Program Optimization and Parallel Libraries

Dan Stanzione

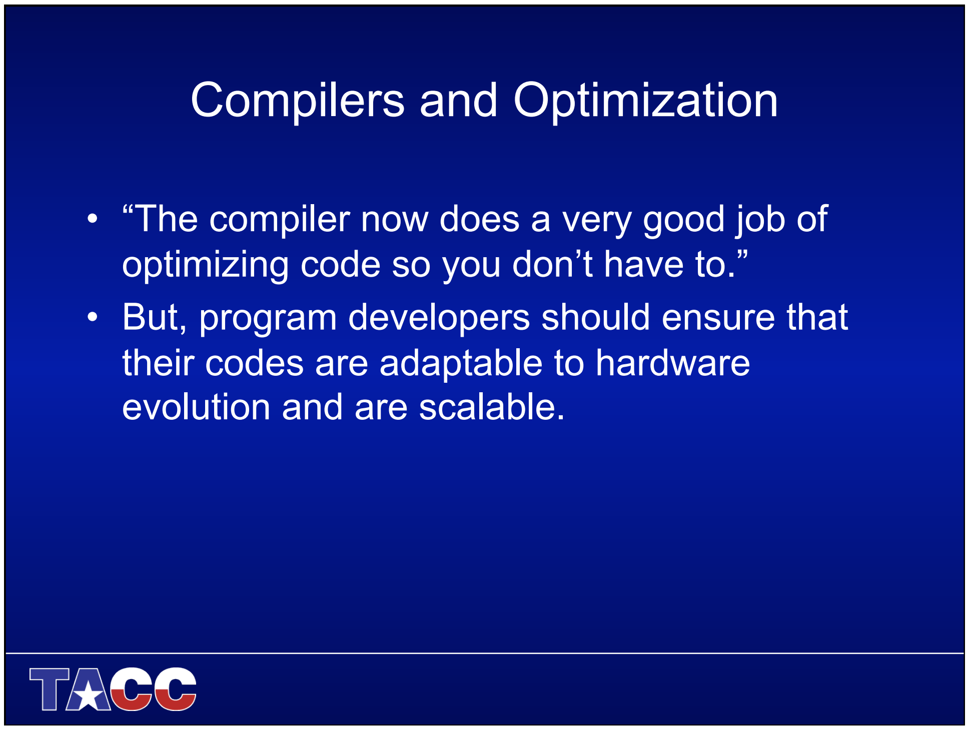
Arizona State University

July 14, 2008

Outline

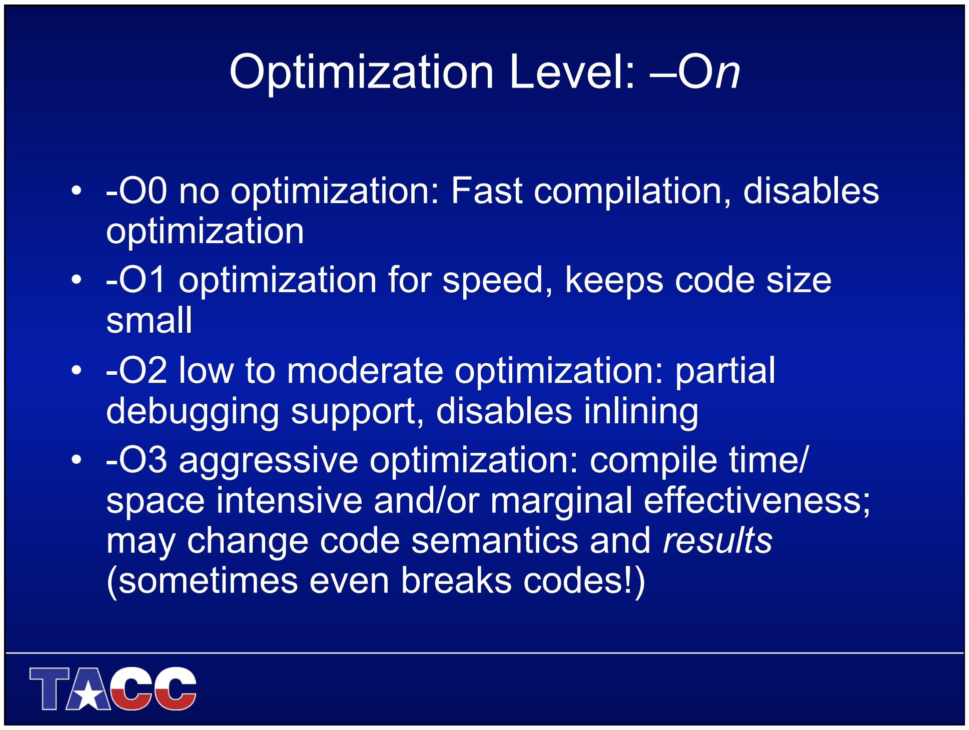
Two talks in one:

- EM64T/Opteron compiler optimizations
1. Using Parallel Libraries



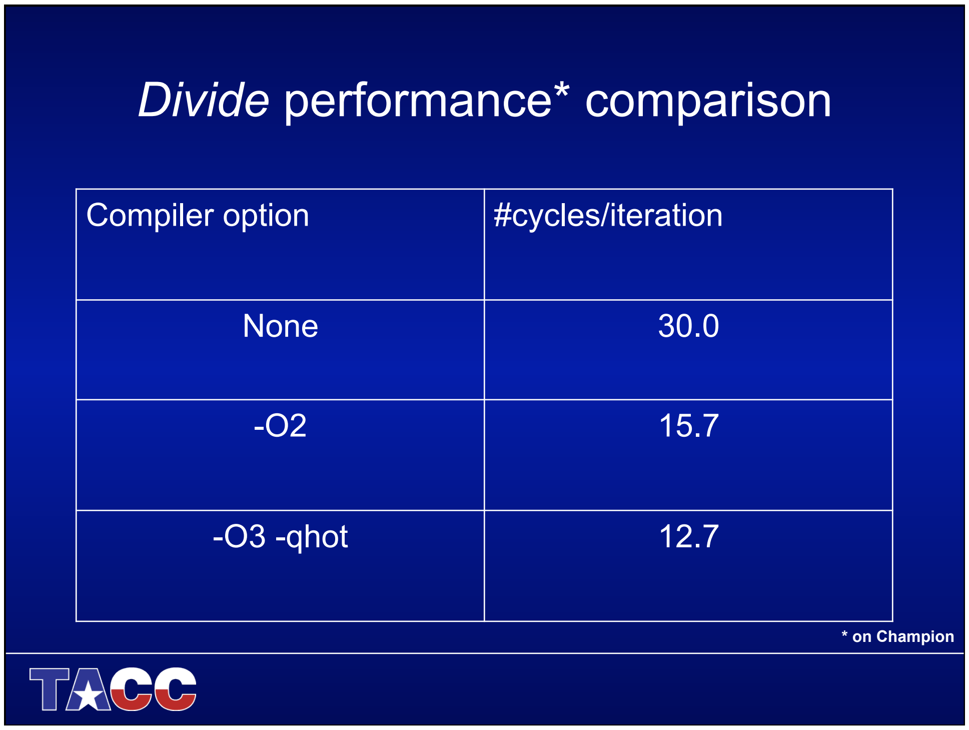
Compilers and Optimization

- “The compiler now does a very good job of optimizing code so you don’t have to.”
- But, program developers should ensure that their codes are adaptable to hardware evolution and are scalable.



Optimization Level: –O*n*

- -O0 no optimization: Fast compilation, disables optimization
- -O1 optimization for speed, keeps code size small
- -O2 low to moderate optimization: partial debugging support, disables inlining
- -O3 aggressive optimization: compile time/ space intensive and/or marginal effectiveness; may change code semantics and *results* (sometimes even breaks codes!)

