A User’s View of OpenMP: The Good, The Bad, and The Ugly

William D. Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
http://www.mcs.anl.gov/~gropp
Collaborators

- Dinesh K. Kaushik
  MCS Division, Argonne National Laboratory & CS Department, Old Dominion University

- David E. Keyes
  Math. & Stat. Department, Old Dominion University & ISCR, Lawrence Livermore National Laboratory

- Barry F. Smith
  MCS Division, Argonne National Laboratory
But First: MPI Fact and Fiction

- MPI requires buffering
  - False. MPI was specifically designed to avoid buffering
  - A few implementations need work (sometime in the OS)
- MPI requires $n^2$ buffers for $n$ processes
  - False, but most implementations need work
- MPI defined in the 80’s
  - MPI Forum’s first meeting was in January 1993
- MPI was derived from PVM
  - MPI emerged from a broad consensus of message-passing vendors, researchers, and users.
- MPI thread safety
  - MPI (the standard) was designed to *allow* thread-safe implementations but not require them (performance tradeoffs)
  - `MPI_Init_thread` (MPI-2) allows an application to request and discover the level of thread safety (4 levels defined)
Outline

- The Good
  - Successful use of incremental parallelism
  - (Relatively) easy realization of better algorithms
- The Not so Good
  - Limitations in OpenMP impacted code
  - OpenMP version 2 fixes some (Thanks!)
- The Bad
  - Lack of effective support for modularity and libraries
  - Incorrect programs (that run) are too easy to write
- The Ugly
  - Implementation Issues
  - Mixed C and Fortran applications
What We’ve Done

- Fun3d-PETSc (1999 Gordon Bell winner)
- Tetrahedral vertex-centered unstructured grid code developed by W. K. Anderson (NASA LaRC) for steady compressible and incompressible Euler and Navier-Stokes equations (with one-equation turbulence modeling)
- Used in airplane, automobile, and submarine applications for analysis and design
- Standard discretization is 2nd-order Roe for convection and Galerkin for diffusion
- Original code used Newton-Krylov solver with global point-block-ILU preconditioning
- Parallel version uses Newton-Krylov-Schwarz, with domain-induced point-block ILU preconditioning
Fun3d Performance

Used mixed MPI/SMP model

Performance close to “achievable peak” based on memory bandwidth
Primary PDE Solution Kernels

- Vertex-based loops
  - State vector and auxiliary vector updates
- Edge-based "stencil op" loops
  - Residual evaluation
  - Approximate Jacobian evaluation
  - Jacobian-vector product (often replaced with matrix-free form, involving residual evaluation)
- Sparse, narrow-band recurrences
  - Approximate factorization and back substitution
- Vector inner products and norms
  - Orthogonalization/conjugation
  - Convergence progress and stability checks
- **Preconditioned linear (and nonlinear) solution**
Multi-level Numerical Methods

- Domain Decomposition Preconditioner
  - Efficient method *independent* of parallelism
  - Multilevel method is a good match to multilevel memory hierarchy without sacrificing convergence rate

Leads to an efficient algorithm for solving nonlinear PDEs:
Time-Implicit Newton-Krylov-Schwarz Method

for (l = 0; l < n_time; l++) {
    select time step
    for (k = 0; k < n_Newton; k++) {
        compute nonlinear residual and Jacobian
        for (j = 0; j < n_Krylov; j++) {
            forall (i = 0; i < n_Precon; i++) {
                solve subdomain problems concurrently
            } // End of loop over subdomains
        } // End of linear solver
        perform Jacobian-vector product
        enforce Krylov basis conditions
        update optimal coefficients
        check linear convergence
    } // End of nonlinear loop
} // End of time-step loop

This is implemented in a parallel library…
Separation of Concerns: User Code/PETSc Library

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

Application Initialization

Function Evaluation

Jacobian Evaluation

Post-Processing

PETSc

User code

PETSc code
Background of PETSc

- Developed by Gropp, Smith, McInnes & Balay (ANL) to support research, prototyping, and production parallel solutions of operator equations in message-passing environments
- Distributed data structures as fundamental objects—index sets, vectors/gridfunctions, and matrices/arrays
- Iterative linear and nonlinear solvers, combinable modularly and recursively, and extensibly
- Portable, and callable from C, C++, Fortran
- Uniform high-level API, with multi-layered entry
- Aggressively optimized: copies minimized, communication aggregated and overlapped, caches and registers reused, memory chunks preallocated, inspector-executor model for repetitive tasks (e.g., gather/scatter)
- Supports a wide variety of sparse matrix formats, including user-defined.
- Extensible with user-defined preconditioners, iterative methods, etc.
Parallel Fun3d

