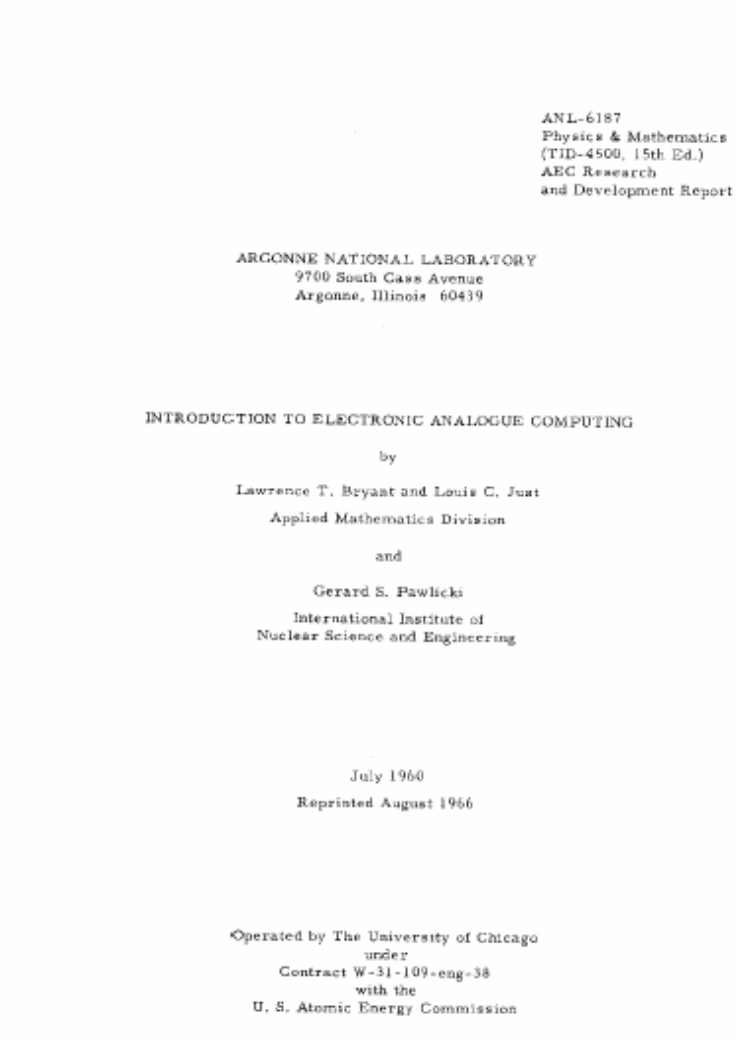
Software for Exaflops Computing

William Gropp
Mathematics and Computer Science Division
www.mcs.anl.gov/~gropp

Why New Programming Approaches?

- Massive Parallelism
- Different Hardware
- Frequent Faults
- Higher Productivity
  - Whorfian hypothesis
    - Strong form: Language controls both thought and behavior
      - Rejected by linguistics community

We Haven't Always Been Digital



ANL-6187
Physics & Mathematics
(TID-4500, 15th Ed.)
AEC Research
and Development Report

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

INTRODUCTION TO ELECTRONIC ANALOGUE COMPUTING

by

Lawrence T. Bryant and Louis C. Just

Applied Mathematics Division

and

Gerard S. Pawlicki

International Institute of
Nuclear Science and Engineering

July 1960

Reprinted August 1966

Operated by The University of Chicago
under
Contract W-31-109-eng-38
with the
U. S. Atomic Energy Commission

Historical Context

- Pasadena workshop (1992)
- PetaFlops workshops (1994—)
- Gloom and doom
- Success!
  - But we won't admit it

Quotes from "System Software and Tools for High Performance Computing Environments"