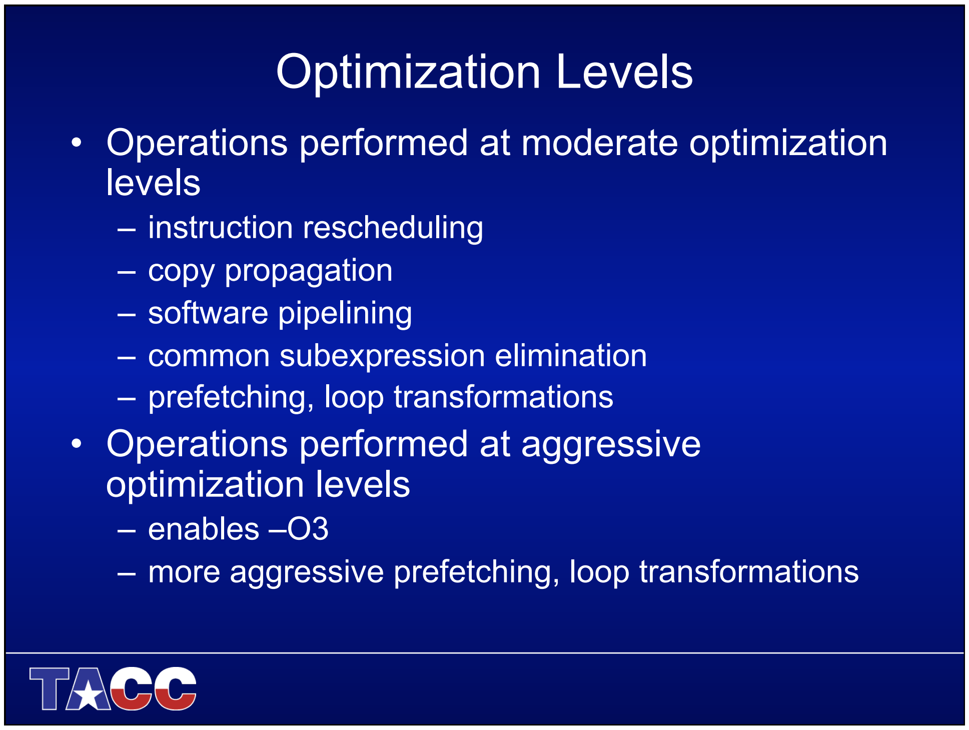


Divide performance* comparison

Compiler option	#cycles/iteration
None	30.0
-O2	15.7
-O3 -qhot	12.7

* on Champion

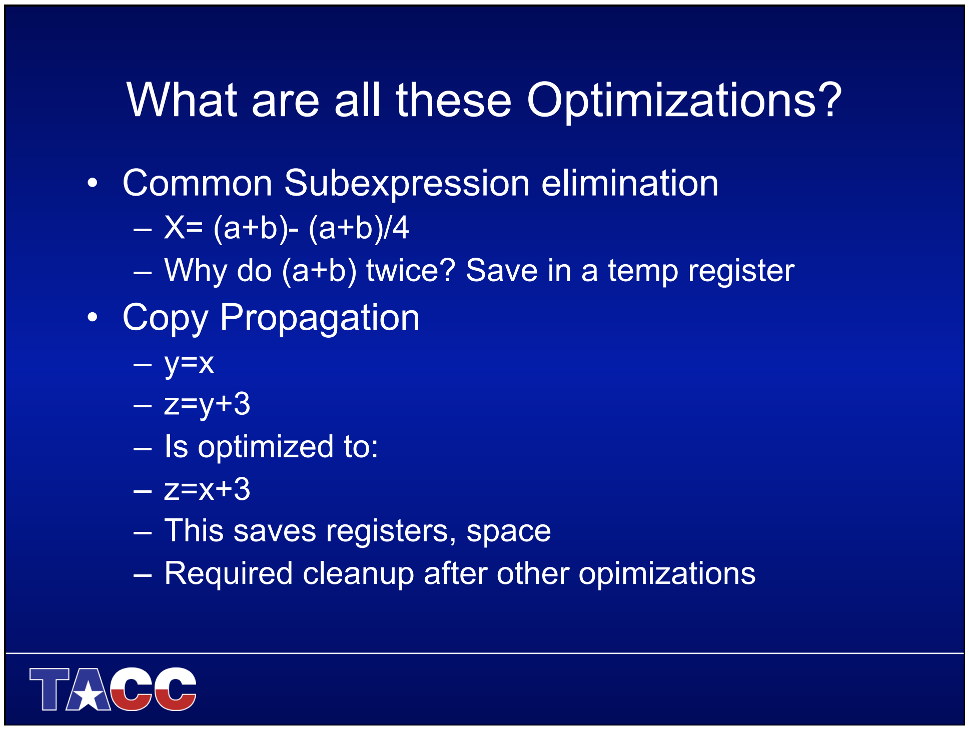
TACC



Optimization Levels

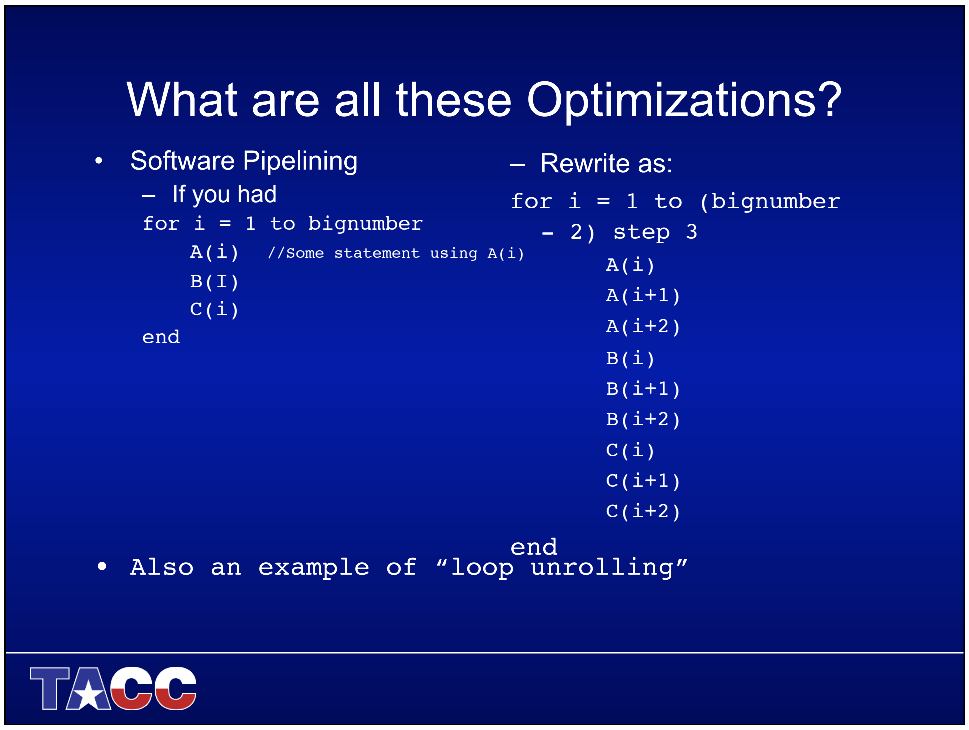
- Operations performed at moderate optimization levels
  - instruction rescheduling
  - copy propagation
  - software pipelining
  - common subexpression elimination
  - prefetching, loop transformations
- Operations performed at aggressive optimization levels
  - enables –O3
  - more aggressive prefetching, loop transformations

TACC



What are all these Optimizations?

- Common Subexpression elimination
  - X= (a+b)- (a+b)/4
  - Why do (a+b) twice? Save in a temp register
- Copy Propagation
  - y=x
  - z=y+3
  - Is optimized to:
  - z=x+3
  - This saves registers, space
  - Required cleanup after other opimizations



What are all these Optimizations?

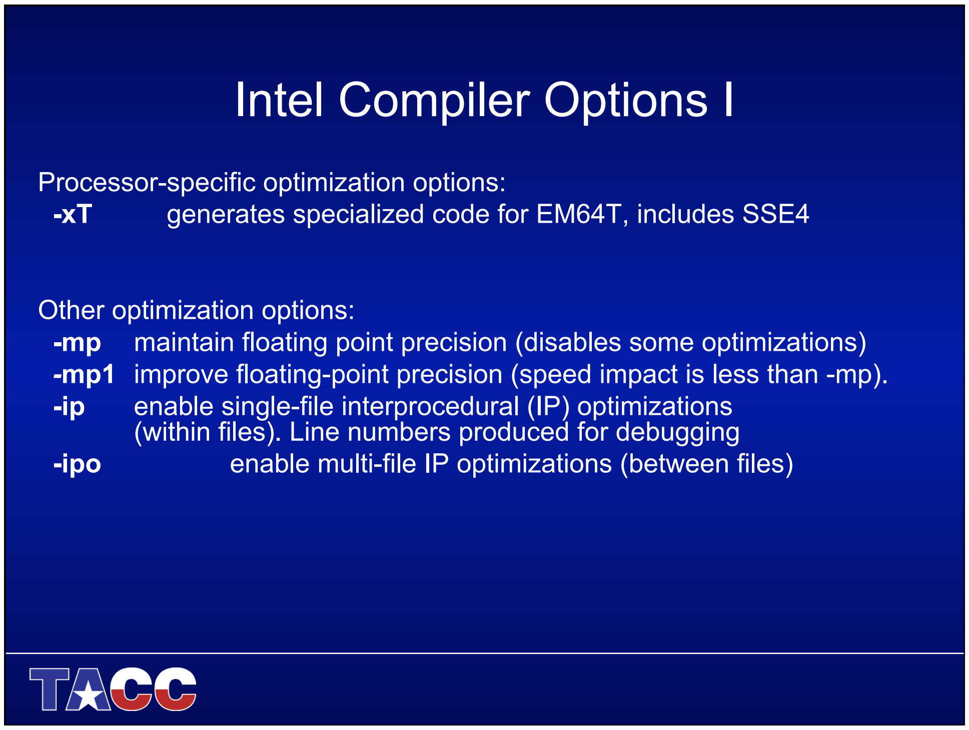
- Software Pipelining
  - If you had

```
for i = 1 to bignumber
    A(i)   //Some statement using A(i)
    B(I)
    C(i)
end
```

  - Rewrite as:

```
for i = 1 to (bignumber
  - 2) step 3
        A(i)
        A(i+1)
        A(i+2)
        B(i)
        B(i+1)
        B(i+2)
        C(i)
        C(i+1)
        C(i+2)
end
```

- Also an example of “loop unrolling”



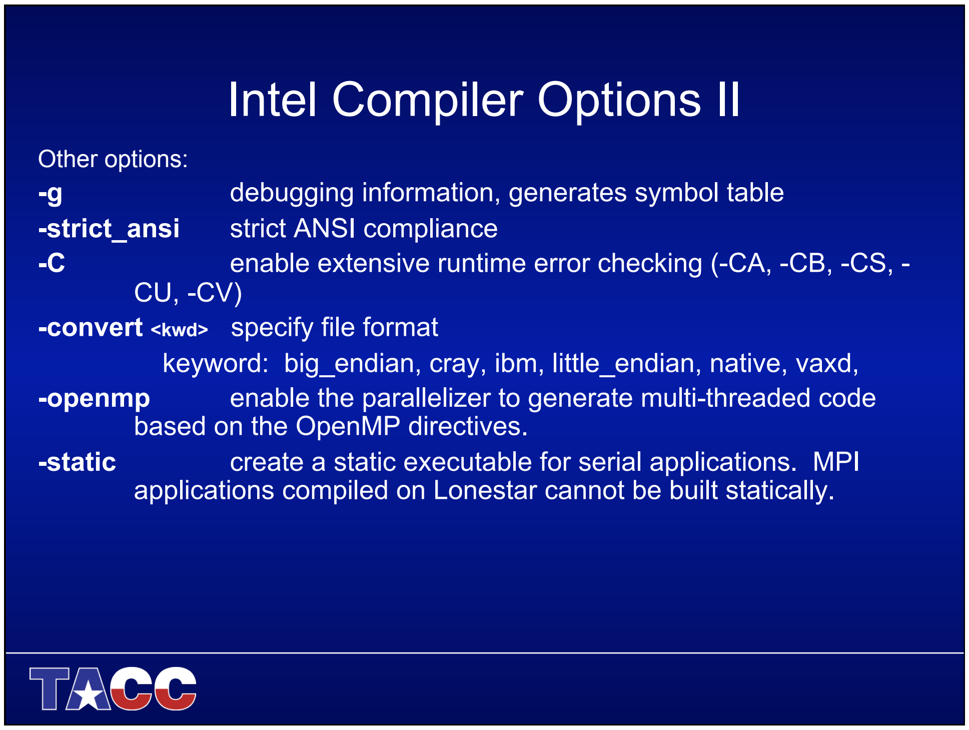
Intel Compiler Options I

Processor-specific optimization options:

-xT	generates specialized code for EM64T, includes SSE4

Other optimization options:

