INRIA SOPHIA ANTIPOLIS

The Impact Of Computer Architectures On Linear Algebra Algorithms and Software

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee

http://www.cs.utk.edu/~dongarra/

1

Outline

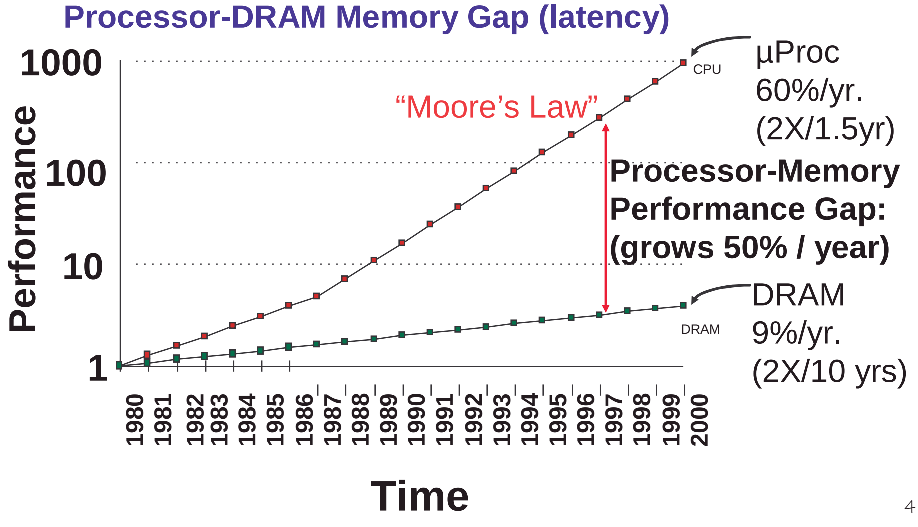
- Performance issues
- Self Adapting Software for Optimization
  - ATLAS and other examples
- Recursive Factorization
  - LU
- Performance Monitoring Tools
  - PAPI

2

High Performance Computers

- ~ 20 years ago
  - $1x10^6$ Floating Point Ops/sec (Mflop/s)
    - Scalar based
- ~ 10 years ago
  - $1x10^9$ Floating Point Ops/sec (Gflop/s)
    - Vector & Shared memory computing, bandwidth aware
    - Block partitioned, latency tolerant
- ~ Today
  - $1x10^{12}$ Floating Point Ops/sec (Tflop/s)
    - Highly parallel, distributed processing, message passing, network based
    - data decomposition, communication/computation
- ~ 10 years away
  - $1x10^{15}$ Floating Point Ops/sec (Pflop/s)
    - Many more levels MH, combination/grids&HPC
    - More adaptive, LT and bandwidth aware, fault tolerant, extended precision, attention to SMP nodes

Where Does the Performance Go? or Why Should I Care About the Memory Hierarchy?



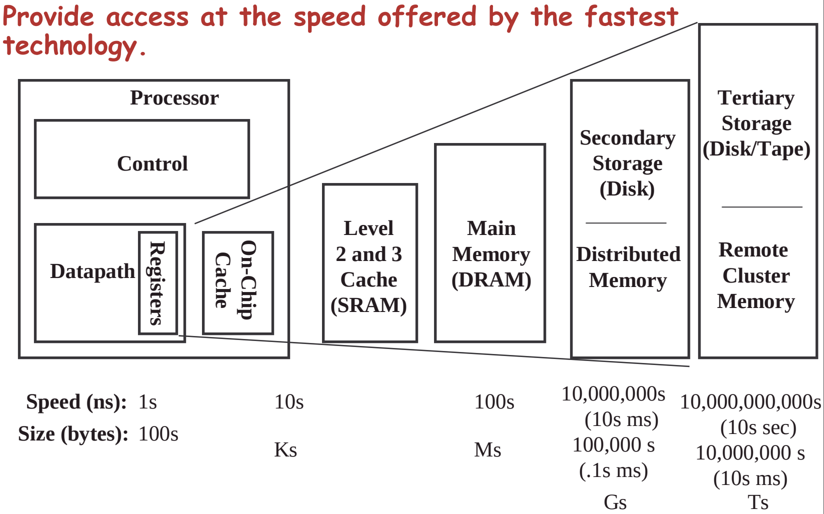
Optimizing Computation and Memory Use

- **Computational optimizations**
  - Theoretical peak:(# fpus)*(flops/cycle) * Mhz
    - Pentium III: (1 fpu)*(1 flop/cycle)*(850 Mhz) = 850 MFLOP/s
    - Pentium 4: (1 fpu)*(2 flops/cycle)*(2.53 Ghz) = 5060 MFLOP/s
    - Athlon: (2 fpu)*(1flop/cycle)*(600 Mhz) = 1200 MFLOP/s
    - Power3: (2 fpu)*(2 flops/cycle)*(375 Mhz) = 1500 MFLOP/s
- **Operations like:**
  - $\alpha = x^T y$: 2 operands (16 Bytes) needed for 2 flops; at 850 Mflop/s will requires 1700 MW/s bandwidth
  - $y = \alpha x + y$: 3 operands (24 Bytes) needed for 2 flops; at 850 Mflop/s will requires 2550 MW/s bandwidth
- **Memory optimization**
  - Theoretical peak: (bus width) * (bus speed)
    - Pentium III: (32 bits)*(133 Mhz) = 532 MB/s = 66.5 MW/s
    - Pentium 4: (32 bits)*(533 Mhz) = 2132 MB/s = 266 MW/s
    - Athlon: (64 bits)*(133 Mhz) = 1064 MB/s = 133 MW/s
    - Power3: (128 bits)*(100 Mhz) = 1600 MB/s = 200 MW/s

5

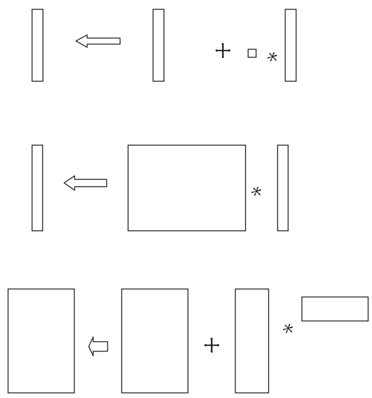
Memory Hierarchy

- **By taking advantage of the principle of locality:**
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Level 1, 2 and 3 BLAS

- **Level 1 BLAS Vector-Vector operations**
- **Level 2 BLAS Matrix-Vector operations**
- **Level 3 BLAS Matrix-Matrix operations**

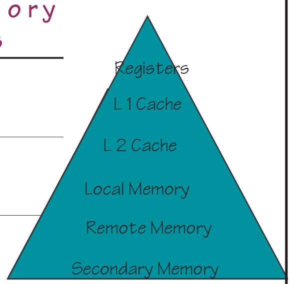


7

Why Higher Level BLAS?

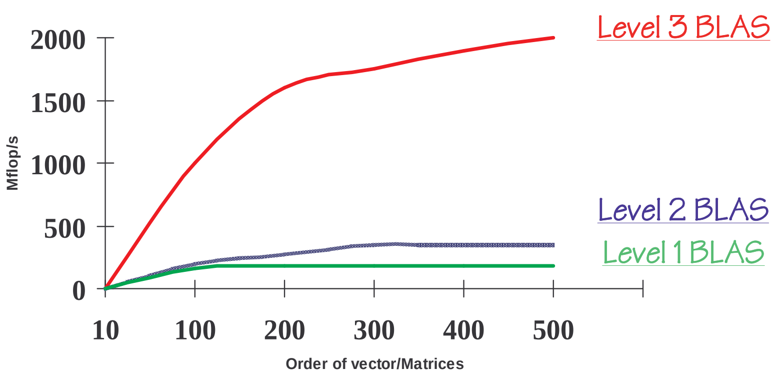
- **Can only do arithmetic on data at the top of the hierarchy**
- **Higher level BLAS lets us do this**

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y = y + \alpha x$	$3n$	$2n$	$2/3$
Level 2 $y = y + Ax$	n^2	$2n^2$	2
Level 3 $C = C + AB$	$4n^2$	$2n^3$	$n/2$



BLAS for Performance

Intel Pentium 4 w/SSE2 1.7 GHz



- Development of blocked algorithms important for performance

9

6 Variations of Matrix Multiple

for _ = 1:n;

  for _ = 1:n;

    for _ = 1:n;

      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$

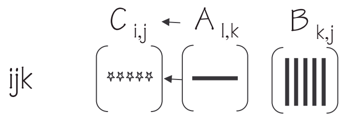
    end

  end

end

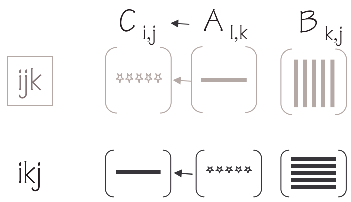
10

6 Variations of Matrix Multiple



for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k}B_{k,j}$
    end
  end
end

6 Variations of Matrix Multiple



for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k}B_{k,j}$
    end
  end
end

6 Variations of Matrix Multiple

for _ = 1:n;

for _ = 1:n;

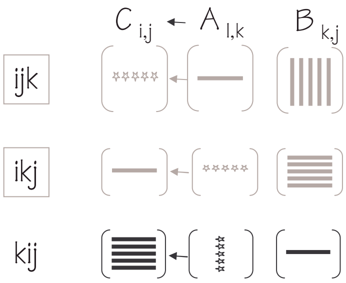
for _ = 1:n;

$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$

end

end

end



13

6 Variations of Matrix Multiple

for _ = 1:n;

for _ = 1:n;