- Uses PETSc for parallelism
  - Almost no MPI in Fun3d itself
  - MPI used only for initialization from data files
- OpenMP
  - Used only for flux evaluation
- Where did programmer time go?
  - Uniprocessor performance tuning
  - Primarily \textit{locality} management
- (Parallel programming is easy compared to performance programming)
- Why was OpenMP only used for the flux evaluation?
Competing for the Available Memory Bandwidth

- The processors on a node compete for the available memory bandwidth.
- The computational phases that are memory-bandwidth limited will not scale.
  - They may even run slower because of the extra synchronizations.
Stream Benchmark on ASCI Red MB/s for the Triad Operation

<table>
<thead>
<tr>
<th>Vector Size</th>
<th>1 Thread</th>
<th>2 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^4$</td>
<td>666</td>
<td>1296</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>137</td>
<td>238</td>
</tr>
<tr>
<td>$10^5$</td>
<td>140</td>
<td>144</td>
</tr>
<tr>
<td>$10^6$</td>
<td>145</td>
<td>141</td>
</tr>
<tr>
<td>$10^7$</td>
<td>157</td>
<td>152</td>
</tr>
</tbody>
</table>
Redundant Storage and Work

- To manage memory updates efficiently, we might need to create extra private work arrays.
- These work arrays need to be copied into a shared array at the end of the parallel region.
  - A memory-bandwidth limited sequential phase
- The vector reduction in OpenMP v.2 may help.
Flux Evaluation in PETSc-FUN3D

- **Variables at each node:**
  - density,
  - momentum \((x, y, z)\),
  - energy,
  - pressure

- **Variables at edge:**
  - identity of nodes,
  - orientation \((x, y, z)\)

- read variables
- compute
- update variables
Apply the “Owner Computes” Rule for OpenMP

- Create the disjoint working sets to eliminate the redundant private arrays (e.g. by coloring the edges and nodes)
- Alternatively, use OpenMP over subdomains
  - each MPI process will repartition its domain
  - each thread will work on its assigned subdomain
- Brings in the complexity of programming as the user is taking care of the memory updates
MPI/OpenMP in PETSc-FUN3D

- Only in the flux evaluation phase as it is not memory-bandwidth bound
- Gives the best execution time as the number of nodes increases because the subdomains are chunkier as compared to pure MPI case

<table>
<thead>
<tr>
<th>Nodes</th>
<th>MPI/OpenMP</th>
<th></th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 Thr</td>
<td>2 Thr</td>
<td>1 Proc</td>
</tr>
<tr>
<td>256</td>
<td>483s</td>
<td>261s</td>
<td>456s</td>
</tr>
<tr>
<td>2560</td>
<td>76s</td>
<td>39s</td>
<td>72s</td>
</tr>
<tr>
<td>3072</td>
<td>66s</td>
<td>33s</td>
<td>62s</td>
</tr>
</tbody>
</table>
For the Fun3d Application:

- The 1-thread/process case shows loop overhead costs in OpenMP implementation.
- OpenMP allows the easy implementation of a better algorithm.
- Vector reduction should improve the OpenMP advantage.
The Good

• Effective Incremental Parallelism
  ♦ Important contributor to ASCI Red results (not exactly OpenMP, but same philosophy)

• Good SMP and SMP-cluster match
  ♦ Larger domain decomposition blocks
  ♦ Encourages more dynamic code
The Not so Good

- **Performance**
  - In apples-to-apples comparison with MPI
  - Data placement important
  - Cache blocking etc. mismatch with OpenMP loop scheduling

- **Restrictions on atomic update/reduce**
  - No vector reduce (p 29) (but see OpenMP 2.0)
  - Complexity *for user* comes from exceptions and limitations
The Bad

- Program correctness
  - It is too easy to write incorrect programs
- Software Modularity
  - At best 2-level modularity
  - Many modern algorithms built out of components; how will OpenMP support them?
  - E.g., each component uses limited parallelism to fit problem into local caches; application uses task parallelism to perform intelligent (not exhaustive parameter-space search) design optimization.
Program Correctness