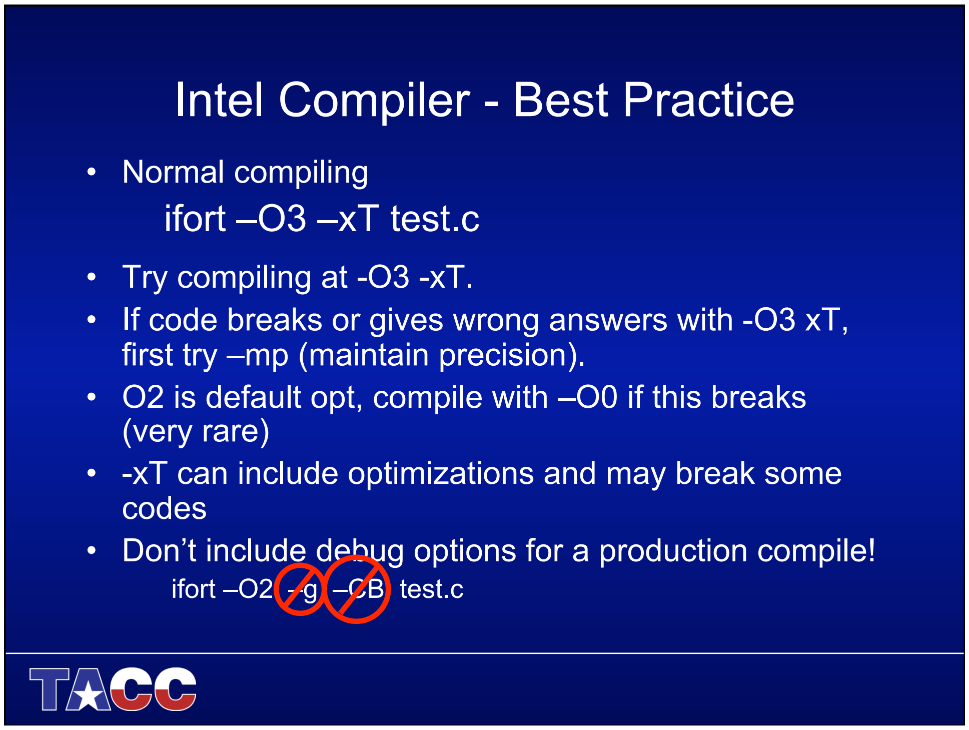
-mp	maintain floating point precision (disables some optimizations)
-mp1	improve floating-point precision (speed impact is less than -mp).
-ip	enable single-file interprocedural (IP) optimizations (within files). Line numbers produced for debugging
-ipo	enable multi-file IP optimizations (between files)



Intel Compiler Options II

Other options:

-g	debugging information, generates symbol table
-strict_ansi	strict ANSI compliance
-C	enable extensive runtime error checking (-CA, -CB, -CS, -CU, -CV)
-convert <kwd>	specify file format
	keyword: big_endian, cray, ibm, little_endian, native, vaxd,
-openmp	enable the parallelizer to generate multi-threaded code based on the OpenMP directives.
-static	create a static executable for serial applications. MPI applications compiled on Lonestar cannot be built statically.



Intel Compiler - Best Practice

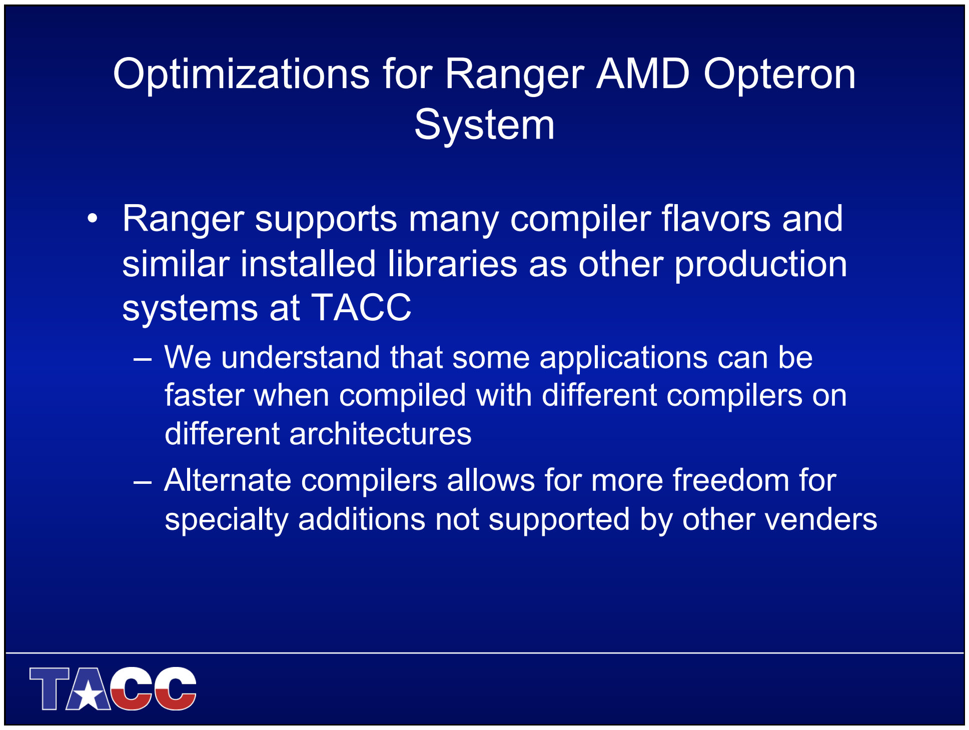
- Normal compiling

  ifort –O3 –xT test.c

- Try compiling at -O3 -xT.
- If code breaks or gives wrong answers with -O3 xT, first try –mp (maintain precision).
- O2 is default opt, compile with –O0 if this breaks (very rare)
- -xT can include optimizations and may break some codes
- Don't include debug options for a production compile!

  ifort –O2 ~~–g~~ ~~–CB~~ test.c

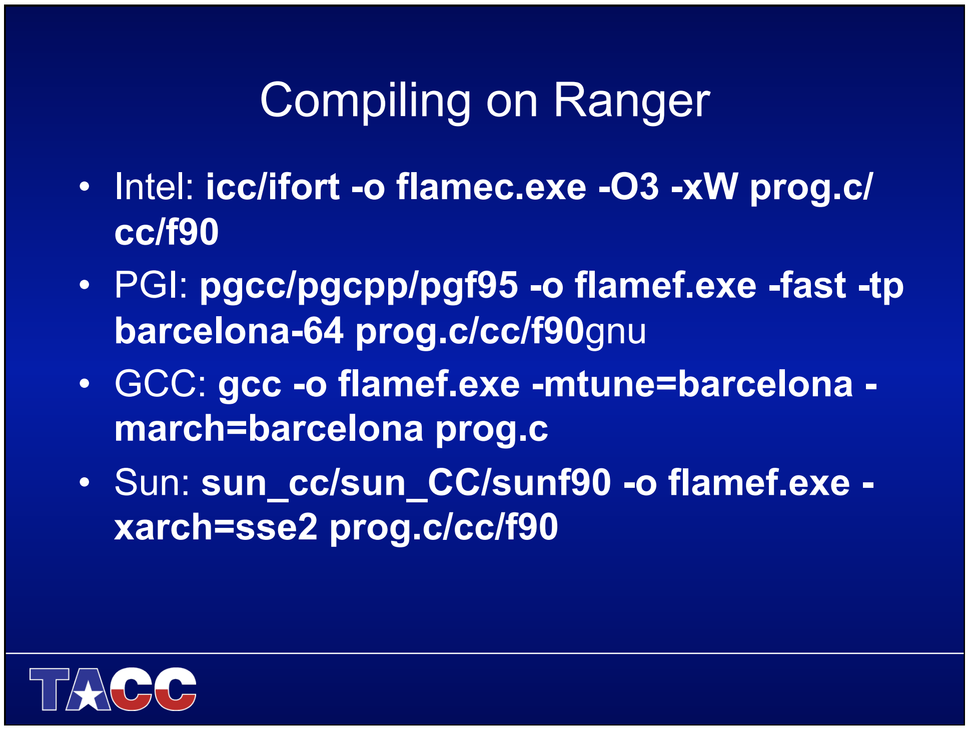
TACC



Optimizations for Ranger AMD Opteron System

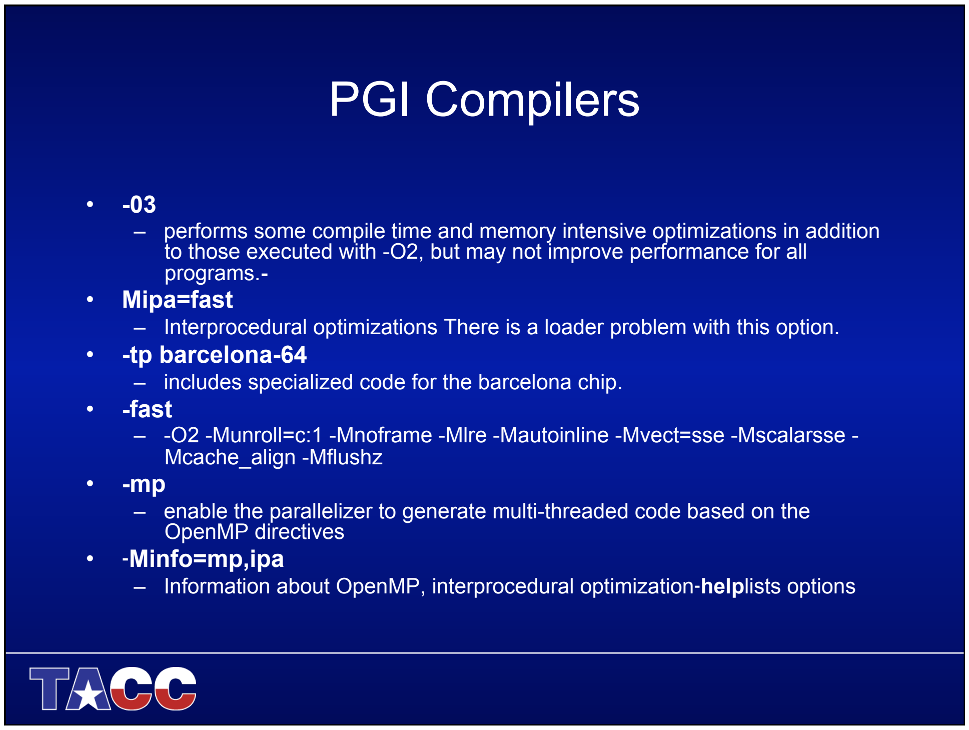
- Ranger supports many compiler flavors and similar installed libraries as other production systems at TACC
  - We understand that some applications can be faster when compiled with different compilers on different architectures
  - Alternate compilers allows for more freedom for specialty additions not supported by other venders

