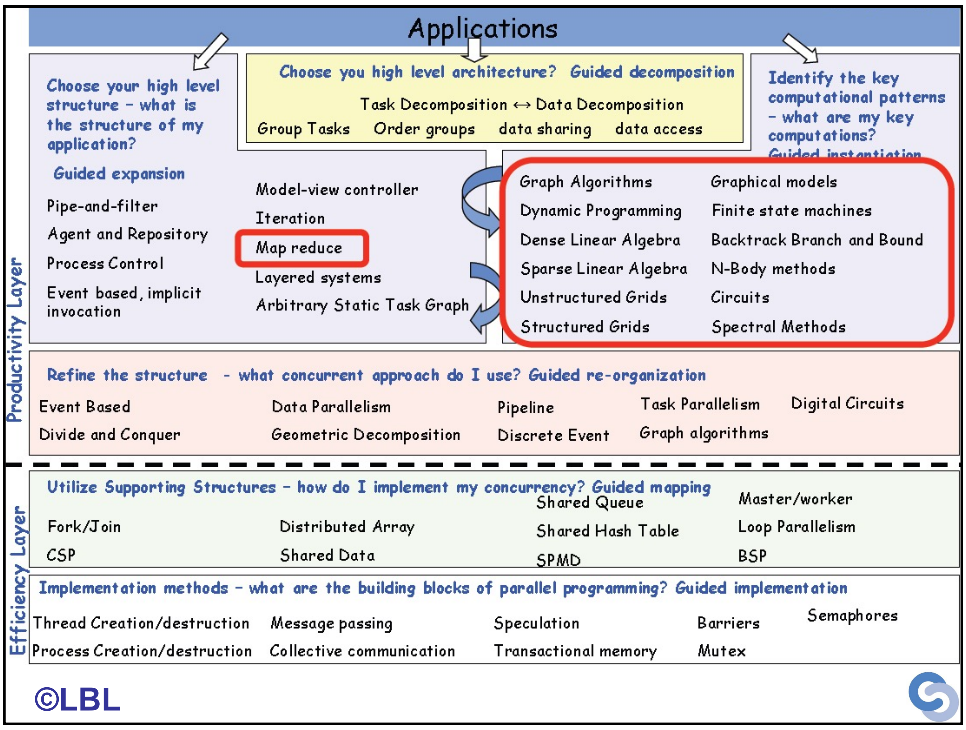
194

“7 Motifs” of High Performance Computing

- Phil Colella (LBL) identified 7 kernels of which most simulation and data-analysis programs are composed:
  1. Dense Linear Algebra
     - Ex: Solve Ax=b or Ax = λx where A is a dense matrix
  2. Sparse Linear Algebra
     - Ex: Solve Ax=b or Ax = λx where A is a sparse matrix (mostly zero)
  3. Operations on Structured Grids
     - Ex: Anew(i,j) = 4*A(i,j) – A(i-1,j) – A(i+1,j) – A(i,j-1) – A(i,j+1)
  4. Operations on Unstructured Grids
     - Ex: Similar, but list of neighbors varies from entry to entry
  5. Spectral Methods
     - Ex: Fast Fourier Transform (FFT)
  6. Particle Methods
     - Ex: Compute electrostatic forces on n particles
  7. Monte Carlo, Embarrassing Parallelism, Map Reduce, …
     - Ex: Many independent simulations using different inputs

194



195

What you want to know about a motif

196

- How to use it
  - What problems does it solve?
  - How to choose solution approach, if more than one?
- How to find the best software available now
  - Best: fastest? most accurate? fewest keystrokes?
- How are the best implementations built?
  - What is the “design space” (w.r.t. math and CS)?
  - How do we search for best solutions (autotuning)?

196

197

The Dense Linear Algebra Motif

- In the beginning was the do-loop…
- Libraries like EISPACK (for eigenvalue problems)
- Then the BLAS (1) were invented (1973-1977)
  - Standard library of 15 operations (mostly) on vectors
    - “AXPY” ($y = \alpha \cdot x + y$), dot product, scale ($x = \alpha \cdot x$), etc
    - Up to 4 versions of each (S/D/C/Z), 46 routines, 3300 LOC
  - Goals
    - Common “pattern” to ease programming, readability
    - Robustness, via careful coding (avoiding over/underflow)
    - Portability + Efficiency via machine specific implementations
  - Why BLAS 1? They do $O(n^1)$ ops on $O(n^1)$ data
  - Used in libraries like LINPACK (for linear systems)
    - Source of the name “LINPACK Benchmark” (not the code!)

