Cornell University
Center for Advanced Computing

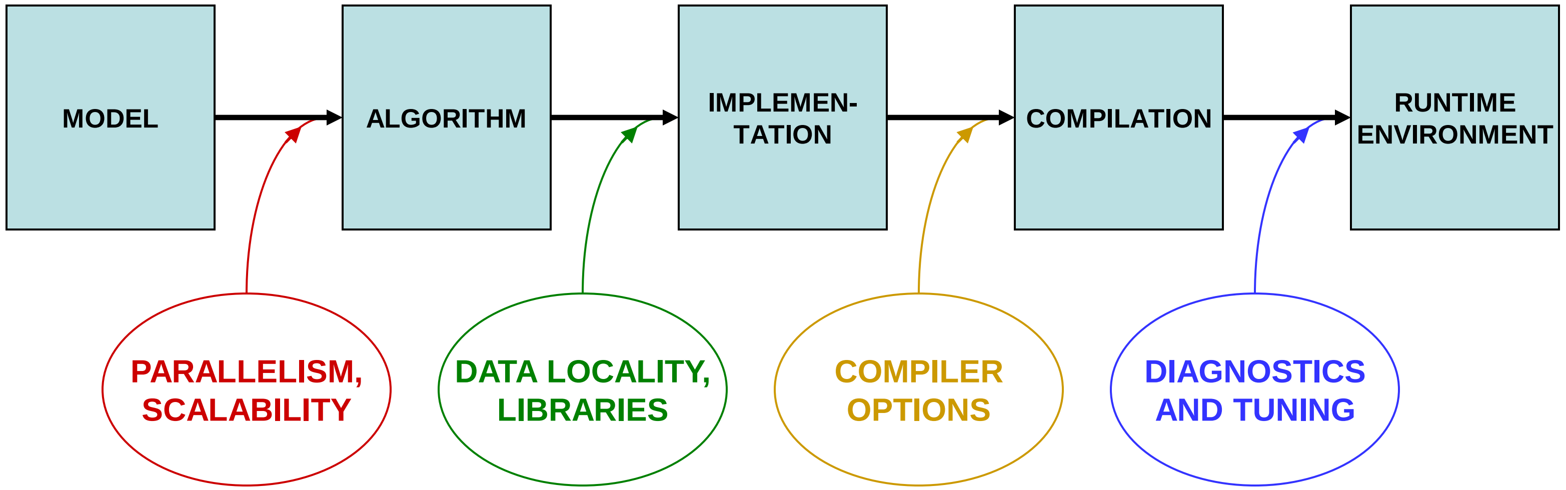
Optimization and Scalability

Steve Lantz
Senior Research Associate
Cornell CAC

Workshop: Parallel Computing on Ranger and Longhorn

May 17, 2012

Putting Performance into Design and Development



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *development* that lead to better performance on a per-core basis

Planning for Parallel

- Consider how your model might be expressed as an algorithm that naturally splits into many concurrent tasks
- Consider alternative algorithms that, even though less efficient for small numbers of processors, scale better so that they become more efficient for large numbers of processors
- Start asking these kinds of questions during the first stages of design, before the top level of the code is constructed
- Reserve matters of technique, such as whether to use OpenMP or MPI, for the implementation phase

Scalable Algorithms

- Generally the *choice of algorithm* is what has the biggest impact on parallel scalability.
- An efficient and scalable algorithm typically has the following characteristics:
  - The work can be separated into numerous tasks that proceed almost totally independently of one another
  - Communication between the tasks is infrequent or unnecessary
  - Lots of computation takes place before messaging or I/O occurs
  - There is little or no need for tasks to communicate globally
  - There are good reasons to initiate as many tasks as possible
  - Tasks retain all the above properties as their numbers grow

What *Is* Scalability?