- "The strongest desire expressed by these users was simply to satisfy the urgent need to get applications codes running on parallel machines as quickly as possible"
- In a list of enabling technologies for mathematical software, "Parallel prefix for arbitrary user-defined associative operations should be supported. Conflicts between system and library (e.g., in message types) should be automatically avoided."
  - Note that MPI-1 provided both
- "For many reasons recoverability mechanisms are important for both batch and interactive systems."
  - Followed by a discussion of checkpointing
- Immediate Goals for Computing Environments:
  - Parallel computer support environment
  - Standards for same
  - Standard for parallel I/O
  - Standard for message passing on distributed memory machines
- "The single greatest hindrance to significant penetration of MPP technology in scientific computing is the absence of common programming interfaces across various parallel computing systems"

Quotes from "Enabling Technologies for Petaflops Computing":

- "The software for the current generation of 100 GF machines is not adequate to be scaled to a TF…"
- "The Petaflops computer is achievable at reasonable cost with technology available in about 20 years [2014]."
  - (estimated clock speed in 2004 — 700MHz
- "Software technology for MPP's must evolve new ways to design software that is portable across a wide variety of computer architectures. Only then can the small but important MPP sector of the computer hardware market leverage the massive investment that is being applied to commercial software for the business and commodity computer market."
- "To address the inadequate state of software productivity, there is a need to develop language systems able to integrate software components that use different paradigms and language dialects."
- (9 overlapping programming models, including shared memory, message passing, data parallel, distributed shared memory, functional programming, O-O programming, and evolution of existing languages)

Is There A Problem?

- Many feel that programming for performance is too hard; there is a *productivity crisis*
- And supporting new algorithms is too difficult
  - Either use new algorithm on slow hardware (general CISC/RISC µprocessor)
  - Or old algorithm on fast hardware (vector/stream processor)
- But despite the gloom and doom, and despite little organized effort to solve all of these problems, we have *applications* running at over 10 TF today.

Contrarian View

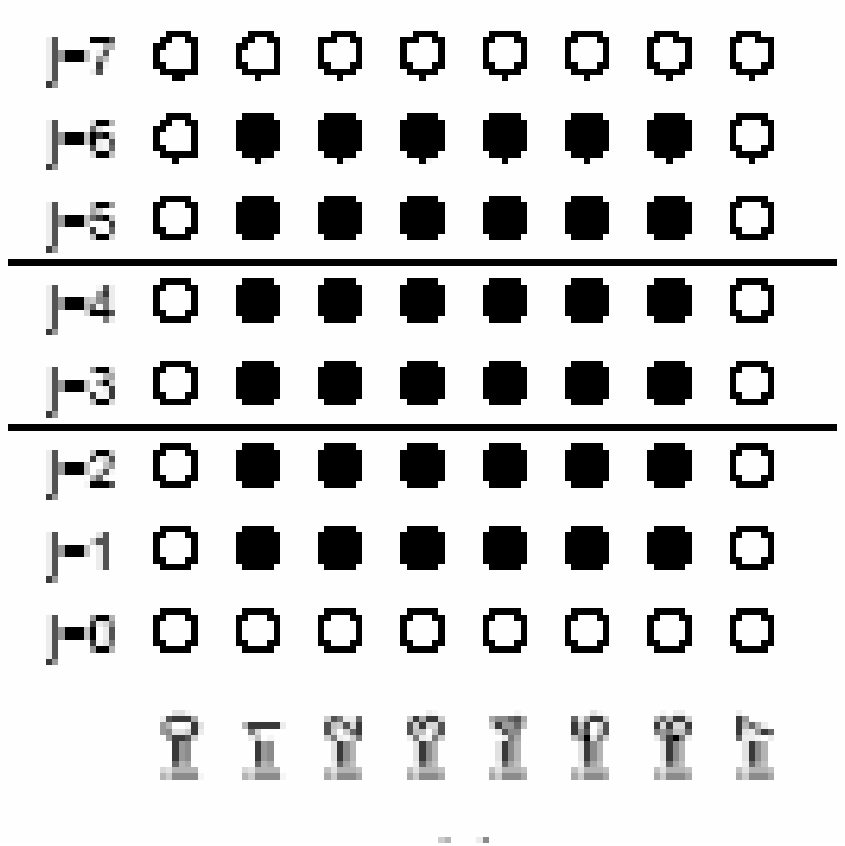
- Algorithms are an expression of the *mathematics*
  - Need new *algorithms*
  - Need better ways to express those algorithms that match hardware realities
    - Parallelism is only one of the easier problems
    - Algorithms must match what the hardware can do well — this is where languages may need to change (Whorf)
- Are new languages really necessary?
  - If so, how should they be evaluated?
    - The must address the hard problems, not just the easy ones
  - If not, how do we solve the problems we face?
- To see the pros and cons of new languages, lets look at some examples…