TACC



Compiling on Ranger

- Intel: **icc/ifort -o flamec.exe -O3 -xW prog.c/ cc/f90**
- PGI: **pgcc/pgcpp/pgf95 -o flamef.exe -fast -tp barcelona-64 prog.c/cc/f90**gnu
- GCC: **gcc -o flamef.exe -mtune=barcelona -march=barcelona prog.c**
- Sun: **sun_cc/sun_CC/sunf90 -o flamef.exe -xarch=sse2 prog.c/cc/f90**



PGI Compilers

- **-03**
  - performs some compile time and memory intensive optimizations in addition to those executed with -O2, but may not improve performance for all programs.-
- **Mipa=fast**
  - Interprocedural optimizations There is a loader problem with this option.
- **-tp barcelona-64**
  - includes specialized code for the barcelona chip.
- **-fast**
  - -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align -Mflushz
- **-mp**
  - enable the parallelizer to generate multi-threaded code based on the OpenMP directives
- -**Minfo=mp,ipa**
  - Information about OpenMP, interprocedural optimization-**help**lists options

Tuning Parameters

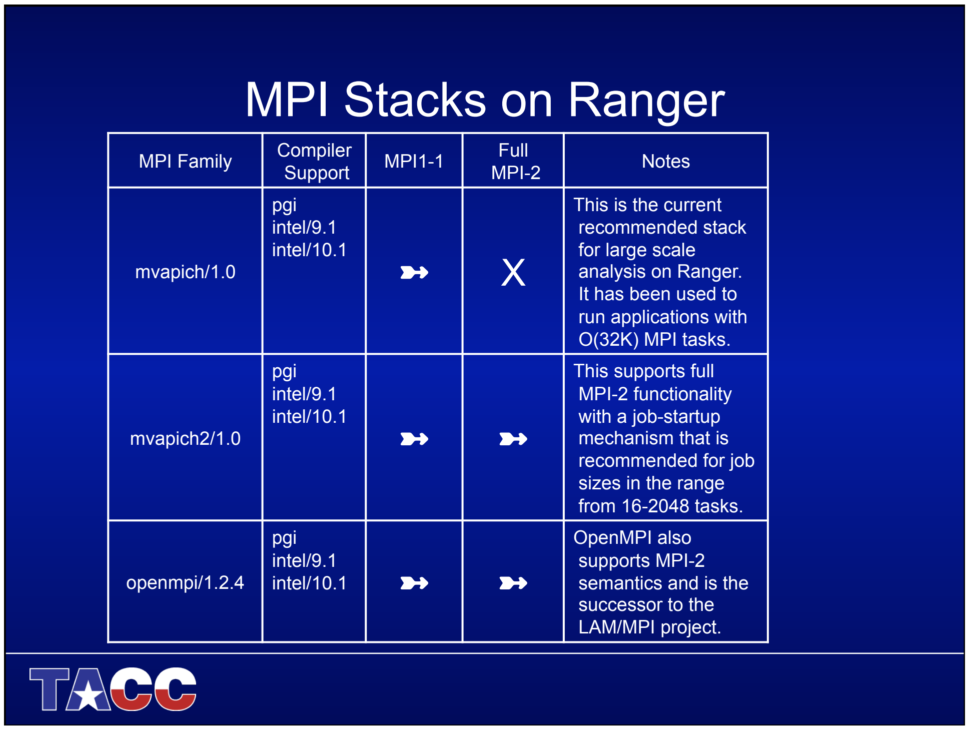
- Different for each compiler. Listed in tables under the ranger userguide
  – http://www.tacc.utexas.edu/services/userguides/ranger/

TACC

EM64T References

- High Performance Computing by Kevin Dowd and Charles Severance (O'Reilly book) -- general study of high performance computing
- TACC Lonestar User Guide
  www.tacc.utexas.edu/resources/userguides/
- Intel documentation for Intel compilers and MKL library
  /opt/intel/compiler9.1/$^{cc}_{fc}$/doc
  /opt/intel/mkl9.0/doc

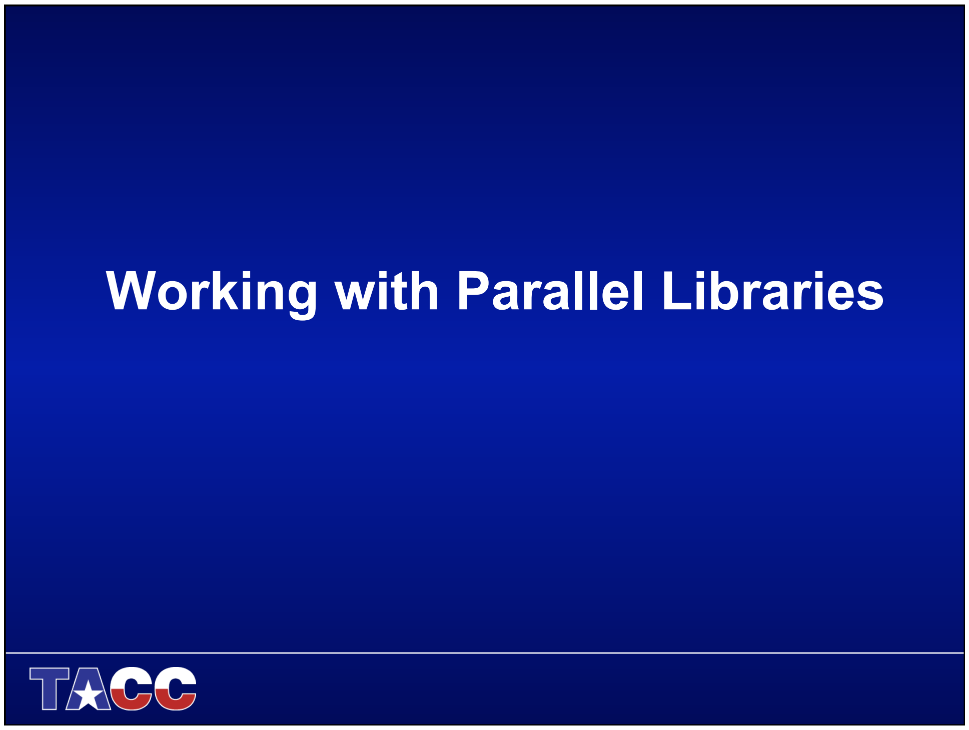
TACC



MPI Stacks on Ranger

MPI Family	Compiler Support	MPI1-1	Full MPI-2	Notes
mvapich/1.0	pgi intel/9.1 intel/10.1	➽	X	This is the current recommended stack for large scale analysis on Ranger. It has been used to run applications with O(32K) MPI tasks.
mvapich2/1.0	pgi intel/9.1 intel/10.1	➽	➽	This supports full MPI-2 functionality with a job-startup mechanism that is recommended for job sizes in the range from 16-2048 tasks.
openmpi/1.2.4	pgi intel/9.1 intel/10.1	➽	➽	OpenMPI also supports MPI-2 semantics and is the successor to the LAM/MPI project.

TACC



Working with Parallel Libraries

TACC

Why Parallel Libraries?

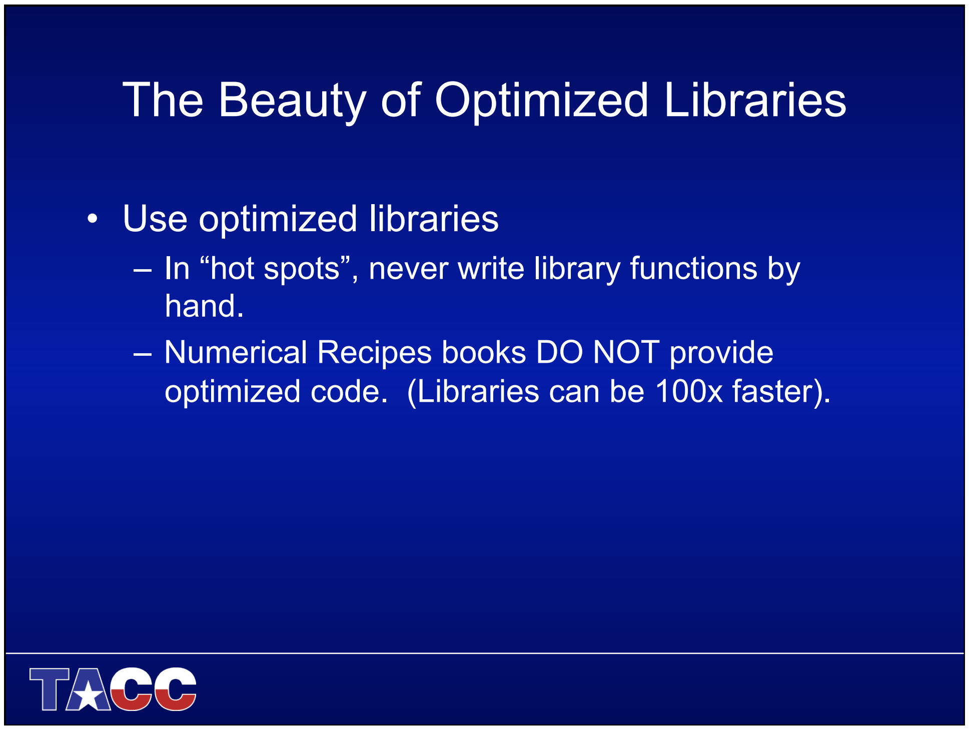
- Like most programming tasks, very little “real” software is created by starting from a blank slate and coding every line of every algorithm (as presented in most classes, including this one).
- Large scale parallel software construction involves significant code reuse, making use of libraries that encapsulate much of what we learned.