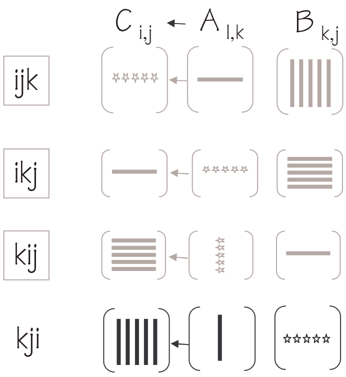
for _ = 1:n;

$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$

end

end

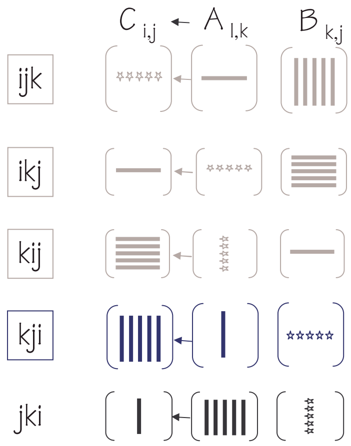
end



14

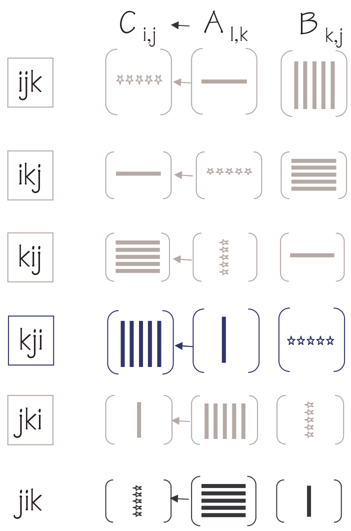
6 Variations of Matrix Multiple

for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end



6 Variations of Matrix Multiple

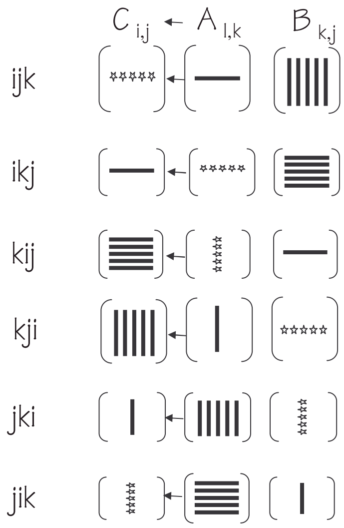
for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end



6 Variations of Matrix Multiple

for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end

C
Fortran

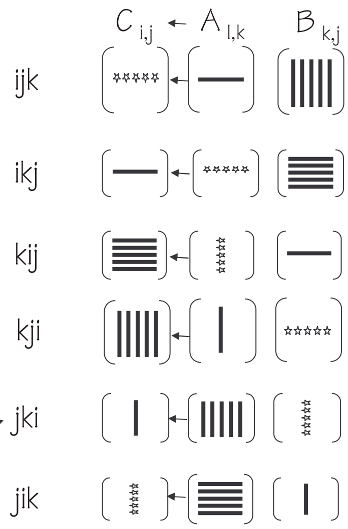


17

6 Variations of Matrix Multiple

for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end

C
Fortran



However, only part of the story

18

Matrices in Cache

For a Pentium III 933 MHz
L1 data cache 16 KB (also has a L1 instruction cache 16 KB)

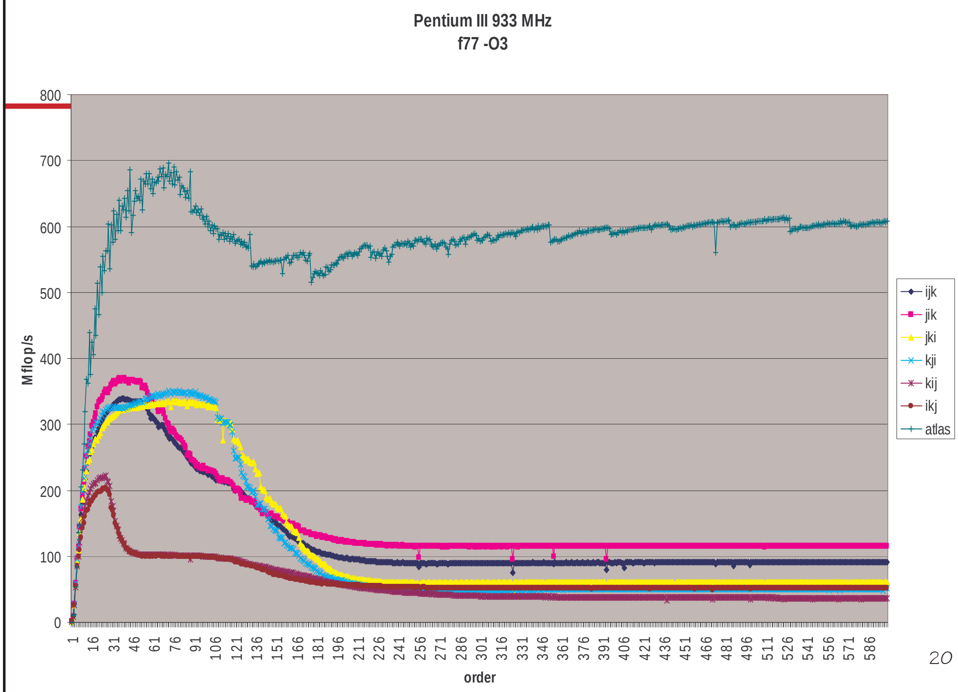
$$\sqrt{16KB/8} \approx 45$$

- L2 cache 256 KB
  - Sqrt(256K/8) = 179

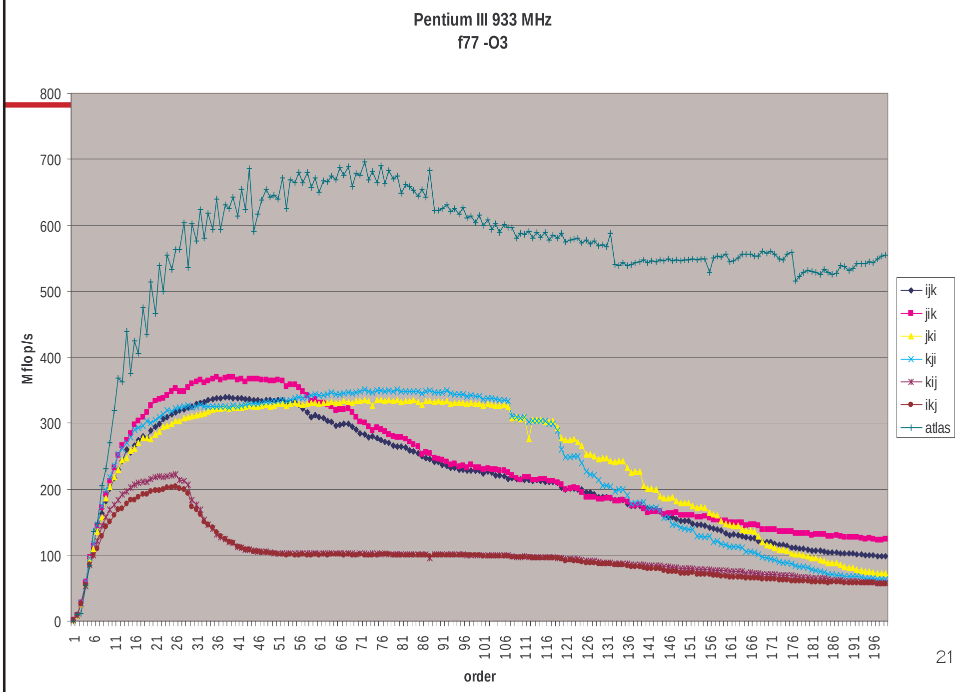
For a Pentium III 550 MHz
L1 data cache 16 KB (also has a L1 instruction cache 16 KB)