Consider These Five Examples

- Three Mesh Problems
  - Regular mesh
  - Irregular mesh
  - C-mesh
- Indirect access
- Broadcast to all processes and allreduce among all processes

Regular Mesh Codes

- Classic example of what is wrong with MPI
  - Some examples follow, taken from *CRPC Parallel Computing Handbook* and ZPL web site, of mesh sweeps



Uniprocessor Sweep

```
do k=1, maxiter
   do j=1, n-1
      do i=1, n-1
         unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                              u(i,j+1) + u(i,j-1) - &
                              h * h * f(i,j) )
      enddo
   enddo
   u = unew
enddo
```

MPI Sweep

```
do k=1, maxiter
    ! Send down, recv up
    call MPI_Sendrecv( u(1,js), n-1, MPI_REAL, nbr_down, k &
           u(1,je+1), n-1, MPI_REAL, nbr_up, k, &
           MPI_COMM_WORLD, status, ierr )
    ! Send up, recv down
    call MPI_Sendrecv( u(1,je), n-1, MPI_REAL, nbr_up, k+1, &
           u(1,js-1), n-1, MPI_REAL, nbr_down, k+1,&
           MPI_COMM_WORLD, status, ierr )
    do j=js, je
       do i=1, n-1
          unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) - &
                      h * h * f(i,j) )
       enddo
     enddo
    u = unew
enddo
```

And the more scalable 2-d decomposition is even messier

HPF Sweep

```
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew WITH u
!HPF$ ALIGN f WITH u
do k=1, maxiter
   unew(1:n-1,1:n-1) = 0.25 * &
        ( u(2:n,1:n-1) + u(0:n-2,1:n-1) + &
        u(1:n-1,2:n) + u(1:n-1,0:n-2) - &
        h * h * f(1:n-1,1:n-1) )
   u = unew
enddo
```

OpenMP Sweep

```
!$omp parallel
!$omp do
   do j=1, n-1
        do i=1, n-1
            unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                        u(i,j+1) + u(i,j-1) - &
                        h * h * f(i,j) )
        enddo
   enddo
!$omp enddo
```

ZPL Sweep

```
region
    R = [ 0..n+1,0..n+1];
direction
    N=[-1,0]; S = [1,0]; W=[0,-1];  E=[0,1];
Var
    u : [BigR] real;
[R] repeat
    u:=0.25*(u@n + u@e + u@s + u@w)-h*h*f;
Until (…convergence…);
```

(Roughly, since I'm not a ZPL programmer)

