Overcoming the Barriers to Sustained Petaflop Performance

William D. Gropp
Mathematics and Computer Science
www.mcs.anl.gov/~gropp
But First...

- Are we too CPU-centric?
- What about I/O?
  - What do applications need (not what are they doing)?
  - Will problems with scalable, parallel I/O be what keeps massively parallel machines from succeeding?

  *Are you sure? How much are you willing to bet? $100M? $200M?*
Where will we get (Sustained) Performance?

- Algorithms that are a better match for the architectures
- Parallelism at all levels
- Concurrency at all levels
- A major challenge is to realize these approaches in code
  - Most compilers do poorly with important kernels in computational science
  - Three examples - sparse matrix vector product, dense matrix-matrix multiply, flux calculation
Realistic Measures of Peak Performance
Sparse Matrix Vector Product
One vector, matrix size, $m = 90,708$, nonzero entries $nz = 5,047,120$

Thanks to Dinesh Kaushik; ORNL and ANL for compute time
Very Few Compilers do well on DGEMM (n=500)
Effect of code transformations for uniprocessor performance

Factor of 7
Performance for Real Applications

- Dense matrix-matrix example shows that even for well-studied, compute-bound kernels, compiler-generated code achieves only a small fraction of available performance
  - “Fortran” code uses “natural” loops, i.e., what a user would write for most code
  - Others use multi-level blocking, careful instruction scheduling etc.
- Algorithms design also needs to take into account the capabilities of the system, not just the hardware
- Adding concurrency (whether multicore or multiple processors) just adds to the problems
Possible solutions

- Single, integrated system
  - Best choice when it works
    - Matlab

- Current Terascale systems and many proposed petascale systems exploit hierarchy
  - Successful at many levels
    - Cluster hardware
    - OS scalability
  - We should apply this to productivity software
    - The problem is hard
    - Apply classic and very successful Computer Science strategies to address the complexity of generating fast code for a wide range of user-defined data structures.

- How can we apply hierarchies?
  - One approach is to use libraries
    - Limited by the operations envisioned by the library designer
  - Another is to enhance the users ability to express the problem in source code
Annotations

- Aid in the introduction of hierarchy into the software
  - It's going to happen anyway, so make a virtue of it
- Create a “market” or ecosystem in transformation tools
- Longer term issues
  - Integrate annotation language into “host” language to ensure type safety, ensure consistency (both syntactic and semantic), closer debugger integration, additional optimization opportunities through information sharing, …
Examples of the Challenges

- Fast code for DGEMM (dense matrix-matrix multiply)
  - Code generated by ATLAS omitted to avoid blindness 😊
  - Example code from “Superscalar GEMM-based Level 3 BLAS”, Gustavson et al on the next slide

- PETSc code for sparse matrix operations
  - Includes unrolling and use of registers
  - Code for diagonal format is fast on cache-based systems but slow on vector systems.
    - Too much code to rewrite by hand for new architectures

- MPI implementation of collective communication and computation
  - Complex algorithms for such simple operations as broadcast and reduce are far beyond a compiler’s ability to create from simple code
SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )

USEC = ISSEC-MOD( ISSEC, 4 )
DO 390 J = JJ, JJ+UJSEC-1, 4
   DO 360 I = II, II+UISEC-1, 4
      F11 = DELTA*C( I,J )
      F21 = DELTA*C( I+1,J )
      F12 = DELTA*C( I,J+1 )
      F22 = DELTA*C( I+1,J+1 )
      F13 = DELTA*C( I,J+2 )
      F23 = DELTA*C( I+1,J+2 )
      F14 = DELTA*C( I,J+3 )
      F24 = DELTA*C( I+1,J+3 )
      F31 = DELTA*C( I+2,J )
      F41 = DELTA*C( I+2,J+1 )
      F32 = DELTA*C( I+2,J+2 )
      F42 = DELTA*C( I+2,J+3 )
      F33 = DELTA*C( I+2,J+3 )
      F43 = DELTA*C( I+2,J+4 )
      F44 = DELTA*C( I+2,J+4 )
   DO 350 L = LL, LL+LSEC-1
      F11 = F11 + T1( L-LL+1, I-II+1 )*
            T2( L-LL+1, J-JJ+1 )
      F21 = F21 + T1( L-LL+1, I-II+2 )*
            T2( L-LL+1, J-JJ+2 )
      F12 = F12 + T1( L-LL+1, I-II+1 )*
            T2( L-LL+1, J-JJ+3 )
      F22 = F22 + T1( L-LL+1, I-II+2 )*
            T2( L-LL+1, J-JJ+3 )
      F13 = F13 + T1( L-LL+1, I-II+3 )*
            T2( L-LL+1, J-JJ+4 )
      F23 = F23 + T1( L-LL+1, I-II+3 )*
            T2( L-LL+1, J-JJ+4 )
      F14 = F14 + T1( L-LL+1, I-II+4 )*
            T2( L-LL+1, J-JJ+4 )
      F24 = F24 + T1( L-LL+1, I-II+4 )*
            T2( L-LL+1, J-JJ+4 )
   CONTINUE
   F11 = F11 + T1( I-II+1, J-JJ+1 )
   F21 = F21 + T1( I-II+2, J-JJ+2 )
   F12 = F12 + T1( I-II+1, J-JJ+3 )
   F22 = F22 + T1( I-II+2, J-JJ+3 )
   F13 = F13 + T1( I-II+3, J-JJ+4 )
   F23 = F23 + T1( I-II+3, J-JJ+4 )
   F14 = F14 + T1( I-II+4, J-JJ+4 )
   F24 = F24 + T1( I-II+4, J-JJ+4 )
END