197

The Dense Linear Algebra Motif (2)

- But BLAS-1 weren't enough
  - Consider AXPY ($y = \alpha \cdot x + y$): 2n flops on 3n read/writes
  - Computational intensity = (2n)/(3n) = 2/3
  - Too low to run near peak speed (read/write dominates)
  - Hard to vectorize on supercomputers of the day (1980s)
- So the BLAS-2 were invented (1984-1986)
  - Standard library of 25 operations on matrix/vector pairs
    - "GEMV": $y = \alpha \cdot A \cdot x + \beta \cdot x$, "GER": $A = A + \alpha \cdot x \cdot y^T$, $x = T^{-1} \cdot x$
    - Up to 4 versions of each (S/D/C/Z), 66 routines, 18K LOC
  - Why BLAS 2 ? They do $O(n^2)$ ops on $O(n^2)$ data
  - So computational intensity still just $\sim(2n^2)/(n^2) = 2$
    - OK for vector machines, but not for machine with caches

The Dense Linear Algebra Motif (3)

- The next step: BLAS-3 (1987-1988)
  - Standard library of 9 operations on matrix/matrix pairs
    - "GEMM": $C = \alpha \cdot A \cdot B + \beta \cdot C$, $C = \alpha \cdot A \cdot A^T + \beta \cdot C$, $B = T^{-1} \cdot B$
    - Up to 4 versions of each (S/D/C/Z), 30 routines, 10K LOC
  - Why BLAS 3 ? They do $O(n^3)$ ops on $O(n^2)$ data
  - So computational intensity $(2n^3)/(4n^2) = n/2$ – big at last!
    - Good for machines with caches, other mem. hierarchy levels
- How much BLAS1/2/3 code so far
  - Source: 142 routines, 31K LOC, Testing: 28K LOC
  - Reference (unoptimized) implementation only
    - Ex: 3 nested loops for GEMM
  - Lots more optimized code
    - Motivates "automatic tuning" of the BLAS

The Dense Linear Algebra Motif (4)

- LAPACK – “Linear Algebra PACKage”: BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of one row to other rows – BLAS-1
    - How do we reorganize GE to use BLAS-3 ? (details later)
  - Contents of LAPACK (summary)
    - Algorithms we can turn into (nearly) 100% BLAS 3: Linear Systems & Least squares
    - Algorithms that are only 50% BLAS 3 (so far): Eigenproblems & Singular Value Decomposition (SVD)
    - Error bounds for everything
    - Lots of variants depending on A’s structure
  - How much code? (Nov 2008) (www.netlib.org/lapack)
    - Source: 1582 routines, 490K LOC, Testing: 352K LOC

200

The Dense Linear Algebra Motif (5)

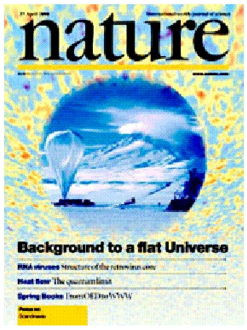
- Is LAPACK parallel?
  - Only if the BLAS are parallel (shared memory)
- ScaLAPACK – “Scalable LAPACK” (1995 – now)
  - For distributed memory – uses MPI
  - More complex data structures, algorithms than LAPACK
    - Only subset of LAPACK’s functionality available
  - All at www.netlib.org/scalapack

201

202

Success Stories for Sca/LAPACK

- Widely used
  - Adopted by Mathworks, Cray, Fujitsu, HP, IBM, IMSL, Intel, NAG, NEC, SGI, NVidia, AMD, ARM
  - 5.5M webhits/year @ Netlib (incl. CLAPACK, LAPACK95)
- New Science discovered through the solution of dense matrix systems
  - Nature article on the flat universe used ScaLAPACK
  - Other articles in Physics Review B that also use it
  - 1998 Gordon Bell Prize
  - www.nersc.gov/news/reports/newNERSCresults050703.pdf



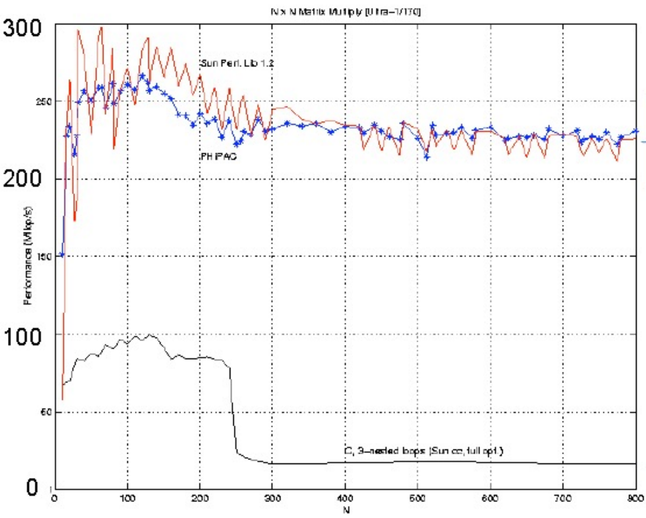
Cosmic Microwave Background Analysis, BOOMERanG collaboration, MADCAP code (Apr. 27, 2000).

