pipelined (e.g., load > multiply > add > store)
  – Therefore, multiple results can be produced every clock cycle
Understanding SSE, SIMD, and Micro-Parallelism

• For “vectorizable” loops with independent iterations, SSE instructions can be employed…

SSE = *Streaming SIMD Extensions*

SIMD = *Single Instruction, Multiple Data*

Instructions operate on multiple arguments simultaneously, in parallel Execution Units
Putting Performance into Development: Libraries

Designing for parallelism and scalability is a topic in itself…

…this talk is about principles and practices during the later stages of development that lead to better performance on a per-core basis.
Performance Libraries

- Optimized for specific architectures (chip + platform + system)
- Offered by different vendors
  - Intel Math Kernel Library (MKL)
  - AMD Core Math Library (ACML)
  - ESSL/PESSSL on IBM systems
  - Cray libsci for Cray systems
  - SCSL for SGI systems
- Usually far superior to hand-coded routines for “hot spots”
  - Writing your own library routines by hand is not like re-inventing the wheel; it’s more like re-inventing the muscle car
  - *Numerical Recipes* books are NOT a source of optimized code: performance libraries can run 100x faster
## HPC Software on Ranger, from Apps to Libs

<table>
<thead>
<tr>
<th>Applications</th>
<th>Parallel Libs</th>
<th>Math Libs</th>
<th>Input/Output</th>
<th>Diagnostics</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBER</td>
<td>PETSc</td>
<td>MKL</td>
<td>NetCDF</td>
<td>TAU</td>
</tr>
<tr>
<td>NAMD</td>
<td>PETSc</td>
<td>MKL</td>
<td>HDF5</td>
<td>PAPI</td>
</tr>
<tr>
<td>GROMACS</td>
<td>PLAPACK</td>
<td>ACML</td>
<td>HDF5</td>
<td>…</td>
</tr>
<tr>
<td>GAMESS</td>
<td>PLAPACK</td>
<td>GSL</td>
<td>PHDF5</td>
<td>…</td>
</tr>
<tr>
<td>NWChem</td>
<td>ScaLAPACK</td>
<td>GotoBLAS</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>SLEPc</td>
<td>GotoBLAS2</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>METIS</td>
<td>FFTW(2/3)</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>ParMETIS</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>SPRNG</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

10/13/2009  www.cac.cornell.edu
Intel MKL 10.0 (Math Kernel Library)

• Is optimized for the IA-32, Intel 64, Intel Itanium architectures
• Supports Fortran and C interfaces
• Includes functions in the following areas:
  – Basic Linear Algebra Subroutines, for BLAS levels 1-3 (e.g., Ax+y)
  – LAPACK, for linear solvers and eigensystems analysis
  – FFT routines
  – Transcendental functions
  – Vector Math Library (VML), for vectorized transcendental
  – …others
Using Intel MKL on Ranger

• Enable MKL
  – module load mkl
  – module help mkl

• Compile and link for C/C++

  mpicc -I$TACC_MKL_INC mkl_test.c -L$TACC_MKL_LIB -lmkl_em64t

• Compile and link for Fortran

  mpif90 mkl_test.f90 -L$TACC_MKL_LIB -lmkl_em64t
GotoBLAS and FFTW

GotoBLAS
• Hand-optimized BLAS, minimizes TLB misses
• Only testing will tell what kind of advantage your code gets

FFTW, the Fastest Fourier Transform in the West
• Cooley-Tukey
• Prime Factor algorithm, most efficient with small prime factors like (2, 3, 5, and 7)
• Automatic performance adaptation
GSL, the Gnu Scientific Library

- Special Functions
- Vectors and Matrices
- Permutations
- Sorting
- Linear Algebra/BLAS Support
- Eigensystems
- Fast Fourier Transforms
- Quadrature
- Random Numbers
- Quasi-Random Sequences
- Random Distributions

- Statistics, Histograms
- N-Tuples
- Monte Carlo Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation
- Root-Finding
- Minimization
- Least-Squares Fitting
Putting Performance into Development: Compilers