Why not just

   do i=1,n
      do j=1,m
         c(i,j) = 0
      do k=1,p
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
   enddo
enddo

Note: This is just part of DGEMM!
Performance of Matrix-Matrix Multiplication
(MFlops/s vs. n2; n1 = n2; n3 = n2*n2)
Intel Xeon 2.4 GHz, 512 KB L2 Cache, Intel Compilers at –O3 (Version 8.1),
February 12, 2006
Observations

- Much use of mechanical transformations of code to achieve better performance
  - Compilers do not do this well
    - *Too many other demands on the compiler*
- Use of carefully crafted algorithms for specific operations such as allreduce, matrix-matrix multiply
  - Far more challenging than the performance transformations
- Increasing acceptance of some degree of automation in creating code
  - ATLAS, PhiPAC, TCE
  - Source-to-source transformation systems
    - *E.g., ROSE, Aspect Oriented Programming ([asod.net](http://asod.net))*
Key Observations

- **90/10 rule**
  - current languages adequate for 90% of code
  - 10% of code causes 90% of trouble

- **Memory hierarchy issues a major source of problems**
  - Significant effort is put into relatively mechanical transformations of code
  - Other transformations are avoided because of their negative impact on the readability and maintainability of the code.
    - *Example is loop fusion for routines that sweep over a mesh to apply different physics. Fusion, needed to reduce memory bandwidth requirements, breaks modularity of routines written by different groups.*

- **Coordination of distributed data structures another major source of problems**
  - But the need for performance encourages a global/local separation
    - *Reflected in PGAS languages*

- **New languages may help, but not anytime soon**
  - New languages will never be the entire solution
  - Applications need help now
One Possible Approach

- Use annotations to augment existing languages
  - Not a new approach; used in HPF, OpenMP, others
  - Some applications already use this approach for performance portability
    - *WRF weather code*

- Annotations do have limitations
  - Fits best when most of the code is independent of the parts affected by the annotations
  - Limits optimizations that are available to approaches that augment the language (e.g., telescoping languages)

- But they also have many advantages…
Annotations example: STREAM triad.c for BG/L

```c
void triad(double *a, double *b, double *c, int n){
    int i;
    double ss = 1.2;
    /* --Align;;var:a,b,c;; */
    if ( ((int)(a) | (int)(b) | (int)(c)) & 0xf == 0) {
        __alignx(16,a);
        __alignx(16,b);
        __alignx(16,c);
        for (i=0;i<n;i++) {
            a[i] = b[i] + ss*c[i];
        }
    } else {
        for (i=0;i<n;i++) {
            a[i] = b[i] + ss*c[i];
        }
    }
    /* --end Align */
}
```
### Simple annotation example: STREAM triad.c on BG/L

<table>
<thead>
<tr>
<th>Size</th>
<th>No Annotations (MB/s)</th>
<th>Annotations (MB/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1920.00</td>
<td>2424.24</td>
<td>2.5X</td>
</tr>
<tr>
<td>100</td>
<td>3037.97</td>
<td>6299.21</td>
<td>2.9X</td>
</tr>
<tr>
<td>1000</td>
<td>3341.22</td>
<td>8275.86</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>1290.81</td>
<td>3717.88</td>
<td></td>
</tr>
<tr>
<td>50000</td>
<td>1291.52</td>
<td>3725.48</td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>1291.77</td>
<td>3727.21</td>
<td>2.9X</td>
</tr>
<tr>
<td>500000</td>
<td>1291.81</td>
<td>1830.89</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>1282.12</td>
<td>1442.17</td>
<td></td>
</tr>
<tr>
<td>2000000</td>
<td>1282.92</td>
<td>1415.52</td>
<td></td>
</tr>
<tr>
<td>5000000</td>
<td>1290.81</td>
<td>1446.48</td>
<td>1.12X</td>
</tr>
</tbody>
</table>
Summary

- Provide tools to help computational scientists build transportable, high-performance applications by working with, not against the compiler.
- Enable an ecosystem so that tools can compete
  - Enables and rewards research and development
- Lowers the barrier to introducing more complex data structures and algorithms

- And don’t forget the I/O!