- Ideal is to get *N* times more work done on *N* processors
- Strong scaling: compute a fixed-size problem *N* times faster
  - Usual metric is parallel speedup $S = T_1 / T_N$
  - Linear speedup occurs when $S = N$
  - Can't achieve it due to Amdahl's Law (no speedup for serial parts)
- Other definitions of scalability are equally valid, yet easier to do
  - Weak scaling: compute a problem that is *N* times bigger in the same amount of time
  - Special case of trivially or "embarrassingly" parallel: *N* independent cases run simultaneously, no need for communication

Capability vs. Capacity

- HPC jobs can be divided into two categories, capability runs and capacity runs
  - A capability run occupies nearly all the resources of the machine for a single job
  - Capacity runs occur when many smaller jobs are using the machine simultaneously
- Capability runs are typically done by codes with weak scaling
  - Strong scaling usually applies only over some finite range of *N* and breaks down when *N* becomes huge
  - Though a trivially parallelizable code is an extreme case of weak scaling, replicating such a code really just fills up the machine with a bunch of capacity runs instead of one big capability run

Predicting Scalability

- Consider the time to compute a fixed workload due to N workers:

```
total time = computation + message initiation + message bulk
computation = workload/N + serial time    (Amdahl's Law)
message initiation = [number of messages] * latency
message bulk = [size of all messages] / bandwidth
```

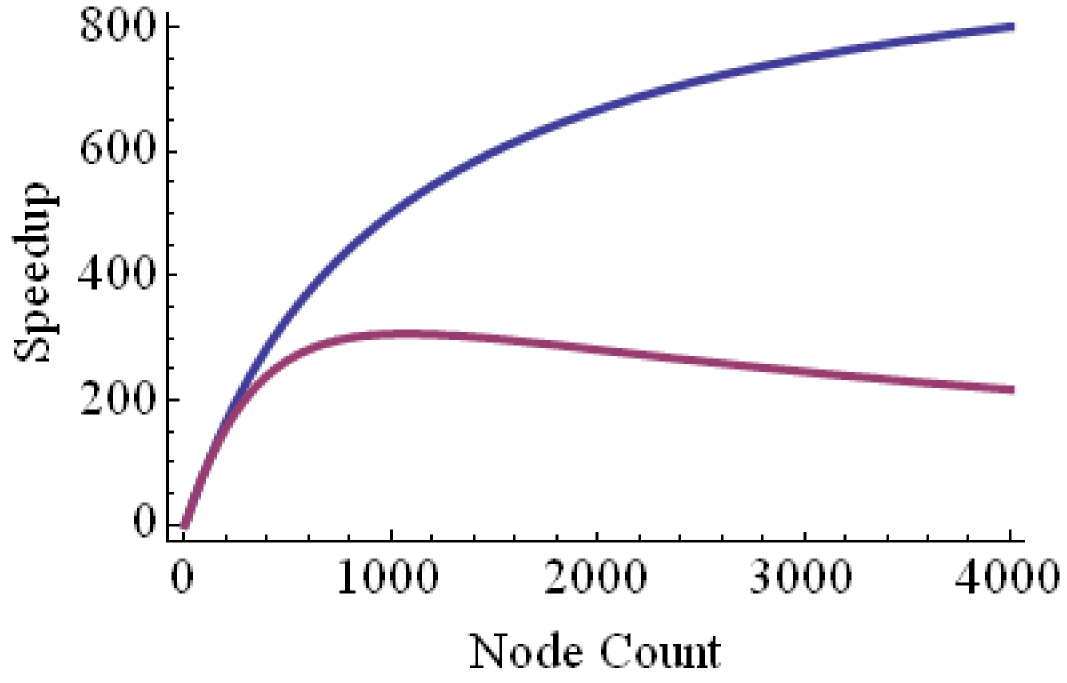
- The number and size of messages might themselves depend on N (unless all travel in parallel!), suggesting a model of the form:

```
total time = workload/N + serial time
                + k0 * N^a * latency + k1 * N^b / bandwidth
```

- Latency and bandwidth depend on hardware and are determined through benchmarks; other constants depend partly on application