TACC

Performance Libraries

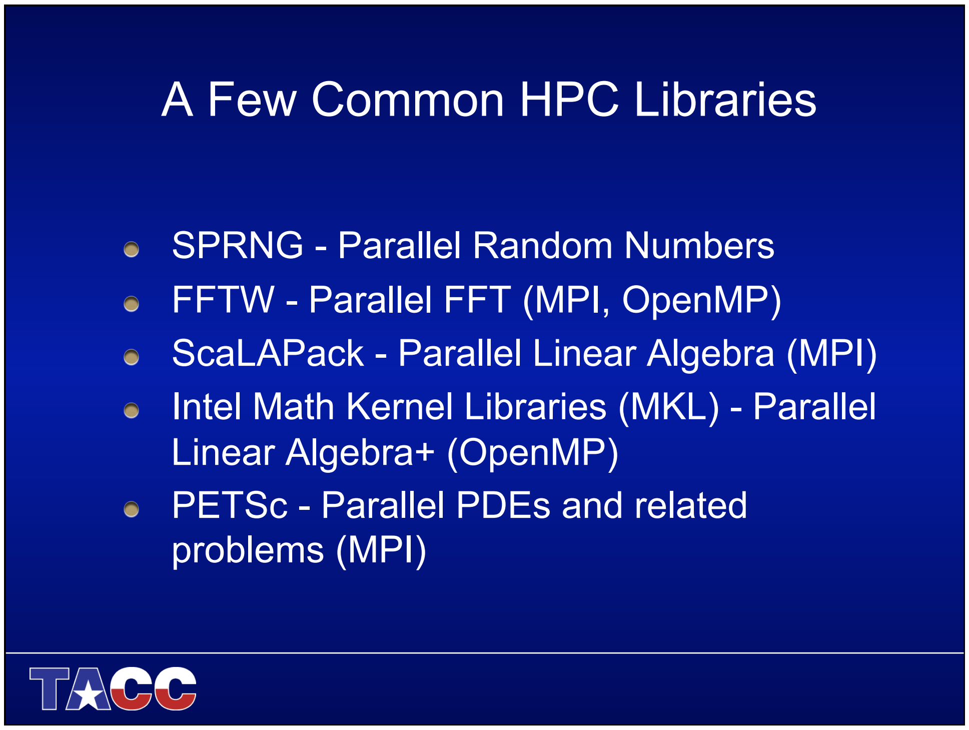
- Optimized for specific architectures
- Use library routines instead of hand-coding your own
- Offered by different vendors (ESSL/PESSL on IBM systems, Intel MKL for IA32, EM64T and IA64, Cray libsci for Cray systems, SCSL for SGI, ACML for AMD)

TACC



The Beauty of Optimized Libraries

- Use optimized libraries
  - In “hot spots”, never write library functions by hand.
  - Numerical Recipes books DO NOT provide optimized code. (Libraries can be 100x faster).



A Few Common HPC Libraries

- SPRNG - Parallel Random Numbers
- FFTW - Parallel FFT (MPI, OpenMP)
- ScaLAPack - Parallel Linear Algebra (MPI)
- Intel Math Kernel Libraries (MKL) - Parallel Linear Algebra+ (OpenMP)
- PETSc - Parallel PDEs and related problems (MPI)

FFTW

Fastest Fourier Transform in the West

- Supports MPI or OpenMP parallelization (through different interfaces)
- 1D, Multi-D, Real or Complex routines
- Well-established, widely used and tested FFTs that generally are considered the fastest
- Relies heavily on MPI_AlltoAll performance

FFTW - Sequential Interface

```
#include <fftw.h>
...
{
     fftw_complex in[N], out[N];
     fftw_plan p;
     ...
     p = fftw_create_plan(N, FFTW_FORWARD, FFTW_ESTIMATE);
     ...
     fftw_one(p, in, out);
     ...
     fftw_destroy_plan(p);
}
```

- The “plan” is reusable, and is a data structure containing the layout for the FFT (setting it up in advance is one of the things that makes it so fast).
- N is the FFT size, forward is the direction, and the last argument is how to build the plan: “estimate” is a best guess, “measure” will run a few different sizes and actually test at runtime for the optimal layout.

FFTW MPI Interface

- Add MPI_Init and Finalize, add communicator to “plan” argument
- FFTW done “in place”, so data ordering must be correct.

```
#include <fftw_mpi.h>

int main(int argc, char **argv)
{
      const int NX = ..., NY = ...;
      fftwnd_mpi_plan plan;
      fftw_complex *data;

      MPI_Init(&argc,&argv);

      plan = fftw2d_mpi_create_plan(MPI_COMM_WORLD,
                                    NX, NY,
                                    FFTW_FORWARD, FFTW_ESTIMATE);

      ...allocate and initialize data...

      fftwnd_mpi(p, 1, data, NULL, FFTW_NORMAL_ORDER);

      ...

      fftwnd_mpi_destroy_plan(plan);
      MPI_Finalize();
}
```

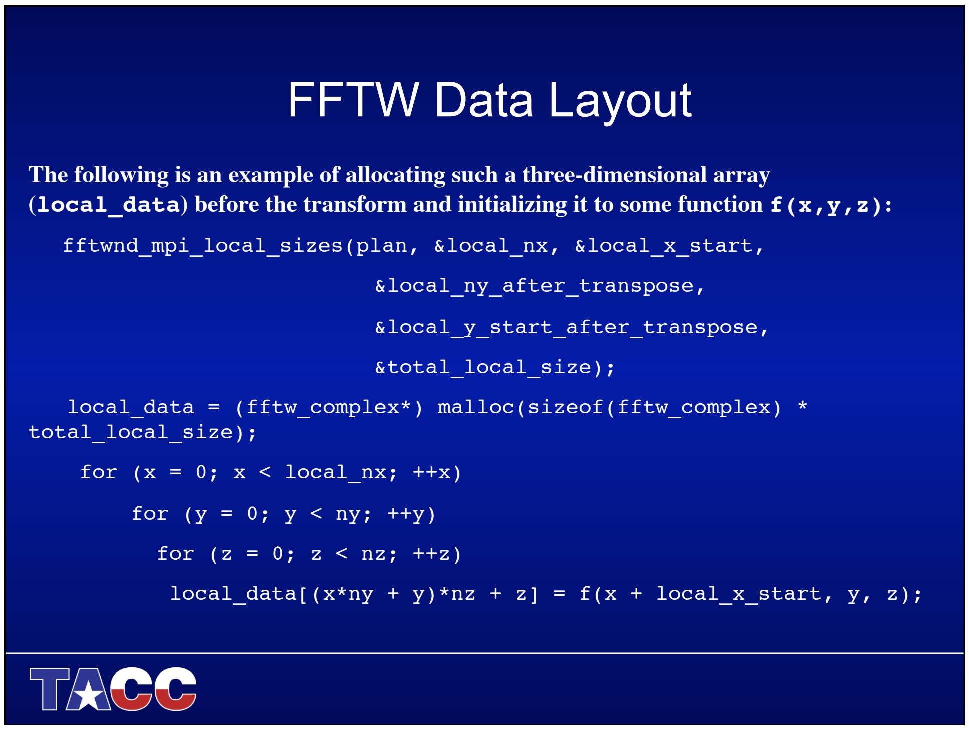
TACC

FFTW Data Layout

```
void fftwnd_mpi_local_sizes(fftwnd_mpi_plan p,
                            int *local_nx,
                            int *local_x_start,
                            int *local_ny_after_transpose,
                            int *local_y_start_after_transpose,
                            int *total_local_size);
```

- A portion of FFT data must reside locally on each processor.
- Divided by blocks of rows (first dimension)
- The function above returns the data that should/will be on the local process

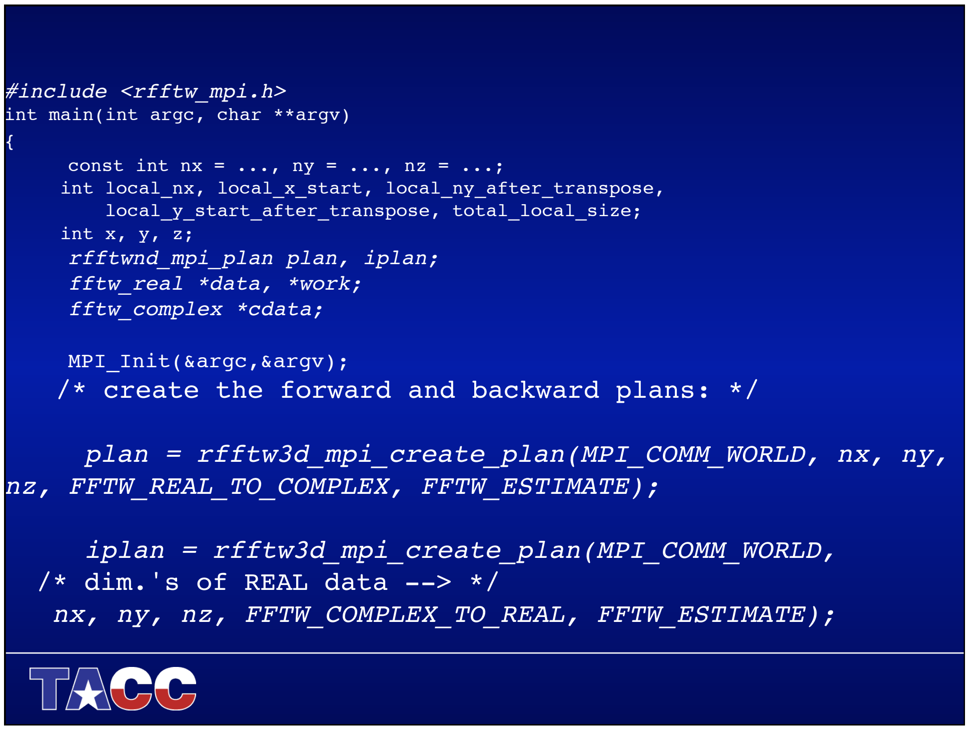
TACC



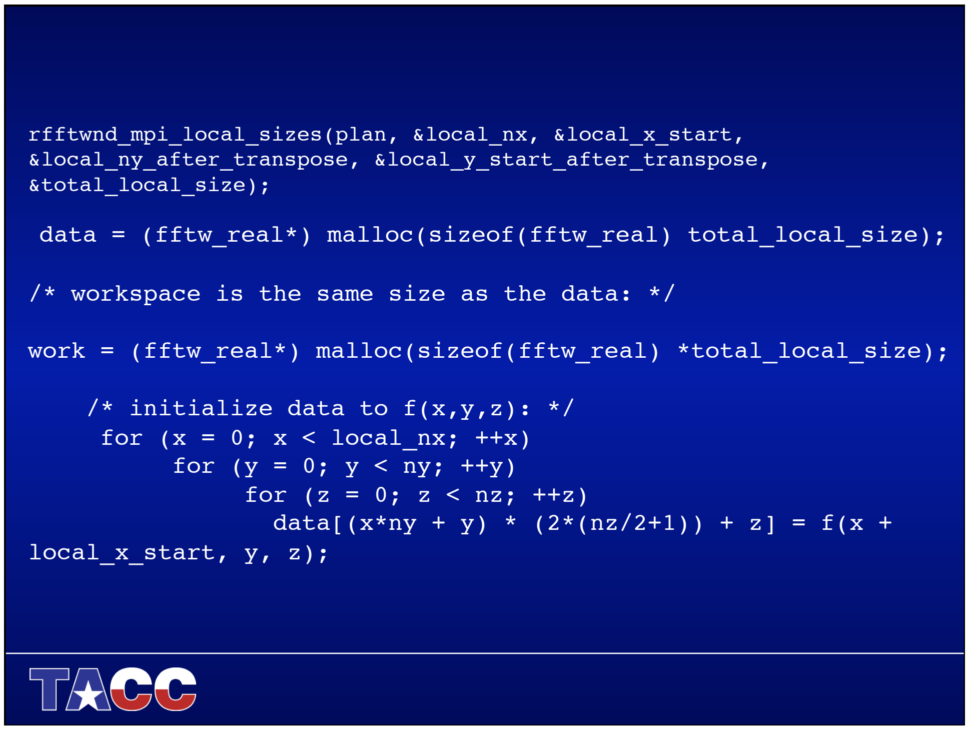
FFTW Data Layout

The following is an example of allocating such a three-dimensional array (`local_data`) before the transform and initializing it to some function `f(x,y,z)`:

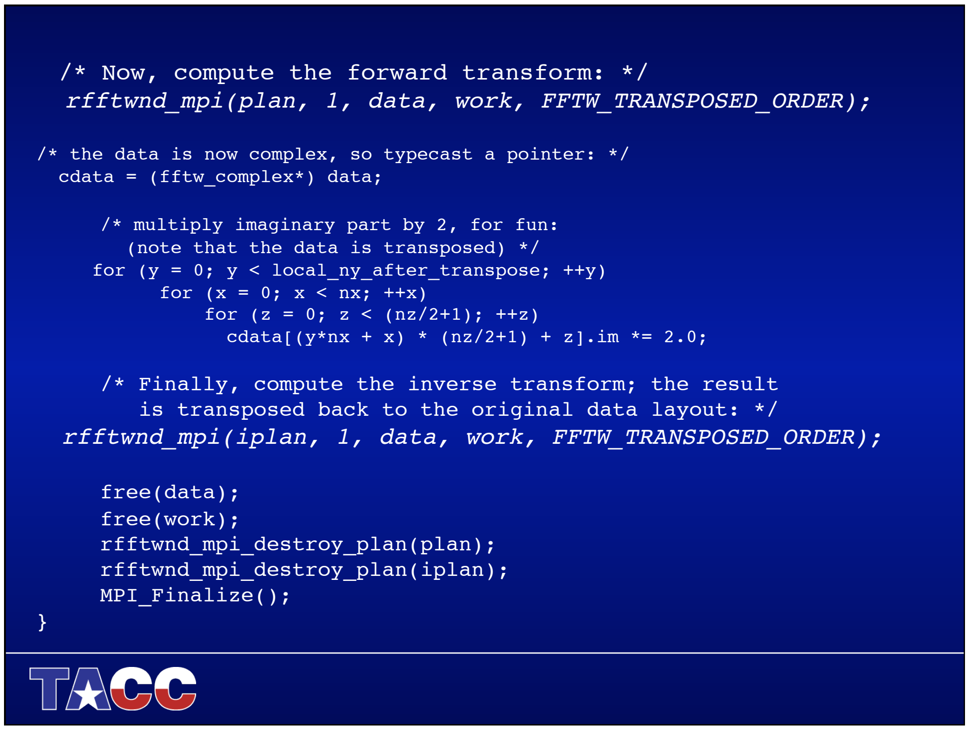
```
   fftwnd_mpi_local_sizes(plan, &local_nx, &local_x_start,
                              &local_ny_after_transpose,
                              &local_y_start_after_transpose,
                              &total_local_size);
   local_data = (fftw_complex*) malloc(sizeof(fftw_complex) *
total_local_size);
    for (x = 0; x < local_nx; ++x)
        for (y = 0; y < ny; ++y)
          for (z = 0; z < nz; ++z)
           local_data[(x*ny + y)*nz + z] = f(x + local_x_start, y, z);
```



```
#include <rfftw_mpi.h>
int main(int argc, char **argv)
{
     const int nx = ..., ny = ..., nz = ...;
     int local_nx, local_x_start, local_ny_after_transpose,
         local_y_start_after_transpose, total_local_size;
     int x, y, z;
     rfftwnd_mpi_plan plan, iplan;
     fftw_real *data, *work;
     fftw_complex *cdata;

     MPI_Init(&argc,&argv);
    /* create the forward and backward plans: */

      plan = rfftw3d_mpi_create_plan(MPI_COMM_WORLD, nx, ny,
nz, FFTW_REAL_TO_COMPLEX, FFTW_ESTIMATE);

      iplan = rfftw3d_mpi_create_plan(MPI_COMM_WORLD,
  /* dim.'s of REAL data --> */
   nx, ny, nz, FFTW_COMPLEX_TO_REAL, FFTW_ESTIMATE);
```



```
rfftwnd_mpi_local_sizes(plan, &local_nx, &local_x_start,
&local_ny_after_transpose, &local_y_start_after_transpose,
&total_local_size);

 data = (fftw_real*) malloc(sizeof(fftw_real) total_local_size);

/* workspace is the same size as the data: */

work = (fftw_real*) malloc(sizeof(fftw_real) *total_local_size);

     /* initialize data to f(x,y,z): */
      for (x = 0; x < local_nx; ++x)
           for (y = 0; y < ny; ++y)
                for (z = 0; z < nz; ++z)
                  data[(x*ny + y) * (2*(nz/2+1)) + z] = f(x +
local_x_start, y, z);
```



```
  /* Now, compute the forward transform: */
  rfftwnd_mpi(plan, 1, data, work, FFTW_TRANSPOSED_ORDER);

 /* the data is now complex, so typecast a pointer: */
   cdata = (fftw_complex*) data;

      /* multiply imaginary part by 2, for fun:
         (note that the data is transposed) */
     for (y = 0; y < local_ny_after_transpose; ++y)
          for (x = 0; x < nx; ++x)
               for (z = 0; z < (nz/2+1); ++z)
                 cdata[(y*nx + x) * (nz/2+1) + z].im *= 2.0;

      /* Finally, compute the inverse transform; the result
         is transposed back to the original data layout: */
  rfftwnd_mpi(iplan, 1, data, work, FFTW_TRANSPOSED_ORDER);

      free(data);
      free(work);
      rfftwnd_mpi_destroy_plan(plan);
      rfftwnd_mpi_destroy_plan(iplan);
      MPI_Finalize();
 }
```

FFTW MPI Tuning

- If possible, the first and second dimensions of your data should be divisible by the number of processes
  - (If only one can be divisible, then you should choose the first dimension.)
  - This allows the computational load to be spread evenly among the processes.
- You should consider the *FFTW_TRANSPOSED_ORDER* output format if it is not too burdensome.
  - The speed gains from communications savings are usually substantial.

TACC

FFTW MPI Tuning

- You should consider allocating a workspace for (r)fftw(nd)_mpi, as this can often (but not always) improve performance (at the cost of extra storage).

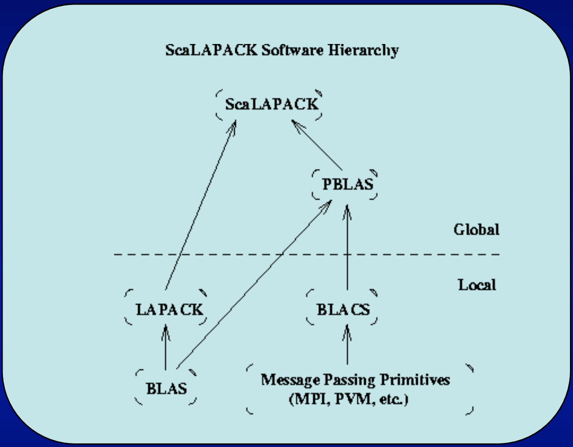
- You should experiment with the best number of processors to use for your problem.
  - (There comes a point of diminishing returns, when the communications costs outweigh the computational benefits).
  - The fftw_mpi_test program can output helpful performance benchmarks. It accepts the same parameters as the uniprocessor test programs and is run like an ordinary MPI program.
  - For example, mpirun -np 4 fftw_mpi_test -s 128x128x128 will benchmark a 128x128x128 transform on four processors,

TACC

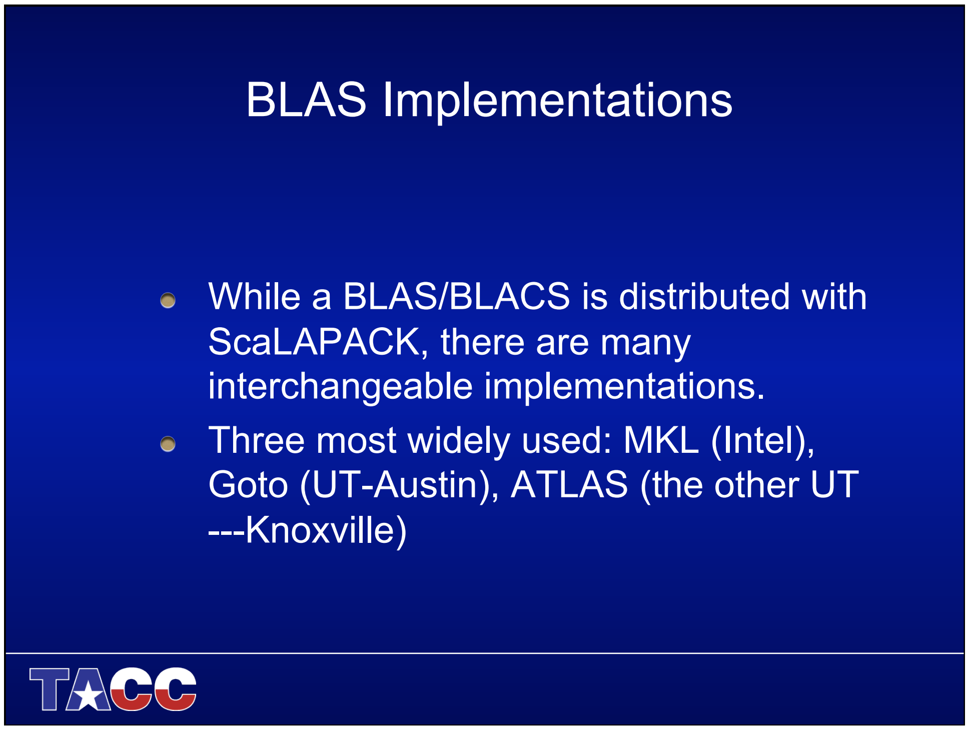
ScaLAPACK

- Scalable Linear Algebra PACKage
- MPI Extensions to the venerable LAPACK library(extended from LINPACK); most used linear algebra library of all time.
- Routines for solving systems of linear equations, least squares problems, and eigenvalue problems.
- Built on BLAS and BLACS