ScaLAPACK

202

203

Optimized Matrix-Multiply



Peak = 330 MFlops.

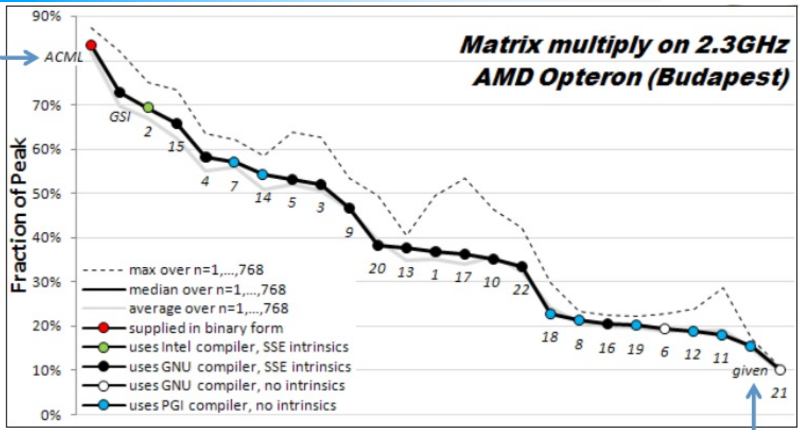
Optimized Implementations: Vendor (Sun) and Autotuned (PHiPAC)

Reference Implementation; Full compiler opt.

203

204

How hard is hand-tuning?



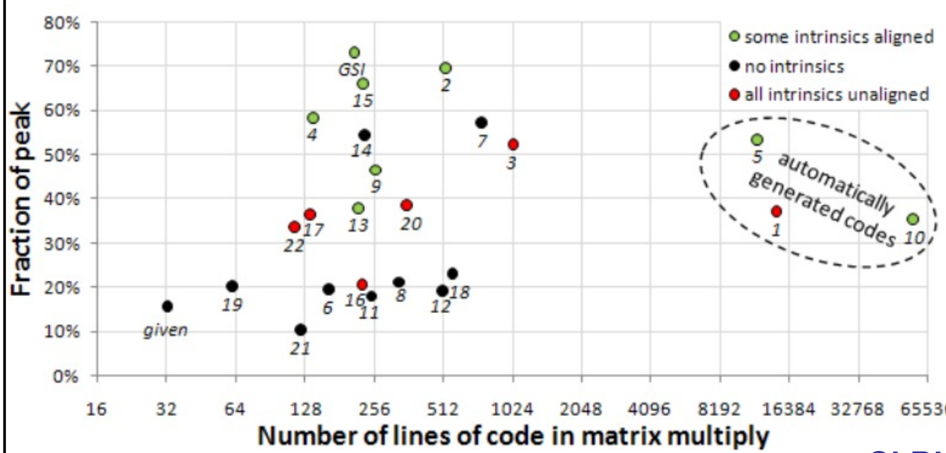
- Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09
- Students given “blocked” code to start with
  - Still hard to get close to vendor tuned performance (ACML)
- For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/
- Naïve matmul: just 2% of peak

©LBL

204

205

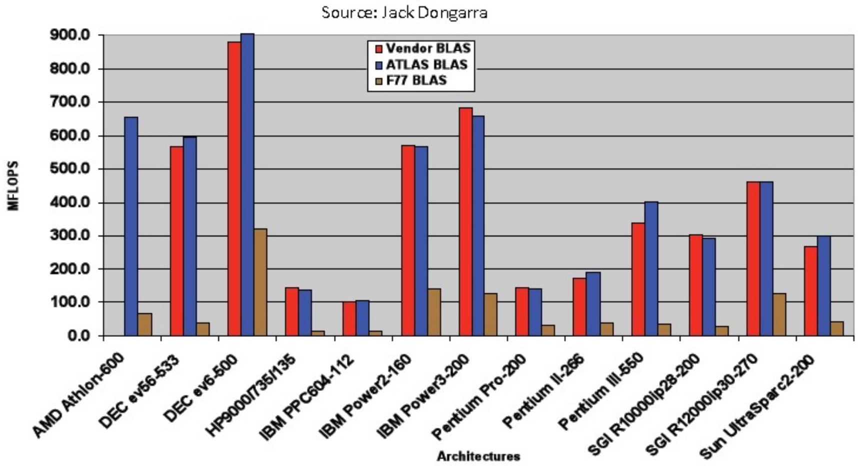
How hard is hand-tuning? (2)