The Shape of Speedup

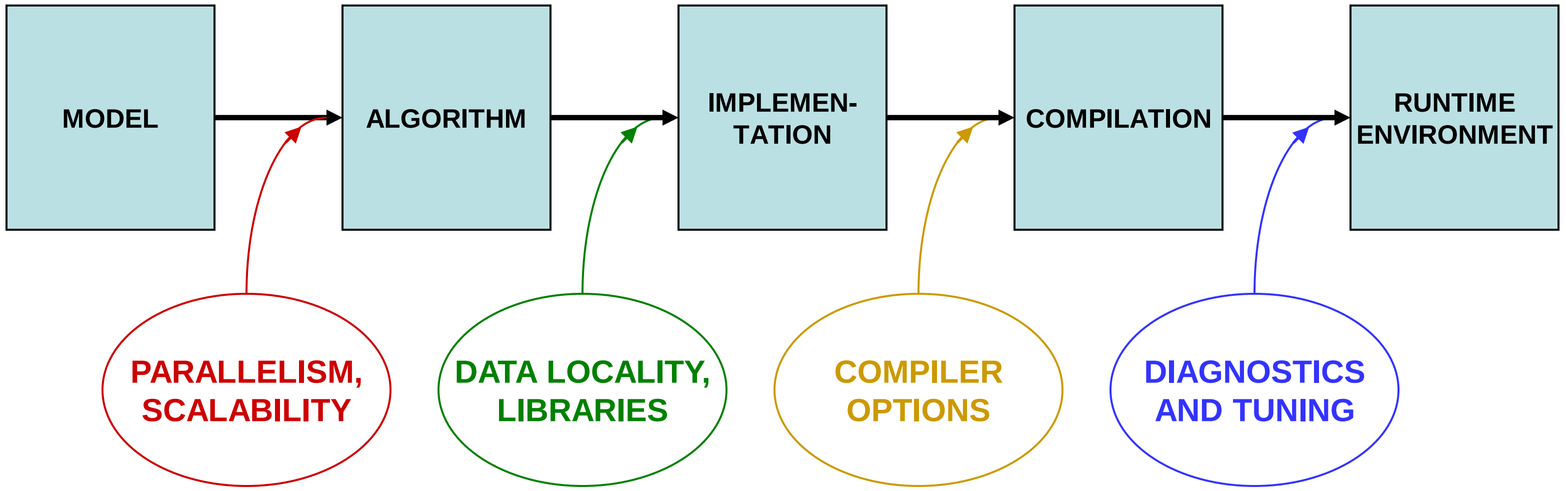
Modeled speedup (purple) could be worse than Amdahl's Law (blue) due to the overhead of message passing. Look for better strategies.



Petascale with MPI?

- Favor local communications over global
  - Nearest-neighbor is fine
  - All-to-all can be trouble
- Avoid frequent synchronization
  - Load imbalances show up as synchronization penalties
  - Even random, brief system interruptions (“jitter” or “noise”) can effectively cause load imbalances
  - Balancing must become ever more precise as the number or processes increases

Putting Performance into Development: Libraries



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *development* that lead to better performance on a per-core basis