Other Solutions

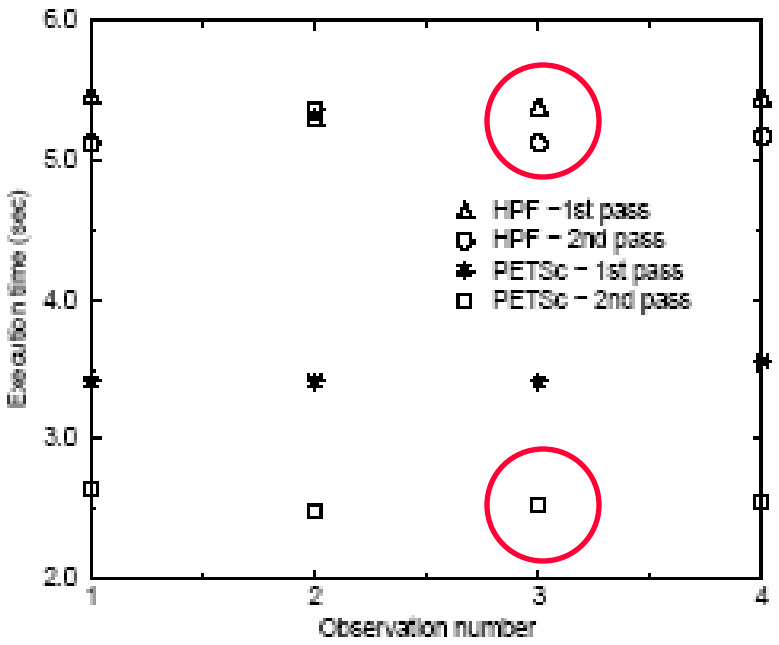
- Similarly nice code for this example can be prepared in other "global name space" languages, such as UPC and CAF (CoArray Fortran)
  - User is responsible for more details than in the examples shown, but code is still simpler than MPI code

Lessons

- Strengths of non-MPI solutions
  - Data decomposition done for the programmer
  - No “action at a distance”
- So why does anyone use MPI?
  - Performance
  - Completeness
  - Ubiquity
    - Does your laptop have MPI on it? Why not?
- But more than that…

Why Not Always Use HPF?

- Performance!
  - From "A Comparison of PETSC Library and HPF Implementations of an Archetypal PDE Computation" (*M. Ehtesham Hayder, David E. Keyes, and Piyush Mehrotra)*
  - PETSc (Library using MPI) performance *double* HPF
- Maybe there's something to explicit management of the data decomposition…