- L2 cache 512 KB
  - Sqrt(512K/8) = 252

19

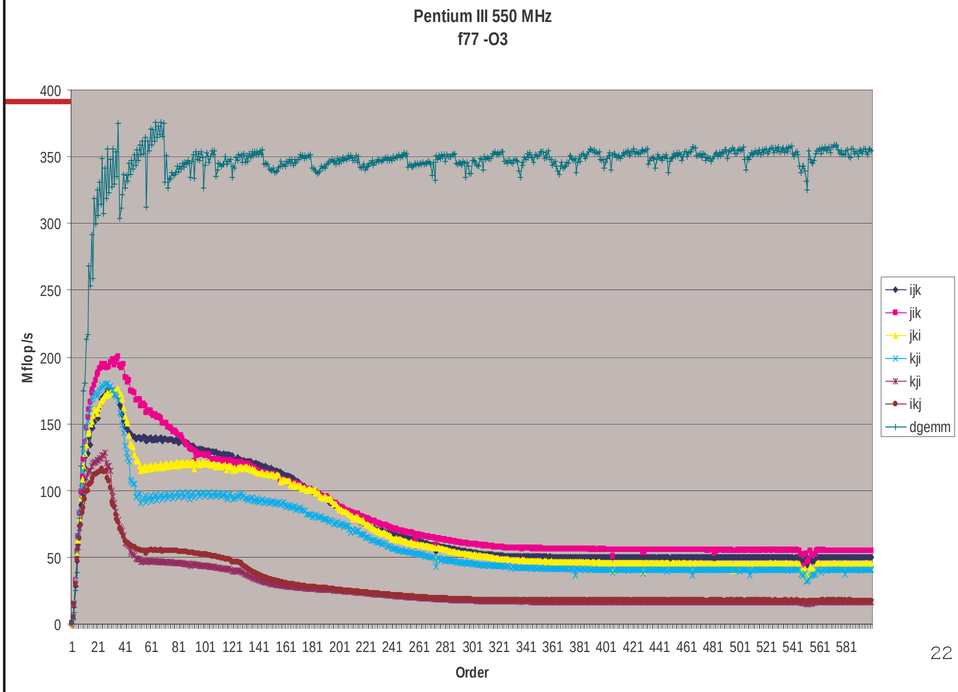


20



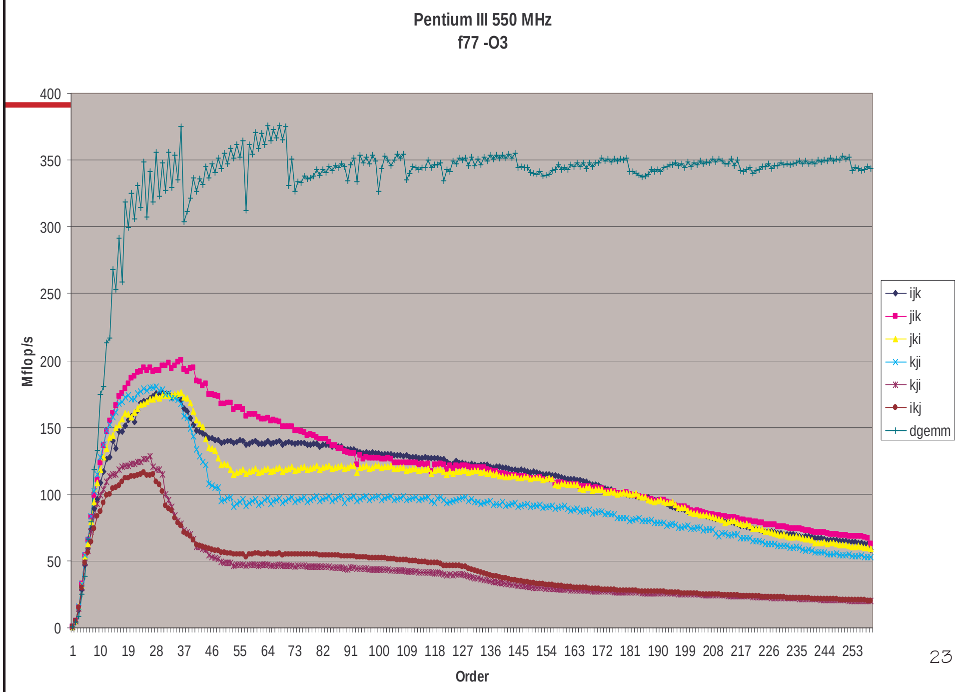
Pentium III 933 MHz
f77 -O3
Mflop/s
order
ijk
jik
jki
kji
kij
ikj
atlas

21



Pentium III 550 MHz
f77 -O3
Mflop/s
Order
ijk
jik
jki
kji
kji
ikj
dgemm

22



23

Matrix Multiply Assumption Data in Cache

- **DOT version – in cache**

```
      DO 30 J = 1, M
         DO 20 I = 1, M
            DO 10 K = 1, L
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
10          CONTINUE
20       CONTINUE
30 CONTINUE
```

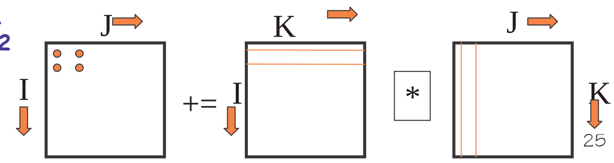
- **Inner loop:**
  - 2 loads, 2 operations, suboptimal.
  - No reuse of registers

24

How to Get Near Peak

```
      DO 30 J = 1, M, 2
         DO 20 I + 1, M, 2
            T11 = C(I,  J  )
            T12 = C(I,  J+1)
            T21 = C(I+1,J  )
            T22 = C(I+1,J+1)
            DO 10 K = 1, L
               T11 = T11 + A(I,  K) *B(K,J  )
               T12 = T12 + A(I,  K) *B(K,J+1)
               T21 = T21 + A(I+1,K)*B(K,J  )
               T22 = T22 + A(I+1,K)*B(K,J+1)
10          CONTINUE
            C(I,  J  ) = T11
            C(I,  J+1) = T12
            C(I+1,J  ) = T21
            C(I+1,J+1) = T22
20       CONTINUE
30 CONTINUE
```

- Inner loop:
  - 4 loads, 8 operations, optimal.
  - Reuse data in registers

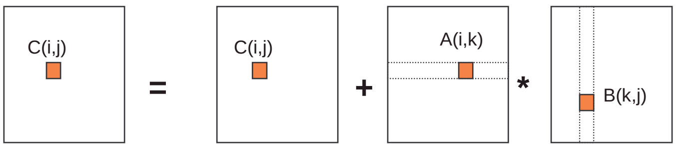


- For a Pentium III 933 MHz
  - Peak 933 Mflop/s
  - Best can do around 2/3 peak, has to do with the stack architecture
  - 2 level of cache 16KB and 256KB
- Note 4 different performance levels
  - Bad cache use
  - Level 1 cache, then exceeds
  - Level 2 cache, then exceeds
  - Putting it all together
- Problems too large for cache, do blocking
- Unrolling for register reuse critical

Matrix Multiply (blocked, or tiled)

Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

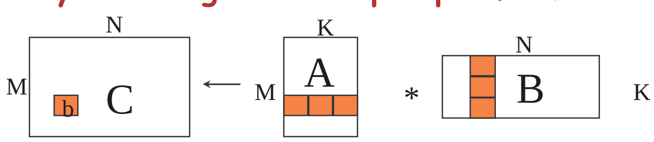
```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
                {read block A(i,k) into fast memory}
                {read block B(k,j) into fast memory}
                C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```



n is the size of the matrix, N blocks of size b; n = N*b

Adaptive Approach for Level 3

- Do a parameter study of the operation on the target machine, done once.
- Only generated code is Level 1 Cache multiply
- BLAS operation written in terms of generated on-chip multiply
- All tranpose cases coerced through data copy to 1 case of on-chip multiply
  - Only 1 case generated per platform

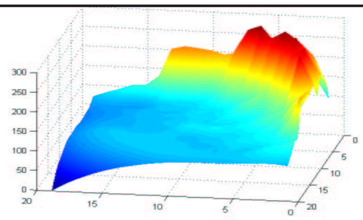


Self-Adapting Numerical Software (SANS)

- Today’s processors can achieve high-performance, but this requires extensive machine-specific hand tuning.
- Operations like the BLAS require many man-hours / platform
  - Software lags far behind hardware introduction
  - Only done if financial incentive is there
- Hardware, compilers, and software have a large design space w/many parameters
  - Blocking sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction schedules.
  - Complicated interactions with the increasingly sophisticated micro-architectures of new microprocessors.
- Need for quick/dynamic deployment of optimized routines.
- ATLAS - Automatic Tuned Linear Algebra Software

29

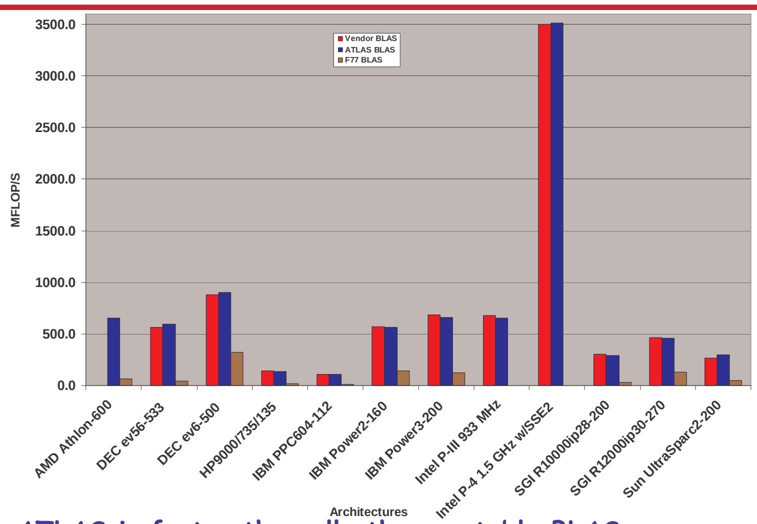
Software Generation Strategy



- Level 1 cache multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory fetch
  - Register reuse
  - Loop overhead minimization
- Takes about 30 minutes to run.
- “New” model of high performance programming where critical code is machine generated using parameter optimization.
- Code is iteratively generated & timed until optimal case is found. We try:
  - Differing NBs
  - Breaking false dependencies
  - M, N and K loop unrolling
- Designed for RISC arch
  - Super Scalar
  - Need reasonable C compiler
- Today ATLAS in use by Matlab, Mathematica, Octave, Maple, Debian, Scyld Beowulf, SuSE, …

30

ATLAS (DGEMM n = 500)



- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

31

MATLAB

- Currently over 500,000 MATLAB licenses
- Matlab gives simplicity and power but not performance
  - Codes prototyped in MATLAB
  - User would rewrite in Fortran or C later
- Well…
- Today MATLAB uses ATLAS BLAS and LAPACK
  - Great performance for these operations
  - But no interoperation optimization in MATLAB
- Demo

32

Some Automatic Tuning Projects

- ATLAS (www.netlib.org/atlas) (Dongarra, Whaley)
- PHIPAC (www.icsi.berkeley.edu/~bilmes/phipac) (Bilmes,Asanovic,Vuduc,Demmel)
- Sparse matrix operations, (Yelick, Im & Dongarra, Eijkhout)
- Communication topologies (Dongarra)
- FFTs and Signal Processing
  - FFTW (www.fftw.org)
    - Won 1999 Wilkinson Prize for Numerical Software
  - SPIRAL (www.ece.cmu.edu/~spiral)
    - Extensions to other transforms, DSPs
  - UHFFT
    - Extensions to higher dimension, parallelism

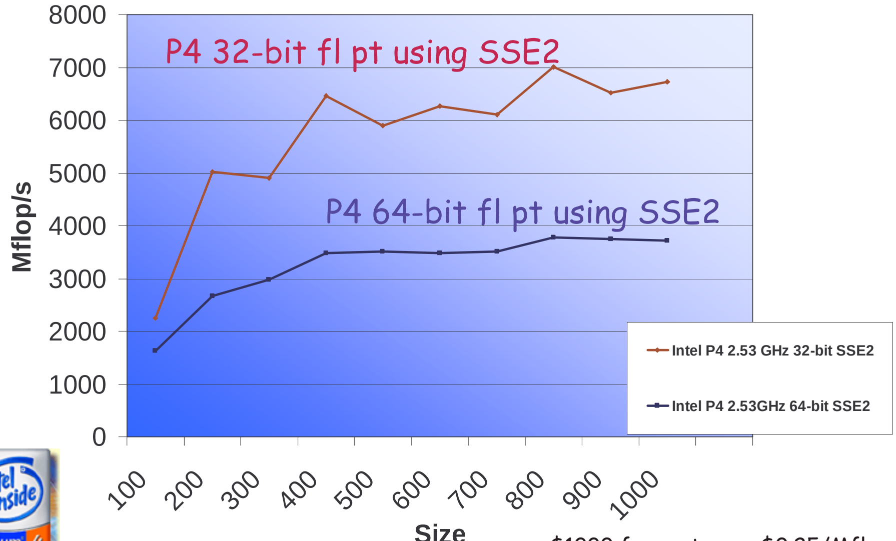
33

Pentium 4 - SSE2
Today's "Sweet Spot" in Price/Performance

- 2.53 GHz, 400 MHz system bus, 16K L1 & 256K L2 Cache, theoretical peak of 2.53 Gflop/s, high power consumption
- Streaming SIMD Extensions 2 (SSE2)
  - which consists of 144 new instructions
  - includes SIMD IEEE double precision floating point
    - Peak for 64 bit floating point 2X (5.06 Gflop/s)
    - Peak for 32 bit floating point 4X (10.12 Gflop/s)
  - SIMD 128-bit integer
  - new cache and memory management instructions.
  - Intel's compiler supports these instructions today
  - ATLAS was trained to probe and detect SSE2

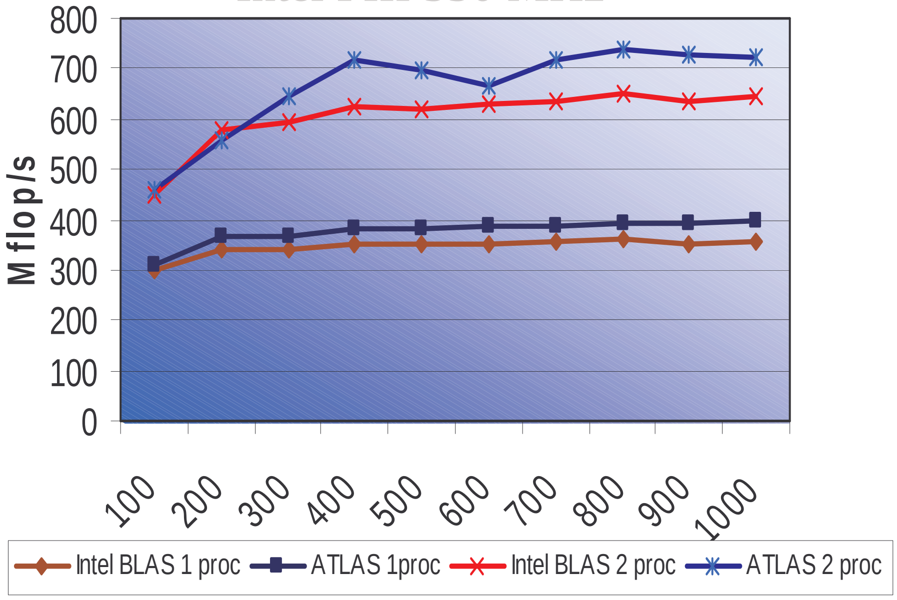
34

ATLAS Matrix Multiply
Intel Pentium 4 at 2.53GHz – using SSE2

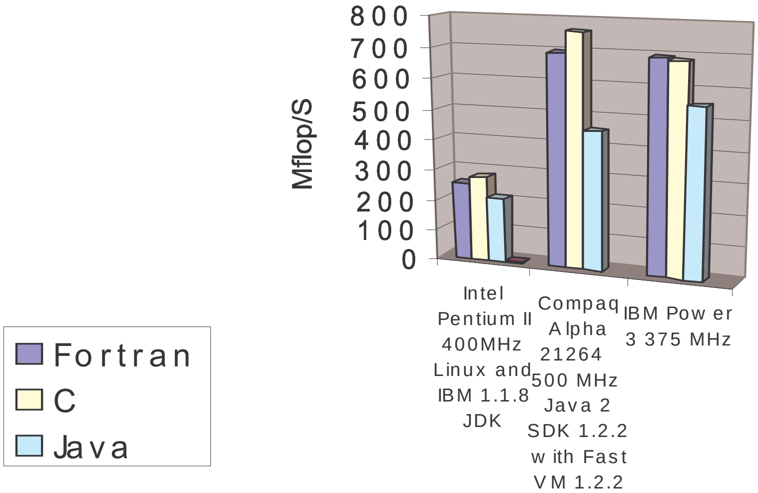


~$1000 for system => $0.25/Mflops !!

Multi-Threaded DGEMM
Intel PIII 550 MHz



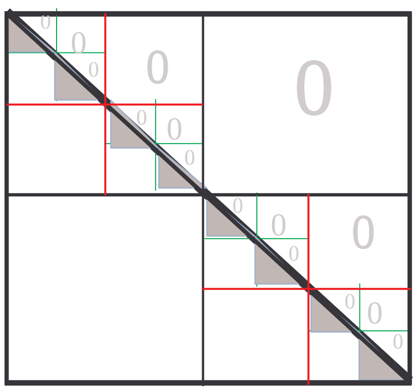
Experiments with C, Fortran, and Java for ATLAS (DGEMM kernel)



Recursive Approach for Other Level 3 BLAS

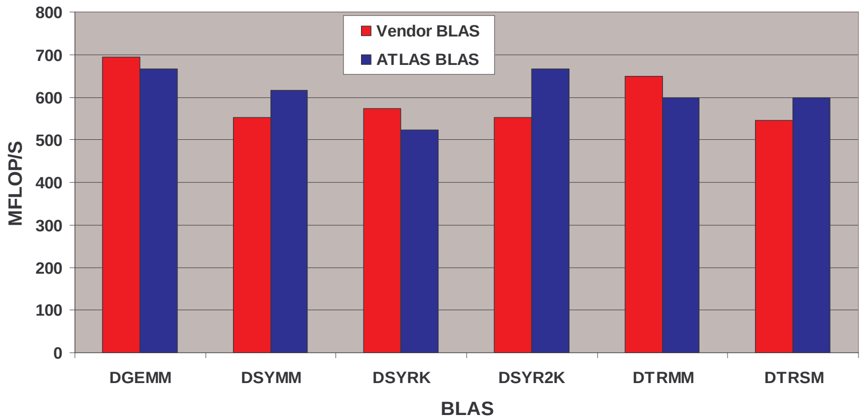
- Recur down to L1 cache block size
- Need kernel at bottom of recursion
  - Use gemm-based kernel for portability

Recursive TRMM



38

Intel PIII 933 MHz
MKL 5.0 vs ATLAS 3.2.0 using Windows 2000



- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

39

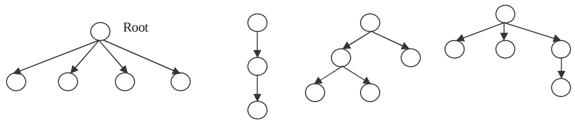
Machine-Assisted Application Development and Adaptation

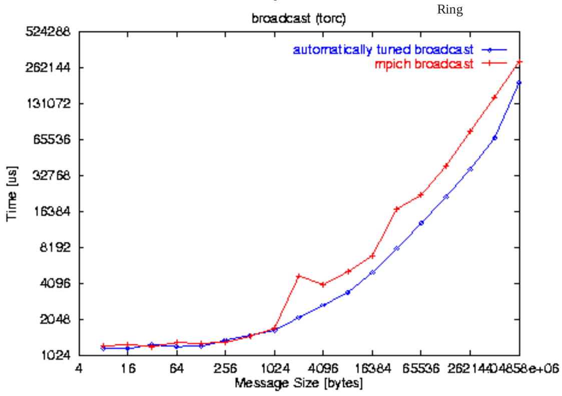
- Communication libraries
  - Optimize for the specifics of one's configuration.
- Algorithm layout and implementation
  - Look at the different ways to express implementation

40

Work in Progress: ATLAS-like Approach Applied to Broadcast

(PII 8 Way Cluster with 100 Mb/s switched network)





Message Size (bytes)	Optimal algorithm	Buffer Size (bytes)
8	binomial	8
16	binomial	16
32	binary	32
64	binomial	64
128	binomial	128
256	binomial	256
512	binomial	512
1K	sequential	1K
2K	binary	2K
4K	binary	2K
8K	binary	2K
16K	binary	4K
32K	binary	4K
64K	ring	4K
128K	ring	4K
256K	ring	4K
512K	ring	4K
1M	binary	4K

1

Reformulating/Rearranging/Reuse

- Example is the reduction to narrow band from for the SVD

$$A_{new} = A - uy^T - wv^T$$

$$y_{new} = A^T u$$

$$w_{new} = A_{new} v$$

- Fetch each entry of A once
- Restructure and combined operations
- Results in a speedup of > 30%

42

CG Variants by Dynamic Selection at Run Time

- Variants combine inner products to reduce communication bottleneck at the expense of more scalar ops.
- Same number of iterations, no advantage on a sequential processor
- With a large number of processor and a high-latency network may be advantages.
- Improvements can range from 15% to 50% depending on size.

Classical
Norm calculation:
$\boxed{error = \sqrt{r^t r}}$
Preconditioner application:
$z \leftarrow M^{-1} r$
Matrix-vector product:
Inner products 1:
$\boxed{\rho \leftarrow z^t r}$
$\beta \leftarrow \rho / \rho_{old}$
Search direction update:
$p \leftarrow z + \beta p$
Matrix-vector product:
$ap \leftarrow A \times p$
Preconditioner application:
Inner products 2:
$\boxed{\pi \leftarrow p^t ap}$
$\alpha = \rho / \pi$
Residual update:
$r \leftarrow r - \alpha Ap$
3 separate inner products

43

CG Variants by Dynamic Selection at Run Time

- Variants combine inner products to reduce communication bottleneck at the expense of more scalar ops.
- Same number of iterations, no advantage on a sequential processor
- With a large number of processor and a high-latency network may be advantages.
- Improvements can range from 15% to 50% depending on size.

Classical	Saad/Meurant	Chronopoulos/Gear	Eijkhout
Norm calculation:			
$\boxed{error = \sqrt{r^t r}}$			
Preconditioner application:			
$z \leftarrow M^{-1} r$	$z \leftarrow z - \alpha q$	$z \leftarrow M^{-1} r$	id
Matrix-vector product:			
		$az \leftarrow A \times z$	id
Inner products 1:			
$\boxed{\rho \leftarrow z^t r}$	$\rho_{predict} \leftarrow -\rho_{true} + \alpha^2 \mu$	$error = \sqrt{r^t r}$ $\rho \leftarrow z^t r$ $\zeta \leftarrow z^t az$	$error = \sqrt{r^t r}$ $\rho \leftarrow z^t r$ $\zeta \leftarrow z^t az$ $\epsilon \leftarrow (M^{-1} r)^t (Ap)$
$\beta \leftarrow \rho / \rho_{old}$	$\beta = \rho_{predict} / \rho_{old}$	$\beta \leftarrow \rho / \rho_{old}$	id
Search direction update:			
$p \leftarrow z + \beta p$	id	id	id
Matrix-vector product:			
$ap \leftarrow A \times p$	id	$ap \leftarrow az + \beta ap$	id
Preconditioner application:	$q \leftarrow M^{-1} ap$		
Inner products 2:			
$\boxed{\pi \leftarrow p^t ap}$	$\pi \leftarrow p^t ap$ $\mu \leftarrow ap^t q$ $error = \sqrt{r^t r}$ $\rho_{true} = z^t r$	$\pi \leftarrow \zeta - \beta^2 \pi$	$\pi \leftarrow \zeta + \beta \epsilon$
$\alpha = \rho / \pi$	$\cdots \rho_{true} \cdots$	$\alpha = \rho / \pi$	
Residual update:			
$r \leftarrow r - \alpha Ap$	id	id	id
3 separate inner products	4 combined	3 combined	4 combined
	1 extra vector update	id	id

44

History of Block Partitioned Algorithms

- Early algorithms involved use of small main memory using tapes as secondary storage.
- Recent work centers on use of vector registers, level 1 and 2 cache, main memory, and "out of core" memory.

45

Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or Q^TC
- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems

46

LAPACK

- Linear Algebra library in Fortran 77
  - Solution of systems of equations
  - Solution of eigenvalue problems
- Combine algorithms from LINPACK and EISPACK into a single package
- Efficient on a wide range of computers
  - RISC, Vector, SMPs
- User interface similar to LINPACK
  - Single, Double, Complex, Double Complex
- Built on the Level 1, 2, and 3 BLAS

47

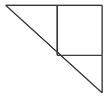
LAPACK

- Most of the parallelism in the BLAS.
- Advantages of using the BLAS for parallelism:
  - Clarity
  - Modularity
  - Performance
  - Portability

48

Derivation of Blocked Algorithms Cholesky Factorization A = U^TU

$$\begin{pmatrix} A_{11} & a_j & A_{13} \\ a_j^T & a_{jj} & \alpha_j^T \\ A_{13}^T & \alpha_j & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ u_j^T & u_{jj} & 0 \\ U_{13}^T & \mu_j & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & u_j & U_{13} \\ 0 & u_{jj} & \mu_j^T \\ 0 & 0 & U_{33} \end{pmatrix}$$



Equating coefficient of the j^{th} column, we obtain

$$a_j = U_{11}^T u_j$$

$$a_{jj} = u_j^T u_j + u_{jj}^2$$

Hence, if U_{11} has already been computed, we can compute u_j and u_{jj} from the equations:

$$U_{11}^T u_j = a_j$$

$$u_{jj}^2 = a_{jj} - u_j^T u_j$$

49

LINPACK Implementation

- **Here is the body of the LINPACK routine SPOFA which implements the method:**