What Matters Most in Per-Core Performance

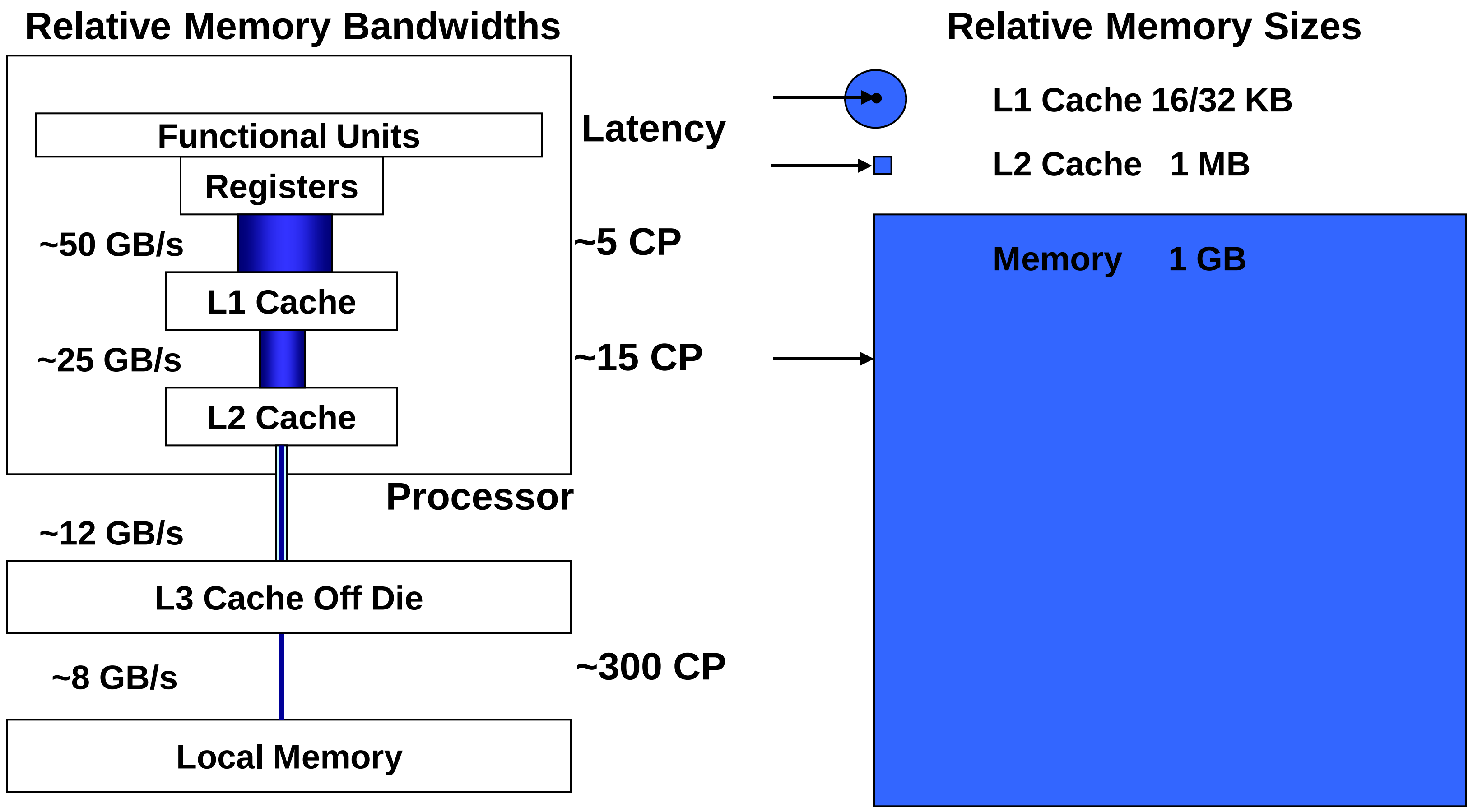
Good memory locality!

- Code accesses ***contiguous, stride-one*** memory addresses
  - Reason: data always arrive in cache lines which include neighbors
  - Reason: loops become vectorizable via SSE (explained in a moment)
- Code emphasizes ***cache reuse***
  - Reason: if multiple operations on a data item are grouped together, the item remains in cache, where access is much faster than RAM
- Data are ***aligned*** on doubleword boundaries
  - Reason: items won't straddle cache lines, so access is more efficient

Goal: make your data stay in cache as long as possible, so that deeper levels of the memory hierarchy are accessed infrequently

- The above is even more important for GPUs than it is for CPUs

Understanding The Memory Hierarchy



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/ints 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
  - New with "Sandy Bridge": Advanced Vector Extensions (AVX), Intel's latest add-ons to the x64 instruction set for 256-bit registers