Designing for parallelism and scalability is a topic in itself…

…this talk is about principles and practices during the later stages of development that lead to better performance on a per-core basis.
Compiler Options

• There are three important categories:
  – Optimization level
  – Architecture specification
  – Interprocedural optimization

• Generally you’ll want to supply one option from each category
Let the Compiler Do the Optimization

• Be aware that compilers can do sophisticated optimization
  – Realize that the compiler will follow your lead
  – Structure the code so it’s easy for the compiler to do the right thing (and for other humans to understand it)
  – Favor simpler language constructs (pointers and OO code won’t help)

• Use the latest compilers and optimization options
  – Check available compiler options
    `<compiler_command> --help` {lists/explains options}
  – Refer to the User Guides, they usually list “best practice” options
  – Experiment with combinations of options
Basic Optimization Level: -On

- -O0 = no optimization: disable all optimization for fast compilation
- -O1 = compact optimization: optimize for speed, but disable optimizations which increase code size
- -O2 = default optimization
- -O3 = aggressive optimization: rearrange code more freely, e.g., perform scalar replacements, loop transformations, etc.

- Note that specifying -O3 is not always worth it…
  - Can make compilation more time- and memory-intensive
  - Might be only marginally effective
  - Carries a risk of changing code semantics and results
  - Sometimes even breaks codes!
-O2 vs. -O3

• Operations performed at default optimization level, -O2
  – Instruction rescheduling
  – Copy propagation
  – Software pipelining
  – Common subexpression elimination
  – Prefetching
  – Some loop transformations

• Operations performed at higher optimization levels, e.g., -O3
  – Aggressive prefetching
  – More loop transformations
Know Your Chip

• SSE level and other capabilities depend on the exact chip

• Taking an AMD Opteron “Barcelona” from Ranger as an example…
  – Supports AMD64, SSE, SSE2, SSE3, and “SSE4a” (subset of SSE4)
  – Does not support AMD’s more recent SSE5
  – Does not support all of Intel’s SSE4, nor its SSSE = Supplemental SSE

• In Linux, a standard file shows features of your system’s architecture
  – `cat /proc/cpuinfo` {shows cpu information}
  – If you want to see even more, do a Web search on the model number

• This information can be used during compilation
Specifying Architecture in the Compiler Options

With -x<code> {code = W, P, T, O, S… } or a similar option, you tell the compiler to use the most advanced SSE instruction set for the target hardware. Here are a few examples of processor-specific options.

Intel 10.1 compilers:
- -xW = use SSE2 instructions (recommended for Ranger)
- -xO = include SSE3 instructions (also good for Ranger)
- -xT = SSE3 & SSSE3 (no good, SSSE is for Intel chips only)
- In Intel 11.0, these become -msse2, -msse3, and -xssse3

PGI compilers:
- -tp barcelona-64 = use instruction set for Barcelona chip
Interprocedural Optimization (IP)

- Most compilers will handle IP within a single file (option -ip)

- The Intel -ipo compiler option does more
  - It places additional information in each object file
  - During the load phase, IP among ALL objects is performed
  - This may take much more time, as code is recompiled during linking
  - It is **important** to include options in `link` command (-ipo -O3 -xW, etc.)
  - All this works because the special Intel xild loader replaces `ld`
  - When archiving in a library, you must use `xiar`, instead of `ar`
Interprocedural Optimization Options

Intel 10.1 compilers:
- `-ip` enable single-file interprocedural (IP) optimizations
  - Limits optimizations to within individual files
  - Produces line numbers for debugging
- `-ipo` enable multi-file IP optimizations (between files)

PGI compilers:
- `-Mipa=fast,inline` enable interprocedural optimization
  *There is a loader problem with this option*
Other Intel Compiler Options

- **-g**  
  generate debugging information, symbol table

- **-vec_report#**  
  {# = 0-5} turn on vector diagnostic reporting

- **-C (or -check)**  
  enable extensive runtime error checking

- **-CB -CU**  
  check bounds, check uninitialized variables

- **-convert kw**  
  specify format for binary I/O by keyword {kw = big_endian, cray, ibm, little_endian, native, vaxd}