TACC

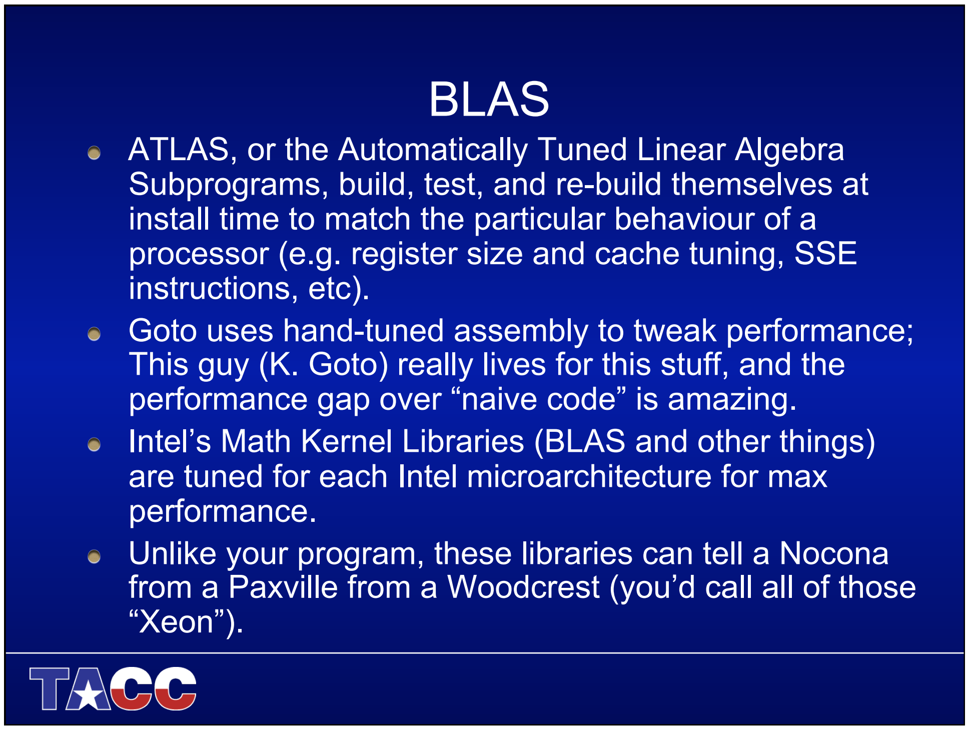


TACC



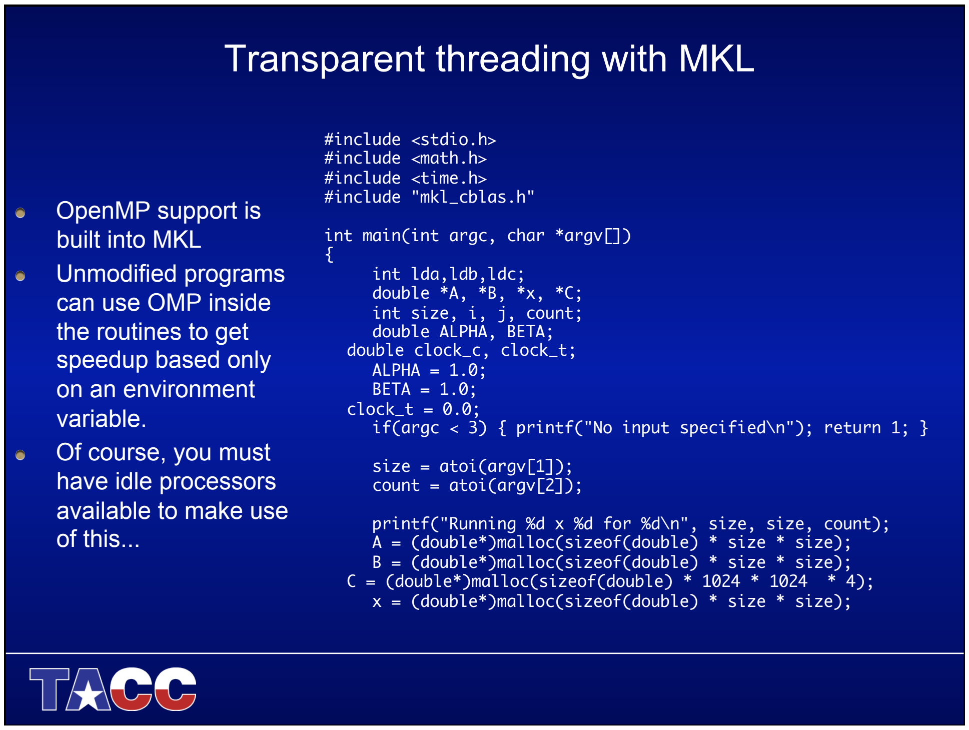
BLAS Implementations

- While a BLAS/BLACS is distributed with ScaLAPACK, there are many interchangeable implementations.
- Three most widely used: MKL (Intel), Goto (UT-Austin), ATLAS (the other UT ---Knoxville)



BLAS

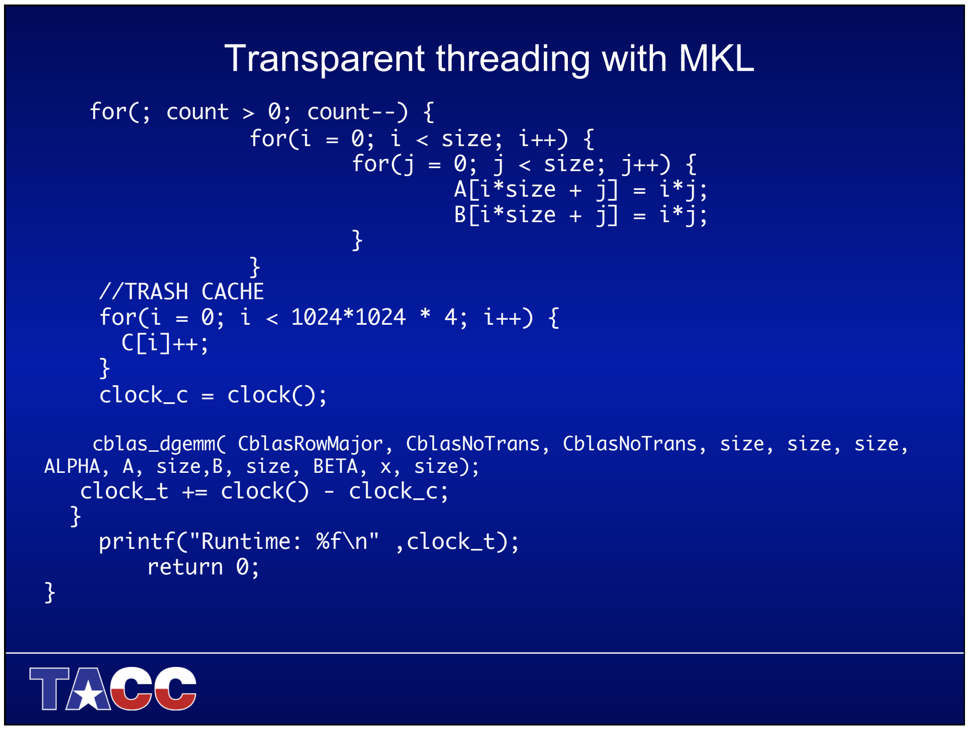
- ATLAS, or the Automatically Tuned Linear Algebra Subprograms, build, test, and re-build themselves at install time to match the particular behaviour of a processor (e.g. register size and cache tuning, SSE instructions, etc).
- Goto uses hand-tuned assembly to tweak performance; This guy (K. Goto) really lives for this stuff, and the performance gap over “naive code” is amazing.
- Intel’s Math Kernel Libraries (BLAS and other things) are tuned for each Intel microarchitecture for max performance.
- Unlike your program, these libraries can tell a Nocona from a Paxville from a Woodcrest (you’d call all of those “Xeon”).



Transparent threading with MKL

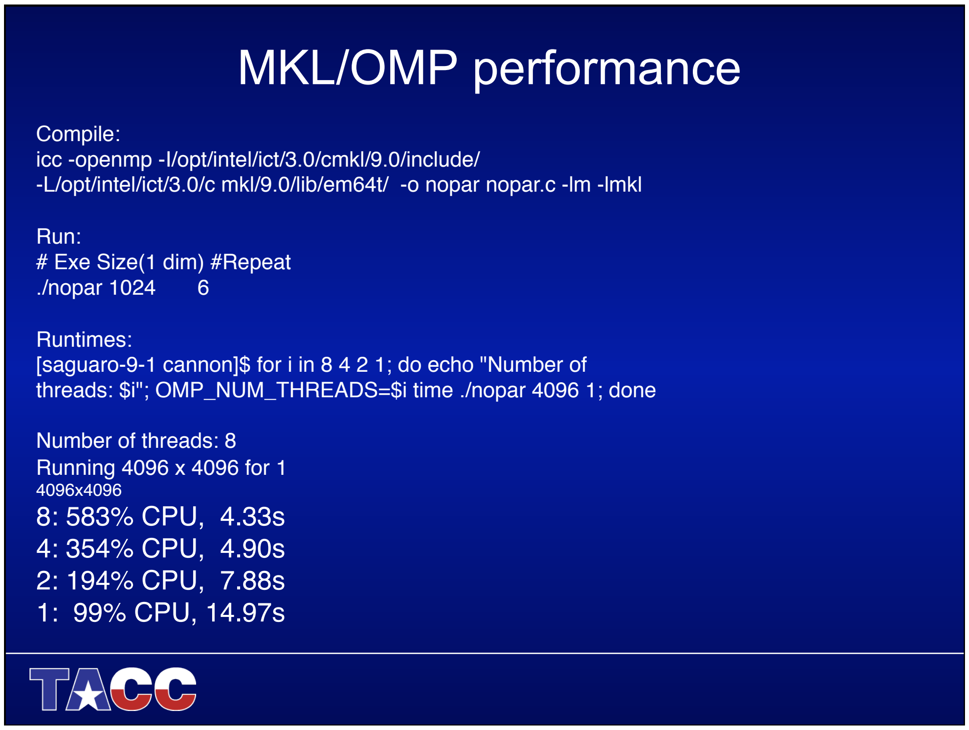
- OpenMP support is built into MKL
- Unmodified programs can use OMP inside the routines to get speedup based only on an environment variable.
- Of course, you must have idle processors available to make use of this...