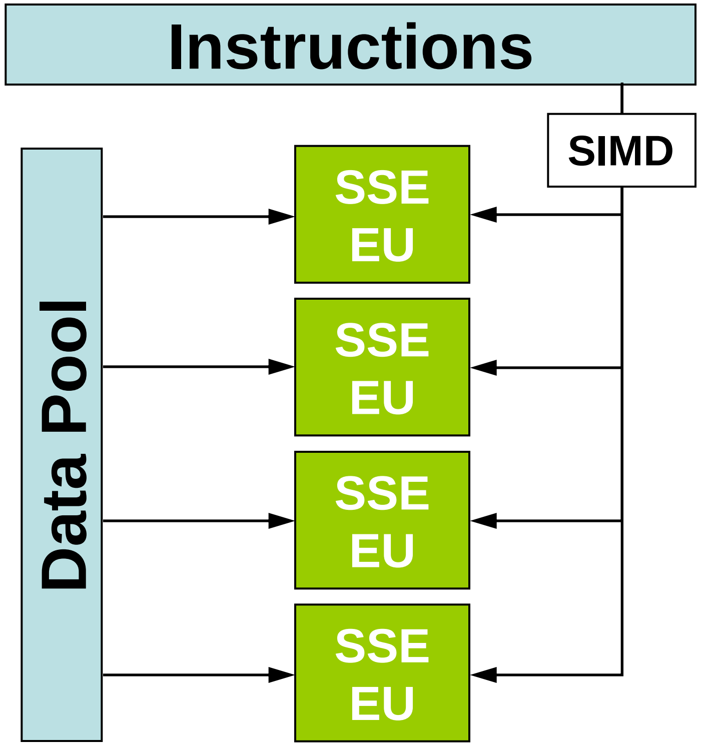
Understanding SSE, SIMD, and Micro-Parallelism

- 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



Performance Libraries

- Optimized for specific architectures (chip + platform + system)
- Offered by different vendors
  - Intel Math Kernel Library (MKL – Ranger and Lonestar)
  - AMD Core Math Library (ACML – Ranger only)
  - ESSL/PESSL 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 merely 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

Applications	Parallel Libs	Math Libs	Input/Output	Diagnostics
AMBER	PETSc	MKL	NetCDF	TAU
NAMD	SLEPc	ACML	HDF5	PAPI
GROMACS		GSL		…
	PLAPACK	GotoBLAS	pNetCDF	
GAMESS	ScaLAPACK	GotoBLAS2	PHDF5	
NWChem			…	
…	METIS	FFTW(2/3)		
	ParMETIS	ATLAS		
	SPRNG	Hypre		
	…	NumPy		
		…		

Intel MKL 10 (Math Kernel Library)

- Accompanies Intel compilers:
  - Ranger has MKL 10.0 for the Intel 10.1 compilers
  - Lonestar has MKL 10.3 for the Intel 11.1 compilers
- 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 transcendentals
  - …others

Using Intel MKL on Ranger

- Enable MKL
  - module load mkl
  - module help mkl
- Compile and link for C/C++ or Fortran

```
mpicc -I$TACC_MKL_INC mkl_test.c -L$TACC_MKL_LIB -lmkl_em64t
mpif90 mkl_test.f90 -L$TACC_MKL_LIB -lmkl_em64t
```

- Add one more option to run the code without “module load mkl”

```
-Wl,-rpath,$TACC_MKL_LIB
```

- Useful website (visit here for Lonestar, e.g.):
  - http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/