©LBL

205

Autotuning DGEMM with ATLAS



ATLAS was faster than all other portable BLAS implementations & comparable with vendor libraries

The Future of Dense Linear Algebra

- Communication-Avoiding for everything
- Extensions for multicore systems
  - PLASMA – Parallel Linear Algebra for Scalable Multicore Architectures
    - Dynamically schedule tasks into which the algorithm is decomposed: minimize synchronization & keep all processors busy
- Extensions for GPUs
  - “Benchmarking GPUs to tune Dense Linear Algebra”
    - Best Student Paper Prize at SC08 (Vasily Volkov)
  - MAGMA – Matrix Algebra on GPU and Multicore Architectures
- How much code generation can we automate?
  - MAGMA , and FLAME (www.cs.utexas.edu/users/flame/)

Sparse Linear Algebra Motif

- Similar problems to dense matrices
  - Ax=b, Least squares, Ax = λx, SVD, …
- But different algorithms!
  - Exploit structure: only store, work on non-zeros
  - Direct methods
    - LU, Cholesky for Ax=b, QR for Least squares
    - See crd.lbl.gov/~xiaoye/SuperLU/index.html for LU codes
    - See crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf for a survey of available serial and parallel sparse solvers
  - Iterative methods – for Ax=b, least squares, eig, SVD
    - Use simplest operation: Sparse-Matrix-Vector-Multiply (SpMV)
    - Krylov Subspace Methods: find "best" solution in space spanned by vectors generated by SpMVs

208

Sparse Linear Algebra Motif (2)

- Fast code must **minimize communication**
  - Especially for sparse matrix computations because communication dominates
- Generating fast code for a single SpMV
  - Design space of possible algorithms must be searched at run-time, when sparse matrix available
  - Design space should be searched automatically
- **Biggest speedups from minimizing communication in an entire sparse solver**
  - Many more opportunities to minimize communication in multiple SpMVs than in one
  - Requires transforming the entire algorithm
- More information can be found: bebop.cs.berkeley.edu

209

210

ODEs and Sparse Matrices

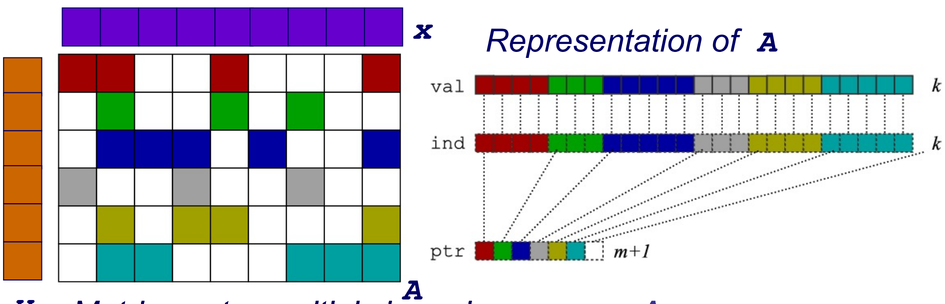
- ODEs/PDEs problems reduce to sparse matrix problems
  - Explicit: sparse matrix-vector multiplication (SpMV).
  - Implicit: solve a sparse linear system
    - Direct solvers (Gaussian elimination)
    - Iterative solvers (Use sparse matrix-vector multiplication)
  - Eigenvalue/vector algorithms may also be explicit or implicit
- Conclusion: SpMV is key to many ODE problems
  - Relatively simple algorithm to study in detail
  - Two key problems: locality and load balance

210

211

SpMV in Compressed Sparse Row (CSR) Format

SpMV: y = y + A*x, only store, do arithmetic, on nonzero entries
CSR format is simplest one of many possible data structures for A



Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j) \cdot x(j)$

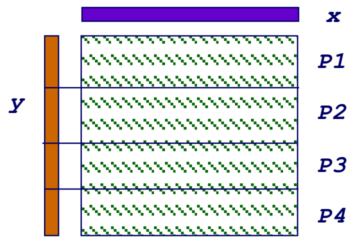
```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
       y[i] = y[i] + val[k]*x[ind[k]]
```