```
    DO 30 J = 1, N
       INFO = J
       S = 0.0E0
       JM1 = J - 1
       IF( JM1.LT.1 ) GO TO 20
       DO 10 K = 1, JM1
          T = A( K, J ) - SDOT( K-1, A( 1, K ), 1,A( 1, J ), 1 )
          T = T / A( K, K )
          A( K, J ) = T
          S = S + T*T
 10    CONTINUE
 20    CONTINUE
       S = A( J, J ) - S
C      ...EXIT
       IF( S.LE.0.0E0 ) GO TO 40
       A( J, J ) = SQRT( S )
 30 CONTINUE
```

50

LAPACK Implementation

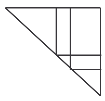
```
DO 10 J = 1, N
   CALL STRSV( 'Upper', 'Transpose', 'Non-Unit', J-1, A, LDA, A( 1, J ), 1 )
   S = A( J, J ) - SDOT( J-1, A( 1, J ), 1, A( 1, J ), 1 )
   IF( S.LE.ZERO ) GO TO 20
   A( J, J ) = SQRT( S )
10 CONTINUE
```

- This change by itself is sufficient to significantly improve the performance on a number of machines.
- From 238 to 312 Mflop/s for a matrix of order 500 on a Pentium 4-1.7 GHz.
- However on peak is 1,700 Mflop/s.
- Suggest further work needed.

51

Derivation of Blocked Algorithms

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12}^T & A_{22} & A_{12} \\ A_{13}^T & A_{12}^T & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & U_{22}^T & 0 \\ U_{13}^T & U_{23}^T & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23}^T \\ 0 & 0 & U_{33} \end{pmatrix}$$



Equating coefficient of second block of columns, we obtain

$$A_{12} = U_{11}^T U_{12}$$

$$A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}$$

Hence, if U_{11} has already been computed, we can compute U_{12} as the solution of the following equations by a call to the Level 3 BLAS routine STRSM:

$$U_{11}^T U_{12} = A_{12}$$

$$U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$$

52

LAPACK Blocked Algorithms

```
 DO 10 J = 1, N, NB
    CALL STRSM( 'Left', 'Upper', 'Transpose','Non-Unit', J-1, JB, ONE, A, LDA,
$            A( 1, J ), LDA )
    CALL SSYRK( 'Upper', 'Transpose', JB, J-1,-ONE, A( 1, J ), LDA, ONE,
$            A( J, J ), LDA )
    CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
    IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

•On Pentium 4, L3 BLAS squeezes a lot more out of 1 proc

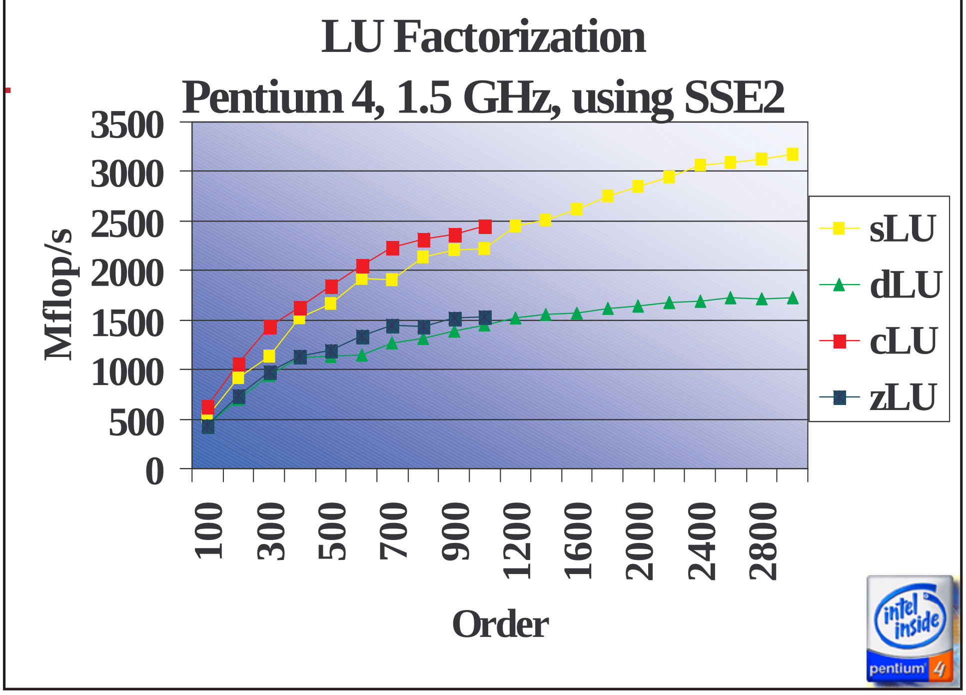
Intel Pentium 4 1.7 GHz N = 500	Rate of Execution
Linpack variant (L1B)	238 Mflop/s
Level 2 BLAS Variant	312 Mflop/s
Level 3 BLAS Variant	1262 Mflop/s

53

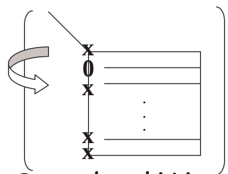
LAPACK Contents

- Combines algorithms from LINPACK and EISPACK into a single package. User interface similar to LINPACK.
- Built on the Level 1, 2 and 3 BLAS, for high performance (manufacturers optimize BLAS)
- LAPACK does not provide routines for structured problems or general sparse matrices (i.e sparse storage formats such as compressed-row, -column, -diagonal, skyline ...).