GotoBLAS, ATLAS, and FFTW

GotoBLAS

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

ATLAS, the Automatically Tuned Linear Algebra Software

- BLAS plus some LAPACK

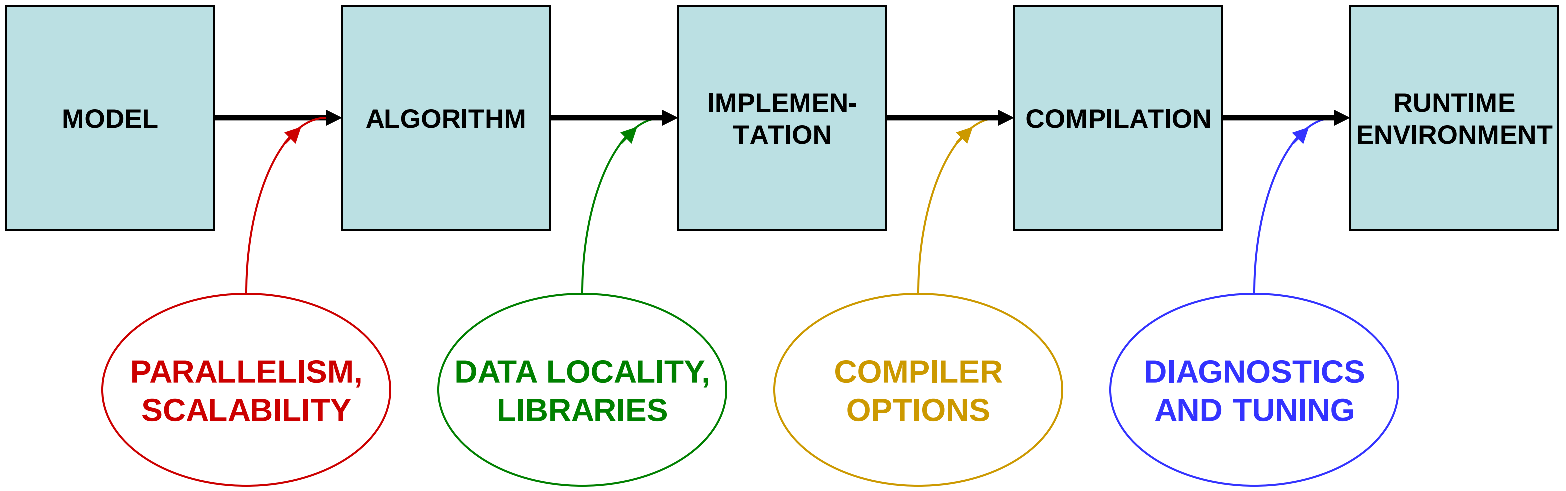
FFTW, the Fastest Fourier Transform in the West

- Cooley-Tukey with automatic performance adaptation
- Prime Factor algorithm, best with small primes like (2, 3, 5, and 7)
- The FFTW interface can also be linked against MKL

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



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *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: -O*n*

- -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

Architecture: 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
  - Do this: `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
- -xSSE4.2 is appropriate for Lonestar

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 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 – *make sure your innermost loops are vectorized*
-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	multithread 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

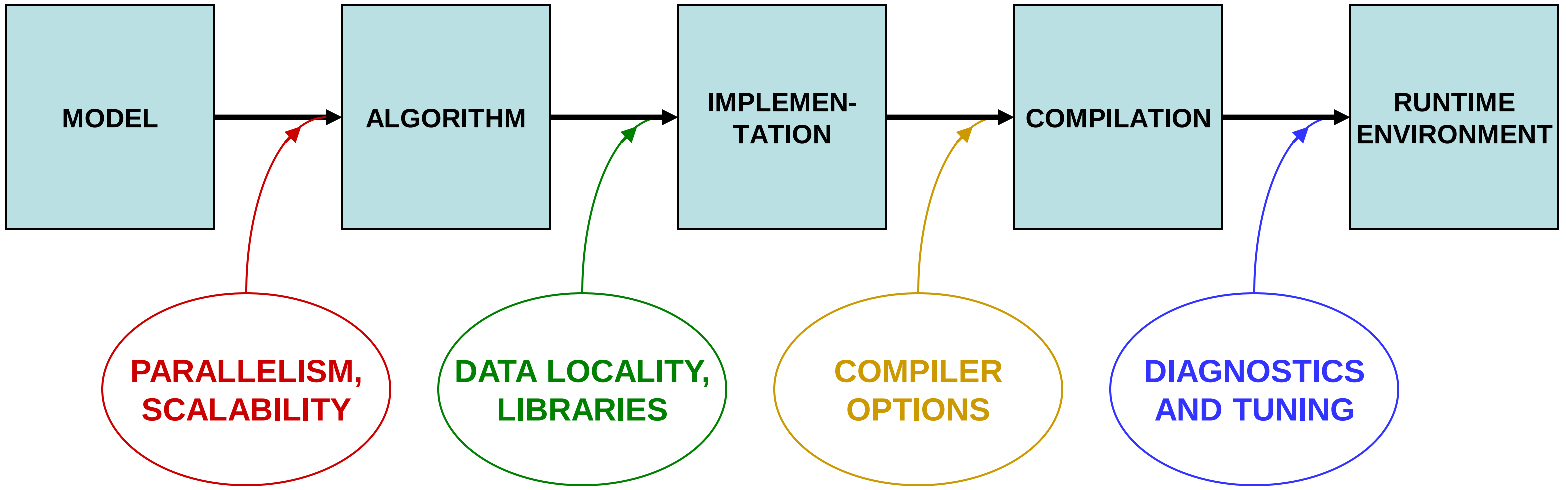
- Normal compiling for Ranger
  - Intel:
    `icc/ifort -O3 -ipo -xW prog.c/cc/f90`
  - PGI:
    `pgcc/pgcpp/pgf95 -fast -tp barcelona-64 prog.c/cc/f90`
  - GNU:
    `gcc -O3 -fast -xipo -mtune=barcelona -march=barcelona prog.c`