- **-openmp**  
  multi-thread the executable based on OpenMP directives

- **-openmp_report#**  
  {# = 0-2} turn on OpenMP diagnostic reporting

- **-static**  
  load libs statically at runtime – *do not use*

- **-fast**  
  same as -O2 -ipo -static; *not allowed on Ranger*
Other PGI Compiler Options

- **-fast** use a suite of processor-specific optimizations: 
  -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline
  -Mvect=sse -Mscalarsse -Mcache_align -Mflushz

- **-mp** multithread the executable based on OpenMP directives

- **-Minfo=mp,ipa** turn on diagnostic reporting for OpenMP, IP
Best Practices for Compilers

• Normal compiling for Ranger
  – Intel:
    icc/ifort -O3 -ipo -xW prog.c/cc/f90
  – PGI:
    pgcc/pgcpp/pgf95 -fast -tp barcelona-64 -Mipa=fast,inline prog.c/cc/f90
  – GNU:
    gcc -O3 -fast -xipo -mtune=barcelona -march=barcelona prog.c

• -O2 is the default; compile with -O0 if this breaks (very rare)
• Effects of Intel’s -xW and -xO options may vary
• Debug options should not be used in a production compilation!
  – Compile like this only for debugging: ifort -O2 -g -CB test.c
Lab: Compiler-Optimized Naïve Code vs. GSL

• Code is from Numerical Recipes to do LU decomposition
• Compare timings with different optimizations
• Compare with implementation in GSL

• Compile with different flags, including “-g”, “-O2”, “-O3”
• Submit a job to see how fast it is
• Recompile with new flags and try again

• Sits in lude.tar.gz
Make/Run Instructions for the Lab

- Edit top of makefile to change compiler and flags
  - \texttt{COMPILER=pgcc}
  - \texttt{FFLAGS=-O2 -tp barcelona-64}
  - \texttt{VERSION=0}
- “\texttt{VERSION}” is tacked onto the end of the executable names
  - \texttt{nr0} and \texttt{gsl0} or \texttt{nr1} and \texttt{gsl1}
- “\texttt{make}” generates executables
- “\texttt{make list}” looks through your directory to find all executables
- \texttt{./nr0 \text{-f \text{-o output_file \text{-n 10000}}}}
  - \texttt{-f} tells it to tell you how you compiled the executable
  - \texttt{-o} is the name of an optional output file to verify results
  - \texttt{-n} is the size of the \textit{nxn} matrix
Obtaining Results for the Lab

- Edit makefile to use “FFLAGS=-g” and VERSION=0, then “make”
- Edit makefile to use “FFLAGS=-O2” and VERSION=1, then “make”
- Edit makefile to use “FFLAGS=-O3” and VERSION=2, then “make”
- “make list” to see that they are all there.
  - ./nr0 pgcc -O2 -tp barcelona-64
  - ./gsl0 pgcc -O2 -tp barcelona-64
  - ./nr1 pgcc -O3 -tp barcelona-64
  - ./gsl1 pgcc -O3 -tp barcelona-64
  - ./nr2 pgcc -g -tp barcelona-64
  - ./gsl2 pgcc -g -tp barcelona-64
- “qsub job.sge” or “make submit”
- Find the runtimes in the output to see the speeds
Putting Performance into Development: Tuning

Designing for parallelism and scalability is a topic in itself…

…this talk is about principles and practices during the later stages of development that lead to better performance on a per-core basis
In-Depth vs. Rough Tuning

In-depth tuning is a long, iterative process:
• Profile code
• Work on most time intensive blocks
• Repeat as long as you can tolerate...