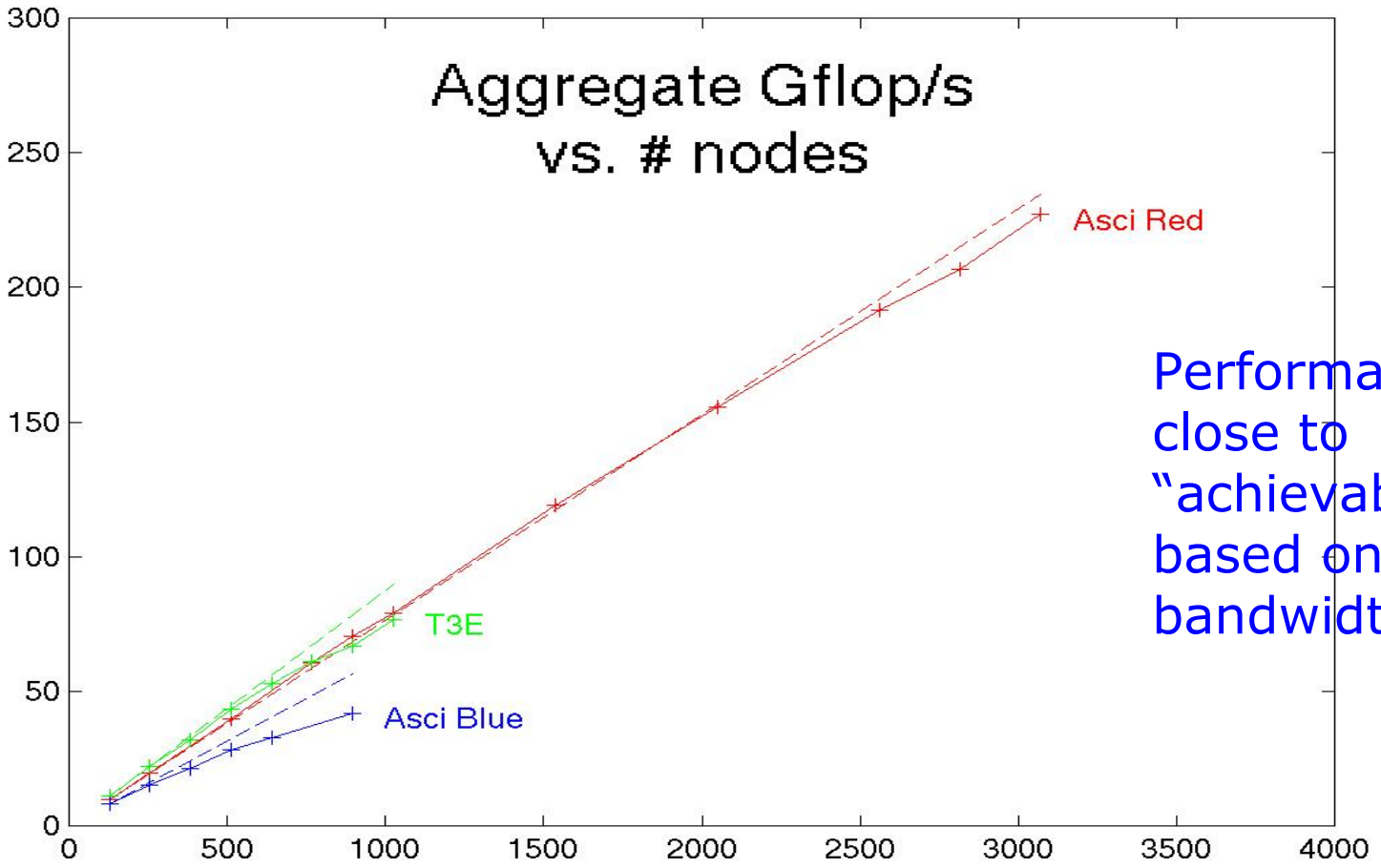
Not All Codes Are Completely Regular

- Examples:
  - Adaptive Mesh refinement
    - How does one process know what data to access on another process?
      - Particularly as mesh points are dynamically allocated
    - (You could argue for fine-grain shared/distributed memory, but performance cost is an unsolved problem in general)
    - Libraries exist (in MPI), e.g., Chombo, KeLP (and successors)
  - Unstructured mesh codes
    - More challenging to write in any language
    - Support for abstractions like index sets can help, but only a little
    - MPI codes are successful here …

FUN3d Characteristics

- 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)
- Won Gordon Bell Prize in 1999
- Uses MPI for parallelism
- Application contains ***ZERO*** explicit lines of MPI
  - All MPI within the PETSc library