- -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. Libraries

- Challenge: how fast can we do a linear solve via LU decomposition?
- Naïve code is copied from Numerical Recipes
- Two alternative codes are based on calls to GSL and LAPACK
  - LAPACK references can be resolved by linking to an optimized library like AMD's ACML or Intel's MKL
- We want to compare the timings of these codes when compiled with different compilers and optimizations
  - Compile the codes with different flags, including "-g", "-O2", "-O3"
  - Submit a job to see how fast the codes run
  - Recompile with new flags and try again
  - Can even try to use the libraries' built-in OpenMP multithreading
- Source sits in ~tg459572/LABS/ludecomp.tgz

Putting Performance into Development: Tuning



Starting with how to *design* for parallelism and scalability…

…this talk is about the principles and practices during various stages of code *development* that lead to better performance on a per-core basis

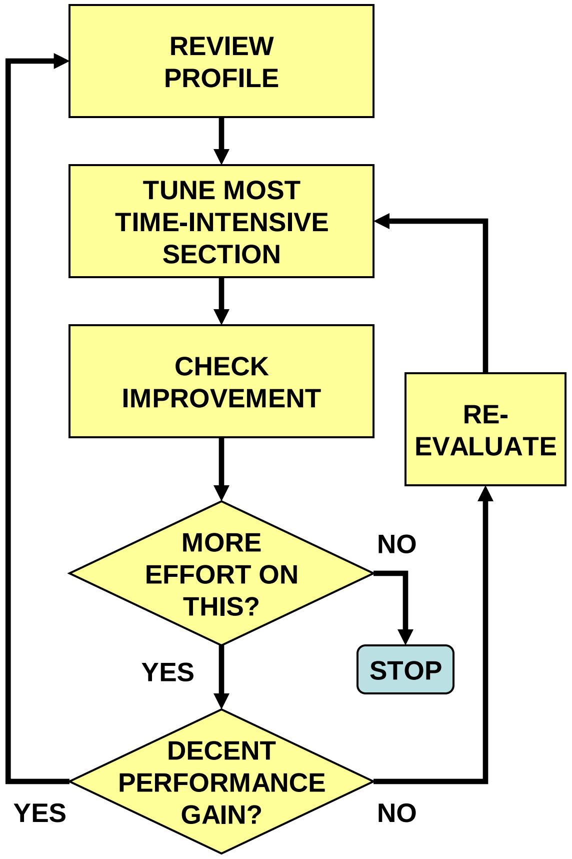
In-Depth vs. Rough Tuning

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



First Rule of Thumb: Minimize Your Stride

- Minimize stride length
  - 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…)