• It is much too easy to write incorrect programs
• Updates to variables
  ♦ Should be atomic unless specifically requested (see p 21)
  ♦ Principle: user omission of a directive shouldn’t create incorrect code
  ♦ Current model is like Fortran implicit typing—convenient if you never make a mistake
• Volatile?
  ♦ Even better, shared non-volatile (read-only shared)
• Consistency model
  ♦ What is the model?
    • Not sequential consistency (see atomic, flush)
• Example: Using flags instead of locks
  ♦ Requires FLUSH (maybe not so bad, but the documentation is not sufficient for users to understand the need for this operation)
Software Modularity

- Libraries must either
  - Use OpenMP at “leaves” (e.g., the loop-level), or
  - Take complete control (user program has no OpenMP parallelism when library is called).
  - But some libraries call other library routines ...
    - E.g., should BLAS use OpenMP? LAPACK? What if user uses OpenMP for task parallelism for a routine that calls an LAPACK routine?

- Using OpenMP at loop-level incurs startup costs
  - Some vendors suggest
    - Program Main
      !omp parallel
      ...
      !omp end parallel
      stop
      end

- OpenMP language bindings poorly chosen for mixed-language programming
  - I.e., programs that use libraries ...
Language Bindings for Mixed Language Programming

- Libraries used by Fortran may be written in C (and vice versa)
  - OpenMP naming convention can make this (nearly) impossible
- C names should *always* be distinguishable from Fortran names
  - Unless bindings are *identical*
  - Using mixed case for C (as in MPI) is an easy way to do this
- Consider (from SGI)
  - `f77 -noappend -c -mp s1.f`
  - `cc -mp -o s2 s2.c s1.o`
Simple Mixed-Language Program

- subroutine setnthreads(req)
  integer req
  call omp_set_num_threads(req)
  end

- #include <stdio.h>
  #include <omp.h>
  int main(int argc, char *argv[])
  {
    int n_c, n_f, req=4;
    setnthreads(&req);
    #pragma omp parallel
    {
      n_f = omp_get_num_threads();
    }
    omp_set_num_threads(req);
    #pragma omp parallel
    {
      n_c = omp_get_num_threads();
    }
    printf( “n_c=%d n_f=%d
”, n_c, n_f );
  }

What is printed out?

n_c=4 n_f=8

8 is the default maximum number of threads
Performance

• Data distribution matters for performance
  ♦ There are no UMA machines
    • (cache, vector registers, even if all main memory is uniformly far away)
• C mallocs (all shared; scalability?)
  ♦ Task parallel applications; data is primarily private
  ♦ Ok for SMP platforms, but what about DSM?
• No way to get the compiler to compute good dynamic blocking (default chunk = 1)
  ♦ OpenMP directives tell the compiler to do something specific
  ♦ Does not match user model
    • E.g., -O often includes “unroll by a good amount”
    • Does not mean that user-control is not valuable, just that some decisions are system dependent
The Ugly
(E.g. Implementation Problems)

- David Bailey’s rule #8 (roughly)
  - Base the operation count on the parallel implementation, not the best sequential implementation
- Early tests with Fun3D showed base OpenMP case (1 thread/process) took longer than reference MPI case.
- Consider the performance of the `jacobi.f` example from [www.openmp.org](http://www.openmp.org) :
Scaling of an OpenMP Example

![Graph showing scaling of an OpenMP example with speedup and real speedup curves.](image-url)
Data Placement

• Performance often depends on managing memory motion
• First touch is inadequate
  ♦ Requires code just for OpenMP version
  ♦ Conflicts with incremental parallelism
    • Requires parallelization of initialization
  ♦ Conflicts with libraries that may share data
• Dependent on page/cache line size
  ♦ Architecture-dependent information
  ♦ The compiler (often) has this information: let OpenMP use it
Conclusions

- OpenMP provides good support for incremental parallelism; integrates well with other tools.
- Needs attention to:
  - Modularity
    - Good support for single-level and two-level codes
    - Thread groups or something else needed for libraries
  - Software engineering
    - Incorrect programs are too easy to write
    - Mixed-language programming needs to be fixed
  - Performance
    - Data motion expensive
- Backward-compatible improvements can be made.