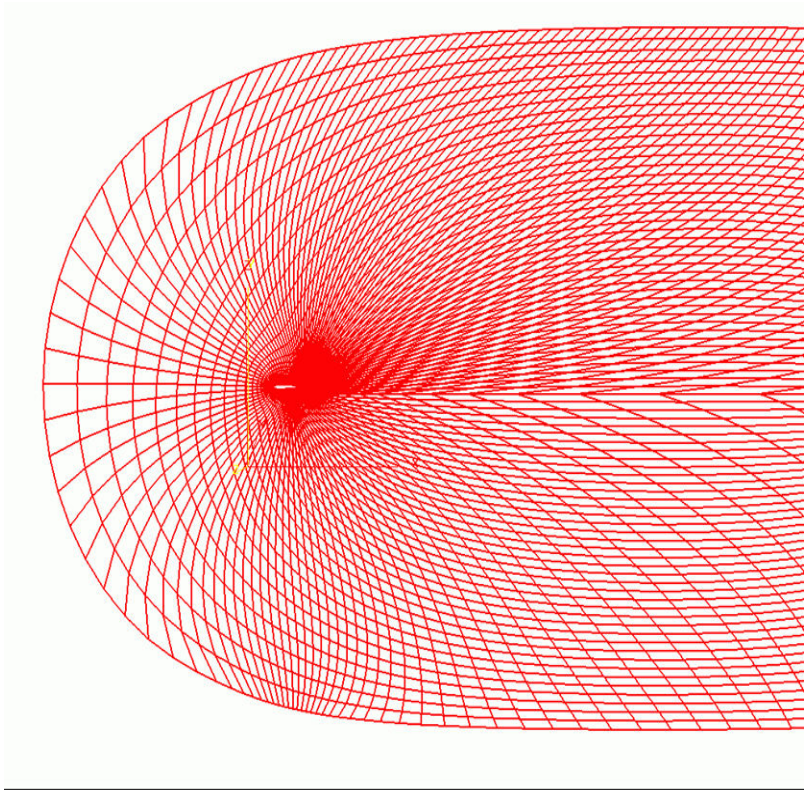
Fun3d Performance



Performance close to "achievable peak" based on memory bandwidth

Another Example: Regular Grids—But With a Twist

- "C Grids" common for certain geometries
- Communication pattern is regular but not part of "mesh" or "matrix" oriented languages
  - |i-n/2|>L, use one rule, otherwise, use a different rule
  - No longer transparent in HPF or ZPL
  - Convenience features are *brittle*
    - Great when they match what you want
    - But frustrating when they don't
  - (I haven't even started on staggered meshes or mortar element methods or 1-irregular grids or LUMR …)



Irregular Access

- For j=1, zillion
  table[f(j)] ^= intable[f(j)]
- Table, intable are "global" arrays (distributed across all processes)
- Seems simple enough
  - ^ is XOR, which is associative and commutative, so order of evaluation is irrelevant
- Core of the GUPS (also called TableToy) example
  - Two versions: MPI and shared memory
  - MPI code is much more complicated

But…

- MPI version produces the same answer every time
- Shared/Distributed memory version *does not*
  - Race conditions are present
  - Benchmark is from a problem domain where getting the same answer every time is not required
  - Scientific simulation often does not have this luxury
- You *can* make the shared memory version produce the same answer every time, but
  - You either need fine-grain locking
    - In software, costly in time, may reduce effective parallelism
    - In hardware, with sophisticated remote atomic operations (such as a remote compare and swap), but costly in $
  - Or you can aggregate operations
    - Code starts looking like MPI version …

Broadcast And Allreduce

- Simple in MPI:
  - MPI_Bcast, MPI_Allreduce
- Simple in shared memory (?)
  - do i=1,n
  a(i) = b(i) ! B (shared) broadcast to A
enddo
  - do i=1,n
  sum = sum + A(i) ! A (shared) reduced to sum
enddo
- But wait — how well would those perform?
  - Poorly. Very Poorly (much published work in shared-memory literature)
  - Optimizing these operations is not easy (e.g., see papers at EuroPVMMPI03-04)
- Unrealistic to expect a compiler to come up with these algorithms
  - E.g., OpenMP admits this and contains a special operation for scalar reductions (OpenMP v2 adds vector reductions)

Is Ease of Use the *Overriding* Goal?

- MPI often described as "the assembly language of parallel programming"
- C and Fortran have been described as "portable assembly languages"
  - (That's company MPI is proud to keep)
- Ease of use is important. But *completeness* is more important.
  - Don't force users to switch to a different approach as their application evolves
    - Remember the mesh examples

Conclusions: Lessons From MPI

- A successful parallel programming model must enable more than the simple problems
  - It is nice that those are easy, but those weren't that hard to begin with
- Scalability is essential
  - Why bother with limited parallelism?
  - Just wait a few months for the next generation of hardware
- Performance is equally important
  - But not at the cost of the other items