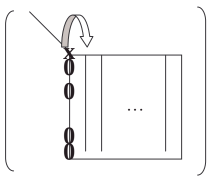
54



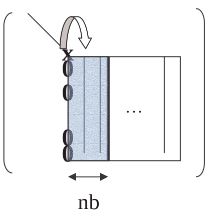
Gaussian Elimination

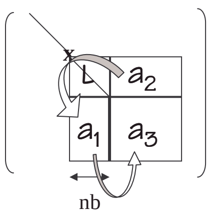


Standard Way
subtract a multiple of a row



LINPACK
apply sequence to a column





$a_2 = L^{-1} a_2$

$a_3 = a_3 - a_1 * a_2$

LAPACK
apply sequence to nb then apply nb to rest of matrix

56

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

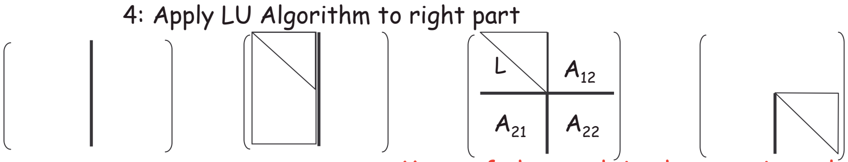
LU Algorithm:

1: Split matrix into two rectangles (m x n/2)
if only 1 column, scale by reciprocal of pivot & return

2: Apply LU Algorithm to the left part

3: Apply transformations to right part
(triangular solve $A_{12} = L^{-1}A_{12}$ and
matrix multiplication $A_{22}=A_{22} - A_{21}*A_{12}$)

4: Apply LU Algorithm to right part



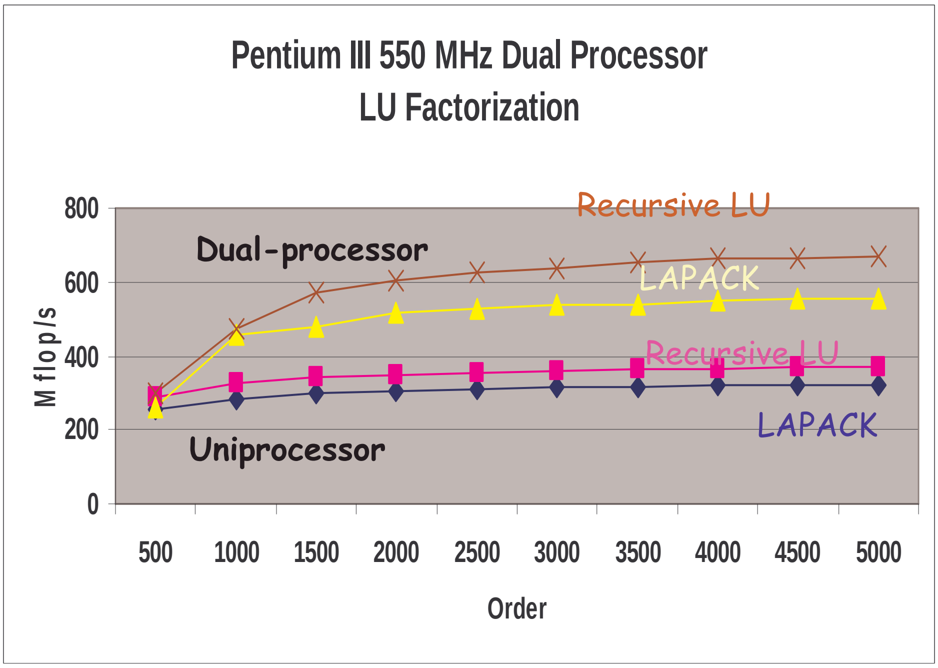
Most of the work in the matrix multiply
Matrices of size n/2, n/4, n/8, …

57

Recursive Factorizations

- Just as accurate as conventional method
- Same number of operations
- Automatic variable blocking
  - Level 1 and 3 BLAS only !
- Extreme clarity and simplicity of expression
- Highly efficient
- The recursive formulation is just a rearrangement of the point-wise LINPACK algorithm
- The standard error analysis applies (assuming the matrix operations are computed the "conventional" way).

58



Dense recursive factorization

- **The algorithm:**

function rlu(A)

begin

  rlu(A_{11}); recursive call

  $A_{21} \leftarrow A_{21} \cdot U^{-1}(A_{11})$; xTRSM() on upper triangular submatrix

  $A_{12} \leftarrow L_1^{-1}(A_{11}) \cdot A_{12}$; xTRSM() on lower triangular submatrix

  $A_{22} \leftarrow A_{22} - A_{21} \cdot A_{12}$; xGEMM()

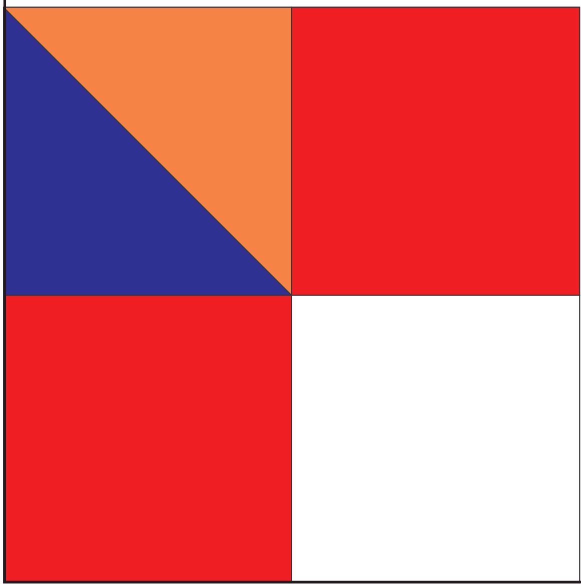
  rlu(A_{22}); recursive call

end.

- **Replace xTRSM and xGEMM with sparse implementations that are themselves recursive**

60

Recursive LU Factorization



function RLU(A)
begin
  RLU(A_{11})

  $A_{21} := A_{21}\, \mathbf{U}^{-1}\, (A_{11})$
  DTRSM()

  $A_{12} := \mathbf{L}_1^{-1}\, (A_{11})\, A_{12}$
  DTRSM()

  $A_{22} := A_{22} - A_{21}\, A_{12}$
  DGEMM()

  RLU(A_{22})

61

Sparse Factorization Assumptions

- Sparse recursive LU factorization
  - Based on recursive formulation of LU factorization
  - No partial pivoting during factorization
    - Diagonal zeros replaced with small elements, eg. ε||A||
    - Iterative refinement used to regain precision
  - Locate dense blocks, performance comes from the use of BLAS Level 3
  - Aimed at improving time to solution – memory usage may suffer

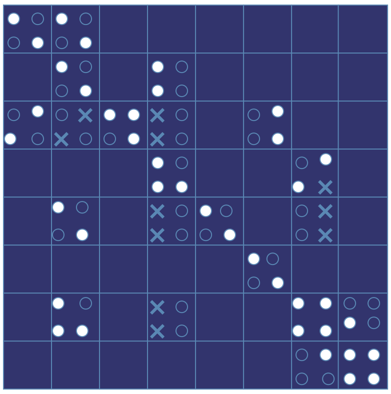
62

Sparse Recursive Factorization Algorithm

- Solutions - continued
  - fast sparse xGEMM() is two-level algorithm
    - recursive operation on sparse data structures
    - dense xGEMM() call when recursion reaches single block
- Uses Reverse Cuthill-McKee ordering causing fill-in around the band
- No partial pivoting
  - use iterative improvement or
  - pivot only within blocks

63

2. Symbolic Factorization
3. Search for Dense blocks



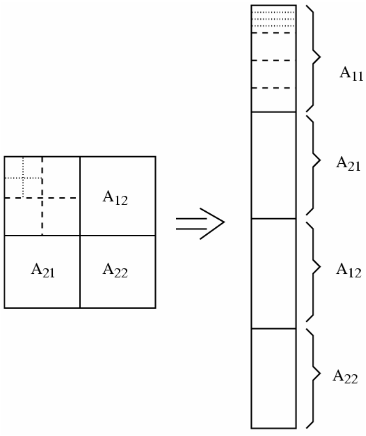
- original nonzero value
  zero value introduced due to fill-in
  zero value introduced due to blocking