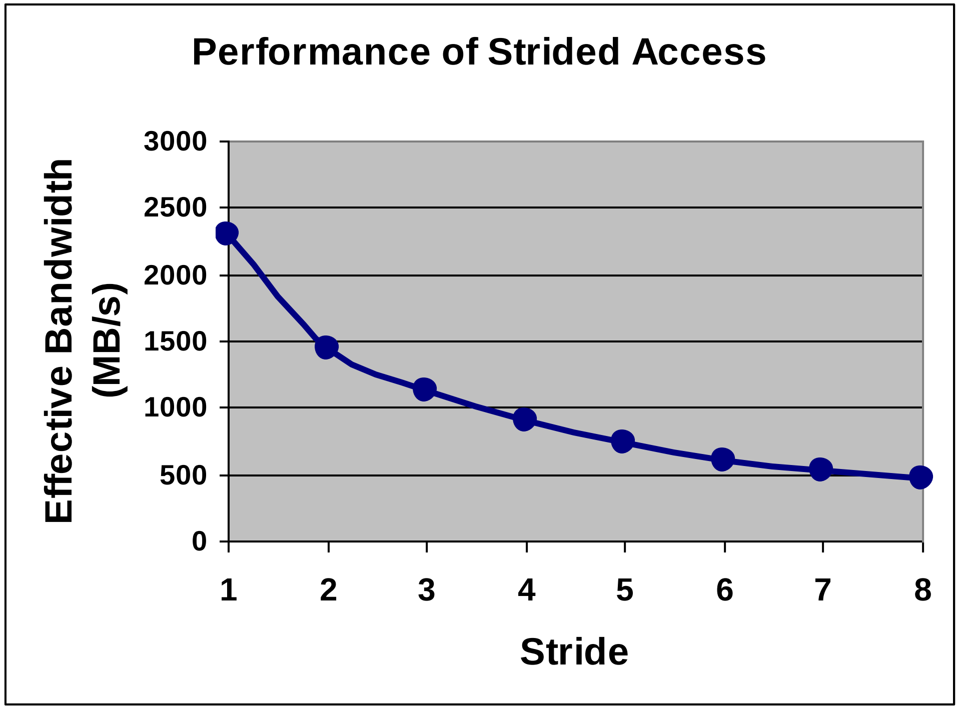
G4/5	**Velocity Engine (SIMD)**
Intel/AMD	**MMX, SSE, SSE2, SSE3 (SIMD)**
Cray	**Vector Units**

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:

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



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:

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:

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];
   }
}
```

Second Rule of Thumb: Inline Your Functions

- 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
```

Low-overhead loop *j* executes *niter* times

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

Best Practices from the Ranger User Guide

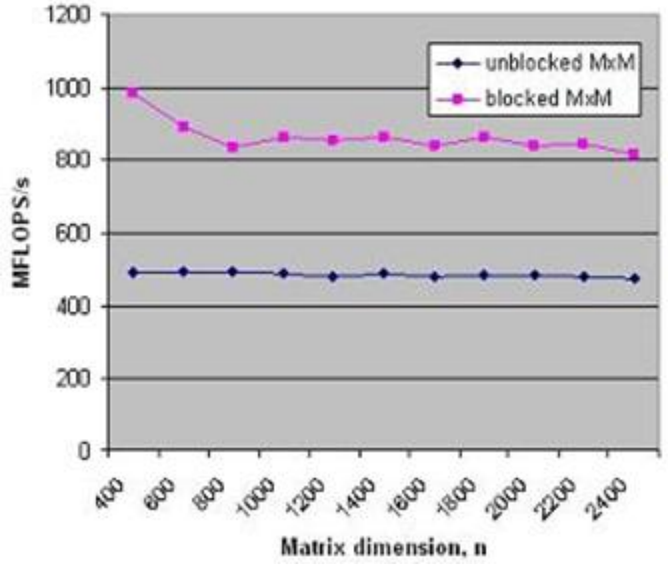
- Avoid excessive program modularization (i.e. too many functions/subroutines)
  - Write routines that can be inlined
  - Use macros and parameters whenever possible
- Minimize the use of pointers
- Avoid casts or type conversions, implicit or explicit
  - Conversions involve moving data between different 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 or MPI-IO to improve the I/O performance
- These techniques are explained in further detail, with examples, in the Memory Subsystem Tuning section of the Lonestar User Guide:
  - http://www.tacc.utexas.edu/user-services/user-guides/lonestar-user-guide#tuning

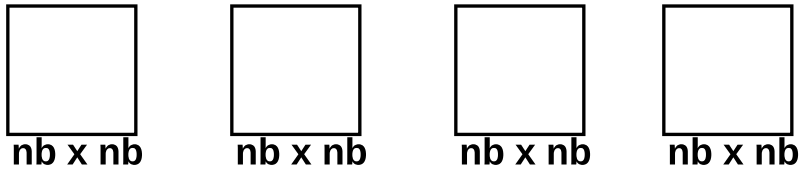
Array Blocking, or Loop Tiling, to Fit Cache

Example: matrix-matrix multiplication



```
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)
```



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