More Lessons

- A general programming model for high-performance technical computing must address many issues to succeed, including:
- Completeness
  - Support the evolution of applications
- Simplicity
  - Focus on *users* not implementors
  - Symmetry reduces users burden
- Portability rides the hardware wave
  - Sacrifice only if the advantage is *huge* and *persistent*
  - Competitive performance and elegant design is not enough
- Composability rides the software wave
  - Leverage improvements in compilers, runtimes, algorithms
  - Matches hierarchical nature of systems

Directions For Future Programming Models

- Enabling Evolution
  - Transformations to legacy code
    - We already need this for memory locality, atomicity
    - Adding better support for detecting and recovering from faults (e.g., independent confirmation of invarient combined with parallel-I/O-enabled, user-directed checkpoints)
- New ways of thinking
  - Different operators (e.g., chemotaxis-like programming for high-fault situations)
  - Probabilistic programming (and results)
- "Small scale" ultracomputing
  - Same technology that gives us exaflops may (should!) give us deskside petaflops
  - Interactive ultracomputing

Improving Parallel Programming

- How can we make the programming of *real* applications easier?
- Problems with the Message-Passing Model
  - User's responsibility for data decomposition
  - "Action at a distance"
    - Matching sends and receives
    - Remote memory access
  - Performance costs of a library (no compile-time optimizations)
  - Need to choose a particular set of calls to match the hardware
- In summary, the lack of abstractions that match the applications

Challenges

- Must avoid the traps:
  - The challenge is not to make easy programs easier. The challenge is to make hard programs *possible*.
  - We need a "well-posedness" concept for programming tasks
    - Small changes in the requirements should only require small changes in the code
    - Rarely a property of "high productivity" languages
      - Abstractions that make easy programs easier don't solve the problem
  - Latency hiding is not the same as low latency
    - Need "Support for *aggregate operations on large collections*"
- An even harder challenge: make it hard to write incorrect programs.
  - OpenMP is not a step in the (entirely) right direction
  - In general, current shared memory programming models are very dangerous.
    - They also perform action at a distance
    - They require a kind of user-managed data decomposition to preserve performance without the cost of locks/memory atomic operations
  - Deterministic algorithms should have provably deterministic implementations

Manual Decomposition of Data Structures

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31
32	33	36	37	40	41	44	45
34	35	38	39	42	43	46	47
48	49	52	53	56	57	60	61
50	51	54	55	58	59	62	63

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

- Trick!
  - This is from a paper on dense matrix tiling for uniprocessors!
- This suggests that managing data decompositions is a common problem for real machines, whether they are parallel or not
  - Not just an artifact of MPI-style programming
  - Aiding programmers in data structure decomposition is an important part of solving the productivity puzzle

Some Questions for a Vendor

1. Do you have a optimized DGEMM?
   - Did you do it by hand?
   - Did you use ATLAS?
   - Should users choose it over the reference implementation from netlib?
2. Do you have an optimizing Fortran compiler
   - Is it effective?