64

Recursive Factorization Applied to Sparse Direct Methods

1. Symbolic Factorization
2. Search for blocks that contain non-zeros
3. Conversion to sparse recursive storage
4. Search for embedded blocks
5. Numerical factorization

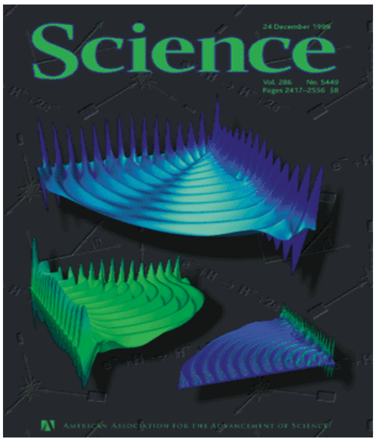
Layout of sparse recursive matrix in storage follows recursion



65

SuperLU - High Performance Sparse Solvers

- SuperLU; X. Li and J. Demmel
  - Solve sparse linear system Ax=b using Gaussian elimination.
  - Efficient and portable implementation on modern architectures:
    - Sequential SuperLU : PC and workstations
      - Achieved up to 40% peak Megaflop rate
    - SuperLU_MT : shared-memory parallel machines
      - Achieved up to 10 fold speedup
    - SuperLU_DIST : distributed-memory parallel machines
      - Achieved up to 100 fold speedup
  - Support real and complex matrices, fill-reducing orderings, equilibration, numerical pivoting, condition estimation, iterative refinement, and error bounds.
- Enabled Scientific Discovery
  - First solution to quantum scattering of 3 charged particles. [Recigno, Baertschy, Isaacs & McCurdy, Science, 24 Dec 1999]
  - SuperLU solved complex unsymmetric systems of order up to 1.79 million, on the ASCI Blue Pacific Computer at LLNL.

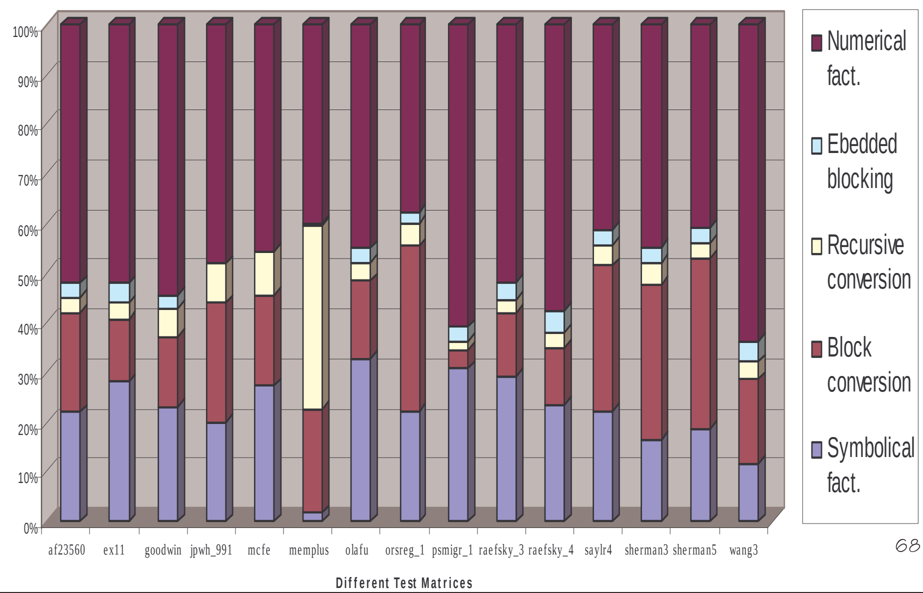


66

Comparison with SuperLU on Pentium III

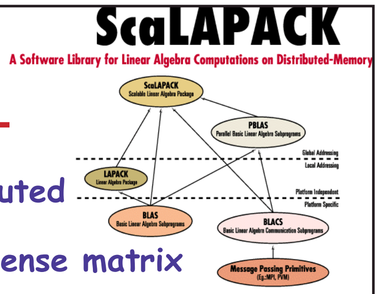
			SuperLU			Recursion		
Name	N	nonzeros	Time[s]	FERR	L+U	Time[s]	FERR	L+U
af23560	23560	460598	44.19	5.80E-14	132.2	31.34	1.80E-14	149.7
ex11	16614	1096948	109.67	2.50E-05	210.2	55.3	1.30E-06	150.6
goodwin	7320	324772	6.49	1.20E-08	31.3	6.74	4.60E-06	35
jpwh_991	991	6027	0.19	2.90E-15	1.4	0.25	2.60E-15	2.3
mcfe	765	24382	0.07	1.20E-13	0.9	0.22	9.10E-13	1.8
memplus	17758	126150	0.29	2.10E-12	5.9	12.67	6.60E-13	195.7
olafu	16146	1015156	26.16	1.10E-06	83.9	22.1	3.70E-09	96.1
orsreg_1	2205	14133	0.46	1.30E-13	3.6	0.45	2.10E-13	3.9
psmigr_1	3140	543162	110.79	7.90E-11	64.6	88.61	1.20E-05	78.4
raefsky3	21200	1488768	62.07	1.40E-09	147.2	69.67	4.40E-13	193.9
raefsky4	19779	1316789	82.45	2.30E-06	156.2	104.29	3.50E-06	234.4
saylr4	3564	22316	0.85	3.20E-11	6	0.95	1.20E-11	7.2
sherman3	5005	20033	0.61	6.00E-13	5	0.67	4.80E-13	7.3
sherman5	3312	20793	0.28	1.40E-13	3	0.32	6.20E-15	3.1
wang3	26064	177168	84.14	2.40E-14	116.7	79.18	1.60E-14	256.7

Breakdown of Time Across Phases For the Recursive Sparse Factorization



68

ScaLAPACK



- ScaLAPACK is a portable distributed memory numerical library
- Complete numerical library for dense matrix computations
- Designed for distributed parallel computing (MPP & Clusters) using MPI
- One of the first math software packages to do this
- Numerical software that will work on a heterogeneous platform
- Funding from DOE, NSF, and DARPA
- In use today by IBM, HP-Convex, Fujitsu, NEC, Sun, SGI, Cray, NAG, IMSL, …
  - Tailor performance & provide support

69

ScaLAPACK

- Library of software dealing with dense & banded routines
- Distributed Memory - Message Passing
- MIMD Computers and Networks of Workstations
- Clusters of SMPs

70

Programming Style

- SPMD Fortran 77 with object based design
- Built on various modules
  - PBLAS Interprocessor communication
  - BLACS
    - PVM, MPI, IBM SP, CRI T3, Intel, TMC
    - Provides right level of notation.
  - BLAS
- LAPACK software expertise/quality
  - Software approach
  - Numerical methods

71

Overall Structure of Software

- Object based - Array descriptor
  - Contains information required to establish mapping between a global array entry and its corresponding process and memory location.
  - Provides a flexible framework to easily specify additional data distributions or matrix types.
  - Currently dense, banded, & out-of-core
- Using the concept of context

72

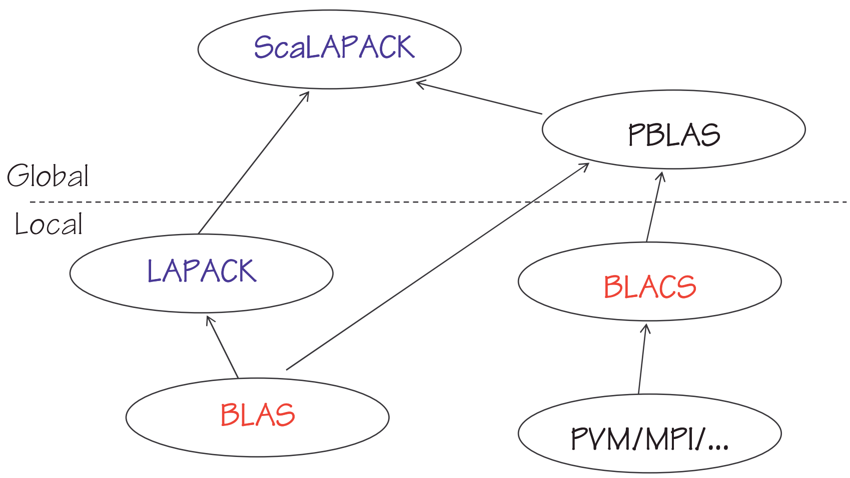
PBLAS

- Similar to the BLAS in functionality and naming.
- Built on the BLAS and BLACS
- Provide global view of matrix

```
CALL DGEXXX  ( M, N, A( IA, JA ), LDA,... )
        ⇩
CALL PDGEXXX( M, N, A, IA, JA, DESCA,... )
```