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "mkl_cblas.h"

int main(int argc, char *argv[])
{
     int lda,ldb,ldc;
     double *A, *B, *x, *C;
     int size, i, j, count;
     double ALPHA, BETA;
   double clock_c, clock_t;
     ALPHA = 1.0;
     BETA = 1.0;
   clock_t = 0.0;
     if(argc < 3) { printf("No input specified\n"); return 1; }

     size = atoi(argv[1]);
     count = atoi(argv[2]);

     printf("Running %d x %d for %d\n", size, size, count);
     A = (double*)malloc(sizeof(double) * size * size);
     B = (double*)malloc(sizeof(double) * size * size);
   C = (double*)malloc(sizeof(double) * 1024 * 1024  * 4);
     x = (double*)malloc(sizeof(double) * size * size);
```



Transparent threading with MKL

```
     for(; count > 0; count--) {
                 for(i = 0; i < size; i++) {
                         for(j = 0; j < size; j++) {
                                 A[i*size + j] = i*j;
                                 B[i*size + j] = i*j;
                         }
                 }
      //TRASH CACHE
      for(i = 0; i < 1024*1024 * 4; i++) {
        C[i]++;
      }
      clock_c = clock();

     cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans, size, size, size,
ALPHA, A, size,B, size, BETA, x, size);
     clock_t += clock() - clock_c;
   }
      printf("Runtime: %f\n" ,clock_t);
          return 0;
}
```



MKL/OMP performance

Compile:

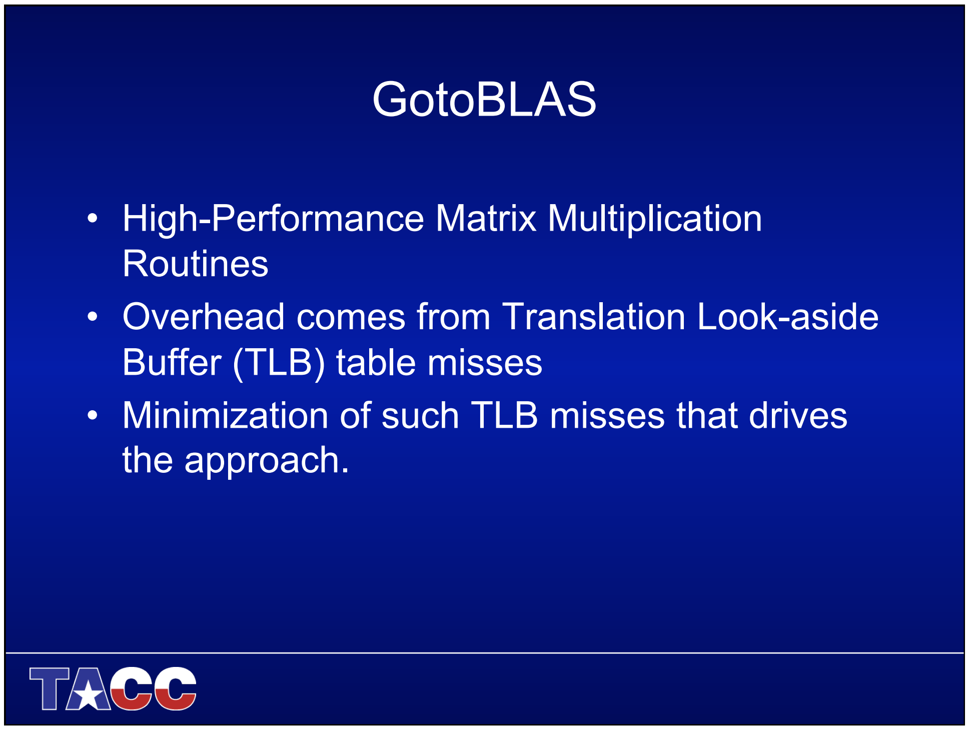
```
icc -openmp -I/opt/intel/ict/3.0/cmkl/9.0/include/
-L/opt/intel/ict/3.0/c mkl/9.0/lib/em64t/  -o nopar nopar.c -lm -lmkl
```

Run:

```
# Exe Size(1 dim) #Repeat
./nopar 1024      6
```

Runtimes:

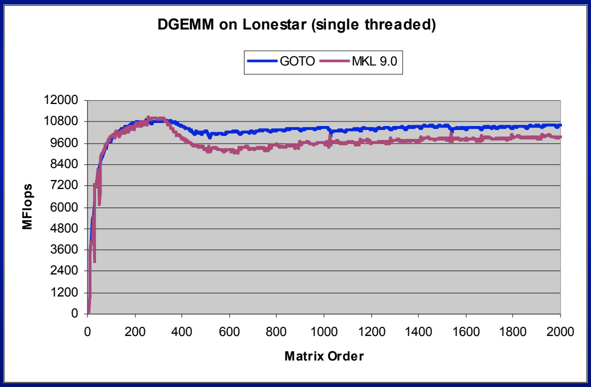
```
[saguaro-9-1 cannon]$ for i in 8 4 2 1; do echo "Number of
threads: $i"; OMP_NUM_THREADS=$i time ./nopar 4096 1; done

Number of threads: 8
Running 4096 x 4096 for 1
4096x4096
8: 583% CPU,  4.33s
4: 354% CPU,  4.90s
2: 194% CPU,  7.88s
1:  99% CPU, 14.97s
```



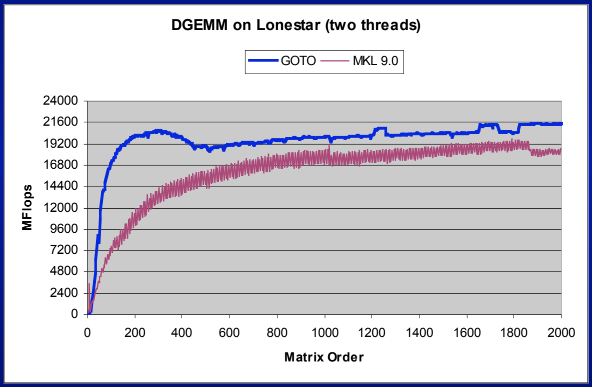
GotoBLAS

- High-Performance Matrix Multiplication Routines
- Overhead comes from Translation Look-aside Buffer (TLB) table misses
- Minimization of such TLB misses that drives the approach.

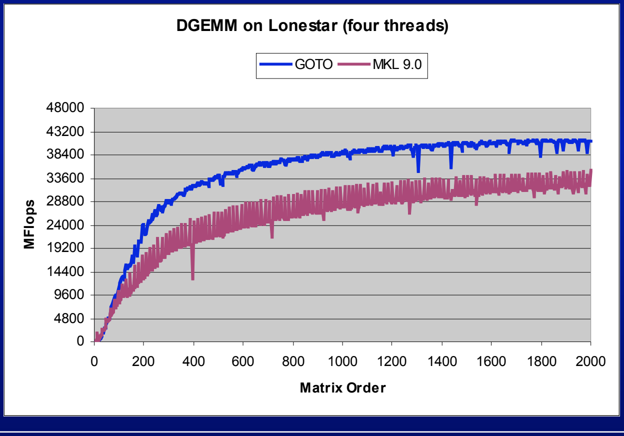
GotoBLAS EM64T Woodcrest Performance



GotoBLAS EM64T Woodcrest Performance



GotoBLAS EM64T Woodcrest Performance



TACC

Using the GotoBLAS Module

- Module load gotoblas

  mpicc test.c –L/$TACC_GOTOBLAS_LIB/
  libgotoblas64.a

- Supplementing MKL Libraries with GotoBLAS
  - Add gotoBLAS first, then MKL
    Module load gotoblas
    Module load mkl

```
mpicc -I$TACC_MKL_INC mkl_test.c \
      -L$TACC_GOTOBLAS_LIB/libgotoblas64.a  \
      -L$TACC_MKL_LIB \
      -lmkl_em64t –lmkl_lapack64
```

TACC

GotoBLAS References

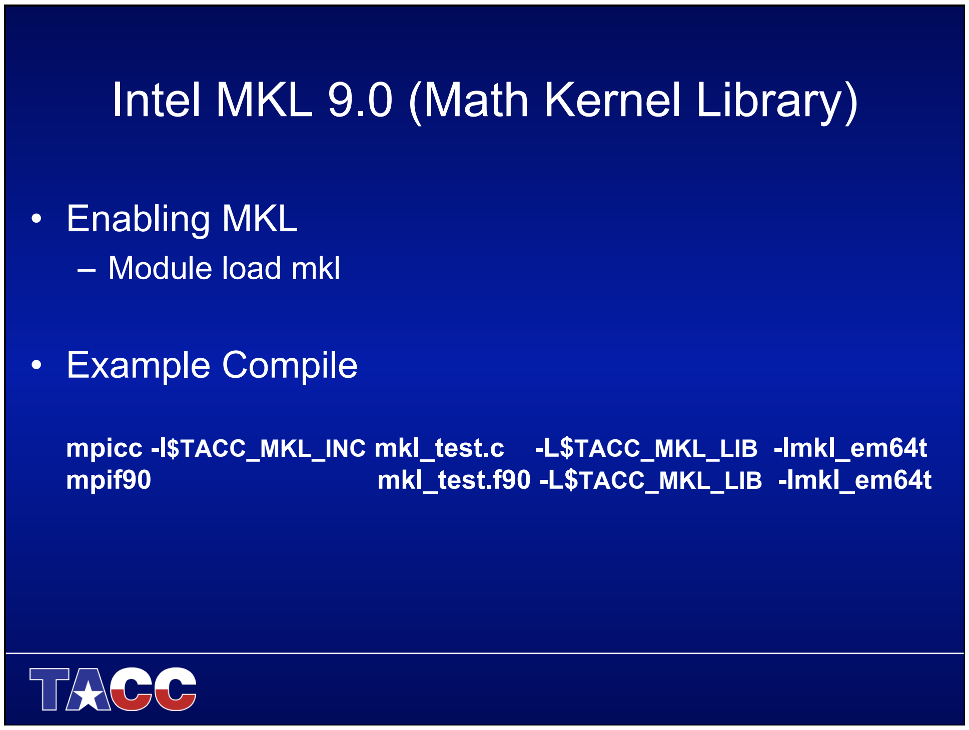
- Kazushige Goto
  - http://www.tacc.utexas.edu/general/staff/goto/
- GotoBLAS
  - http://www.tacc.utexas.edu/resources/software/

TACC

Intel MKL 9.0 (Math Kernel Library)

- Optimized for the IA32, EM64T, IA64 architectures
- supports both Fortran and C interfaces
- Includes functions in the following areas:
  - BLAS (levels 1-3)
  - LAPACK
  - FFT routines
  - … others
  - Vector Math Library (VML)

TACC



Intel MKL 9.0 (Math Kernel Library)

- Enabling MKL
  - Module load mkl
- Example Compile

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

TACC