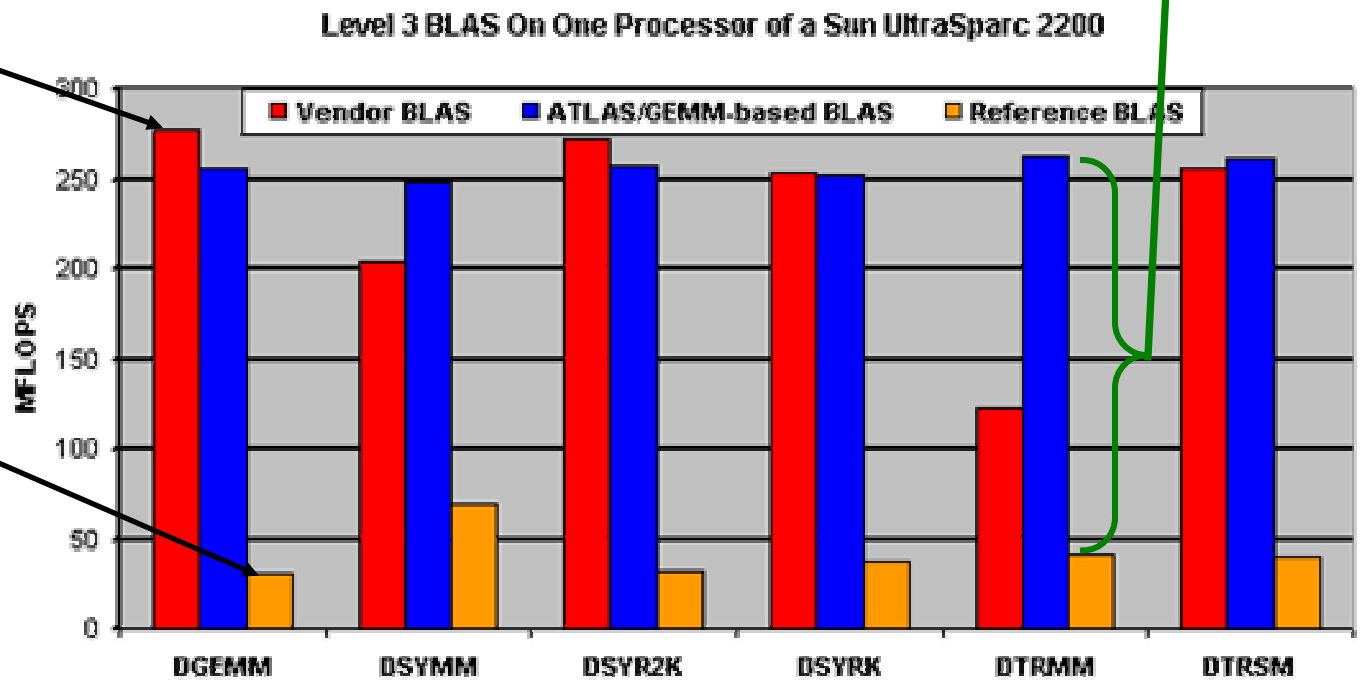
- Aren't the answers to 1 and 2 *incompatible*?

Parallelizing Compilers Are Not the Answer

Large gap between natural code and specialized code

Hand-tuned

Compiler



From Atlas

Enormous effort required to get good performance

What is Needed To Achieve *Real* High Productivity Programming

- Managing Decompositions
  - Necessary for both parallel and uniprocessor applications
  - Many levels must be managed
  - Strong dependence on problem domain (e.g., halos, load-balanced decompositions, dynamic vs. static)
- Possible approaches
  - Language-based
    - Limited by predefined decompositions
      - Some are more powerful than others; Divacon provided a built-in divided and conquer
  - Library-based
    - Overhead of library (incl. lack of compile-time optimizations), tradeoffs between number of routines, performance, and generality
  - Domain-specific languages …

Domain-specific languages

- A possible solution, particularly when mixed with adaptable runtimes
- Exploit composition of software (e.g., work with existing compilers, don't try to duplicate/replace them)
- Example: mesh handling
  - Standard rules can define mesh
    - Including "new" meshes, such as C-grids
  - Alternate mappings easily applied (e.g., Morton orderings)
  - Careful source-to-source methods can preserve human-readable code
  - In the longer term, debuggers could learn to handle programs built with language composition (they already handle 2 languages – assembly and C/Fortran/…)
- Provides a single "user abstraction" whose implementation may use the composition of hierarchical models
  - Also provides a good way to integrate performance engineering into the application

Further Reading

- For a historical perspective (and a reality check),
  - "*Enabling Technologies for Petaflops Computing*", Thomas Sterling, Paul Messina, and Paul H. Smith, MIT Press, 1995
  - "*System Software and Tools for High Performance Computing Environments*", edited by Paul Messina and Thomas Sterling, SIAM, 1993
- For current thinking on possible directions,
  - "*Report of the Workshop on High-Productivity Programming Languages and Models*", edited by Hans Zima, May 2004.