For rough tuning during development:
• It helps to know about common microarchitectural features (like SSE)
• It helps to have a sense of how the compiler tries to optimize instructions, given certain features
Rules of Thumb for Writing Optimizable Code

• Minimize stride length
  – Stride length 1 is optimal for vectorizable code
  – It increases cache efficiency
  – It sets up hardware and software prefetching
  – Stride lengths of large powers of two are typically the worst case, leading to cache and TLB misses (due to limited cache associativity)

• Strive for stride-1 vectorizable loops
  – Can be sent to a SIMD unit
  – Can be unrolled and pipelined
  – Can be parallelized through OpenMP directives
  – Can be “automatically” parallelized (be careful…)

<table>
<thead>
<tr>
<th>G4/5 Velocity Engine (SIMD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel/AMD MMX, SSE, SSE2, SSE3 (SIMD)</td>
</tr>
<tr>
<td>Cray Vector Units</td>
</tr>
</tbody>
</table>
Best Practices from the Ranger User Guide

• Avoid excessive program modularization (i.e. too many functions/subroutines)
  – Write routines that can be inlined
  – Use macros and parameters whenever possible
• Minimize the use of pointers
• Avoid casts or type conversions, implicit or explicit
  – Conversions involve moving data between different execution units
• Avoid branches, function calls, and I/O inside loops
  – Why pay overhead over and over?
  – Structure loops to eliminate conditionals
  – Move loops into the subroutine, instead of looping around a subroutine call
More Best Practices from the Ranger User Guide

• Additional performance can be obtained with these techniques:
  – Memory Subsystem Tuning: Optimize access to the memory by minimizing the stride length and/or employing “cache blocking” techniques such as loop tiling
  – Floating-Point Tuning: Unroll inner loops to hide FP latencies, and avoid costly operations like division and exponentiation
  – I/O Tuning: Use direct-access binary files to improve the I/O performance

• These techniques are explained in further detail, with examples, in a Memory Subsystem Tuning document found online
Inlining

• What does inlining achieve?
  – It replaces a function call with a full copy of that function’s instructions
  – It avoids putting variables on the stack, jumping, etc.

• When is inlining important?
  – When the function is a hot spot
  – When function call overhead is comparable to time spent in the routine
  – When it can benefit from Inter-Procedural Optimization

• As you develop “think inlining”
  – The C “inline” keyword provides inlining within source
  – Use -ip or -ipo to allow the compiler to inline
Example: Procedure Inlining

```
integer :: ndim=2, niter=10000000
real*8  :: x(ndim), x0(ndim), r
integer :: i, j

...do i=1,niter
    ...r=dist(x,x0,ndim)
    ...
end do
end program

real*8 function dist(x,x0,n)
real*8  :: x0(n), x(n), r
integer :: j,n
r=0.0
do j=1,n
    r=r+(x(j)-x0(j))^2
end do
dist=r
end function
```

Trivial function dist is called niter times

```
integer:: ndim=2, niter=10000000
real*8  :: x(ndim), x0(ndim), r
integer :: i, j

...do i=1,niter
    ...r=0.0
    do j=1,ndim
        r=r+(x(j)-x0(j))^2
    end do
    ...
end do
end program

function dist has been inlined inside the i loop
```

Low-overhead loop j executes niter times
Stride 1 in Fortran and C

- The following snippets of code illustrate the correct way to access contiguous elements of a matrix, i.e., stride 1 in Fortran and C

Fortran Example:

```fortran
real*8 :: a(m,n), b(m,n), c(m,n)
...
do i=1,n
    do j=1,m
        a(j,i)=b(j,i)+c(j,i)
    end do
end do
```

C Example:

```c
double a[m][n], b[m][n], c[m][n];
...
for (i=0;i < m;i++){
    for (j=0;j < n;j++){
        a[i][j]=b[i][j]+c[i][j];
    }
}
```
The Penalty of Stride > 1

- For large and small arrays, always try to arrange data so that structures are arrays with a unit (1) stride.

Bandwidth Performance Code:

```fortran
do i = 1,10000000,istride
  sum = sum + data(i)
end do
```

Performance of Strided Access
Array Blocking, or Loop Tiling, to Fit Cache

Example: matrix-matrix multiplication

```fortran
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
    do jj=1,n,nb
        do kk=1,n,nb
            do i=ii,min(n,ii+nb-1)
                do j=jj,min(n,jj+nb-1)
                    do k=kk,min(n,kk+nb-1)
                        c(i,j)=c(i,j)+a(i,k)*b(k,j)
            end do
        end do
    end do
end do
```

Takeaway: all the performance libraries do this, so you don’t have to