211

Parallel Sparse Matrix-vector multiplication

212

- y = A*x, where A is a sparse n x n matrix



- Questions
  - which processors store
    - y[i], x[i], and A[i,j]
  - which processors compute
    - y[i] = sum (from 1 to n) A[i,j] * x[j]

      = (row i of A) * x … a sparse dot product
- Partitioning
  - Partition index set {1,…,n} = N1 ∪ N2 ∪ … ∪ Np.
  - For all i in Nk, Processor k stores y[i], x[i], and row i of A
  - For all i in Nk, Processor k computes y[i] = (row i of A) * x
    - "owner computes" rule: Processor k computes the y[i]s it owns.

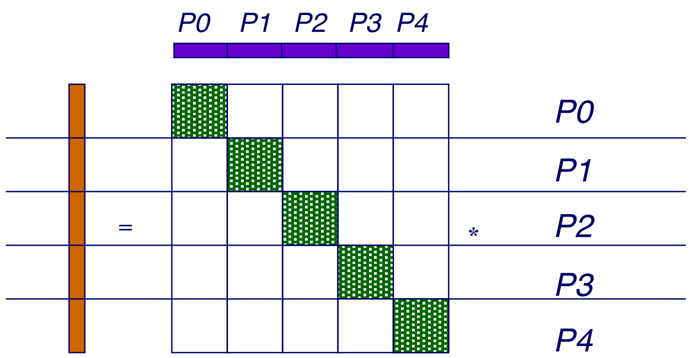
May require communication

212

Matrix Reordering via Graph Partitioning

213

- "Ideal" matrix structure for parallelism: block diagonal
  - p (number of processors) blocks, can all be computed locally
  - If no non-zeros outside these blocks, **no communication needed**
- Can we reorder the rows/columns to get close to this?
  - Most non-zeros in diagonal blocks, few outside



213

Goals of Reordering

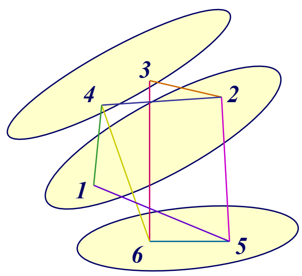
- Performance goals
  - Balance load – how is load measured?
    - Approx equal number of non-zeros (not necessarily rows)
  - Balance storage – how much does each processor store?
    - Approx equal number of non-zeros
  - Minimize communication – how much is communicated?
    - Minimize non-zeros outside diagonal blocks
    - Related optimization criterion is to move non-zeros near diagonal
  - Improve register and cache re-use
    - Group non-zeros in small vertical blocks so source (x) elements loaded into cache or registers may be reused (temporal locality)
    - Group non-zeros in small horizontal blocks so nearby source (x) elements in the cache may be used (spatial locality)
- Other algorithms reorder for other reasons
  - Reduce # non-zeros in matrix after Gaussian elimination
  - Improve numerical stability

214

Graph Partitioning and Sparse Matrices

- Relationship between matrix and graph

	1	2	3	4	5	6
1	1			1	1	
2		1	1	1	1	
3		1	1			1
4	1	1		1		1
5	1	1			1	1
6			1	1	1	1



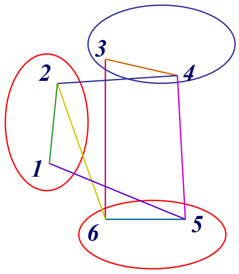
- Edges in the graph are nonzero in the matrix: here the matrix is symmetric (edges are unordered) and weights are equal (1)
- If divided over 3 procs, there are 14 nonzeros outside the diagonal blocks, which represent the 7 (bidirectional) edges

215

Graph Partitioning and Sparse Matrices (2)

- Relationship between matrix and graph

	1	2	3	4	5	6
1	1	1			1	
2	1	1		1		1
3			1	1		1
4		1	1	1	1	
5	1			1	1	1
6		1	1		1	1



- *A "good" partition of the graph has*
  - *equal (weighted) number of nodes in each part (load and storage balance).*
  - *minimum number of edges crossing between (minimize communication).*
- *Reorder the rows/columns by putting all nodes in one partition together.*

Summary: Common Problems

- Load Balancing
  - Dynamically – if load changes significantly during job
  - Statically - Graph partitioning
    - Discrete systems
    - Sparse matrix vector multiplication
- Linear algebra
  - Solving linear systems (sparse and dense)
  - Eigenvalue problems will use similar techniques
- Fast Particle Methods
  - O(n log n) instead of O(n^2)