73

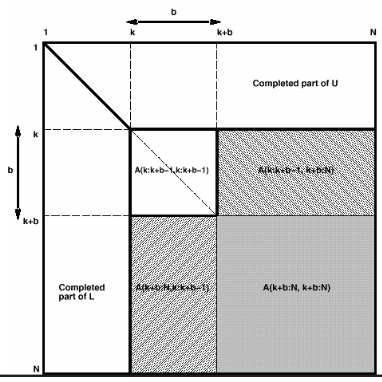
ScaLAPACK Structure



74

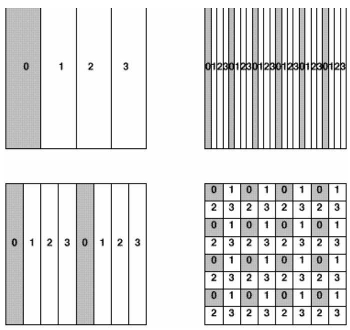
Choosing a Data Distribution

- **Main issues are:**
  - Load balancing
  - Use of the Level 3 BLAS



Possible Data Layouts

- 1D block and cyclic column distributions

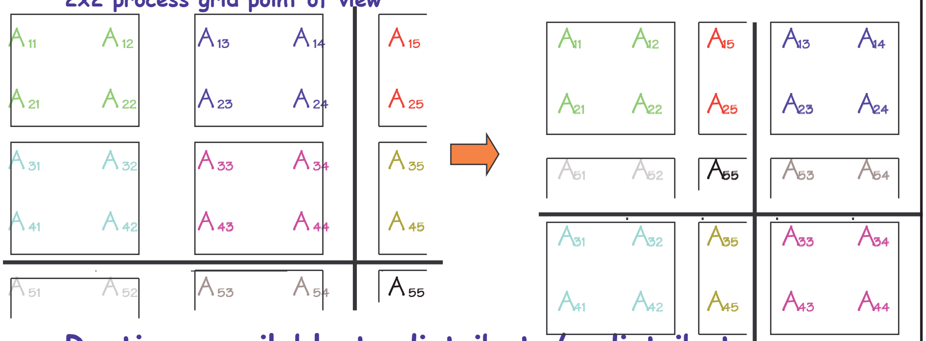


- 1D block-cycle column and 2D block-cyclic distribution
- 2D block-cyclic used in ScaLAPACK for dense matrices

Distribution and Storage

- Matrix is block-partitioned & maps blocks
- Distributed 2-D block-cyclic scheme

5x5 matrix partitioned in 2x2 blocks
2x2 process grid point of view



- Routines available to distribute/redistribute data.

To Use ScaLAPACK a User Must:

- Download the package and auxiliary packages (like PBLAS, BLAS, BLACS, & MPI) to the machines.
- Write a SPMD program which
  - Sets up the logical 2-D process grid
  - Places the data on the logical process grid
  - Calls the numerical library routine in a SPMD fashion
  - Collects the solution after the library routine finishes
- The user must allocate the processors and decide the number of processes the application will run on
- The user must start the application
  - "mpirun –np *N* user_app"
    - Note: the number of processors is fixed by the user before the run, if problem size changes dynamically …
- Upon completion, return the processors to the pool of resources

ScaLAPACK Cluster Enabled

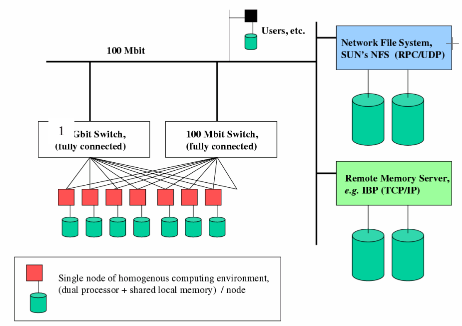
- Implement a version of a ScaLAPACK library routine that runs on clusters.
  - Make use of resources at the user's disposal
  - Provide the best time to solution
  - Proceed without the user's involvement
- Make as few changes as possible to the numerical software.

79

LAPACK For Clusters

- Developing middleware which couples cluster system information with the specifics of a user problem to launch cluster based applications on the "best" set of resource available.

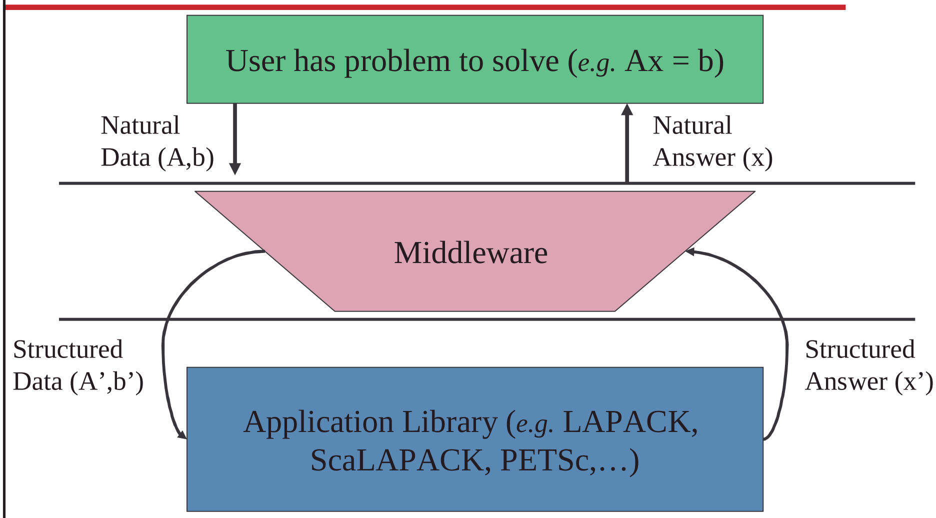
Sample computing environment...



80

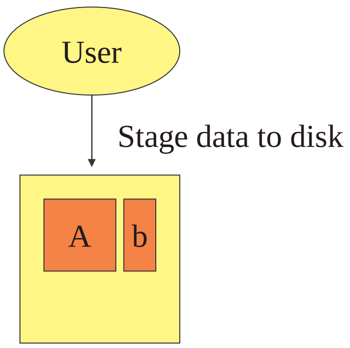
- Using ScaLAPACK as the prototype software

Big Picture…



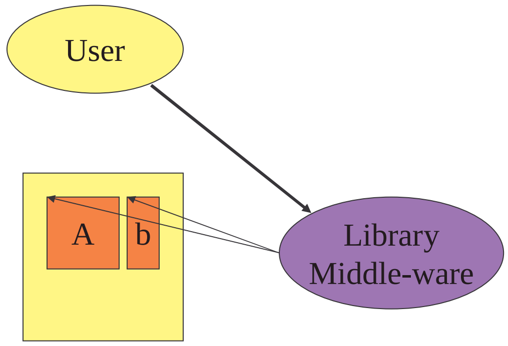
81

Numerical Libraries for Clusters

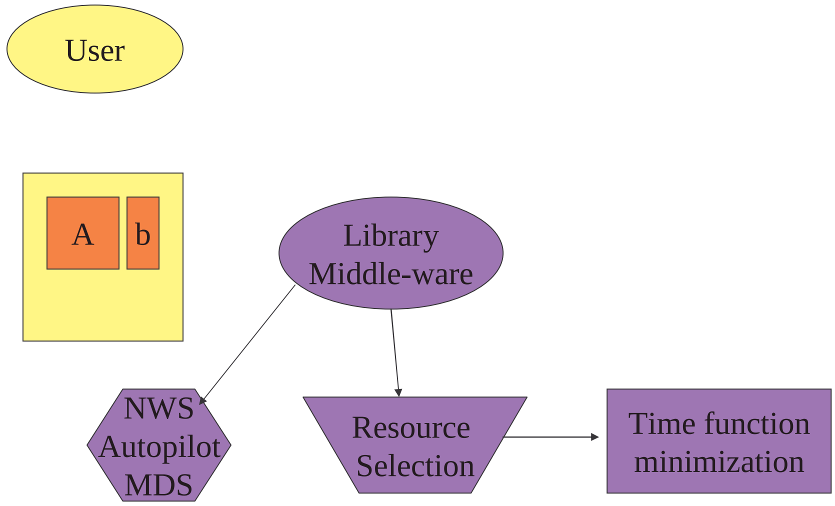


82

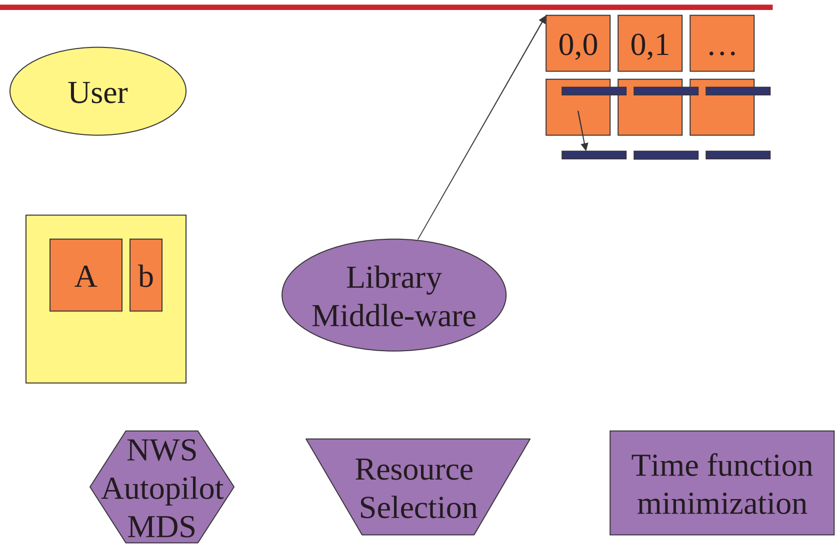
Numerical Libraries for Clusters



Numerical Libraries for Clusters



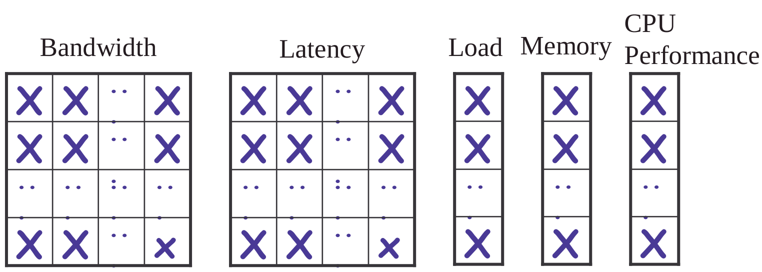
Numerical Libraries for Clusters

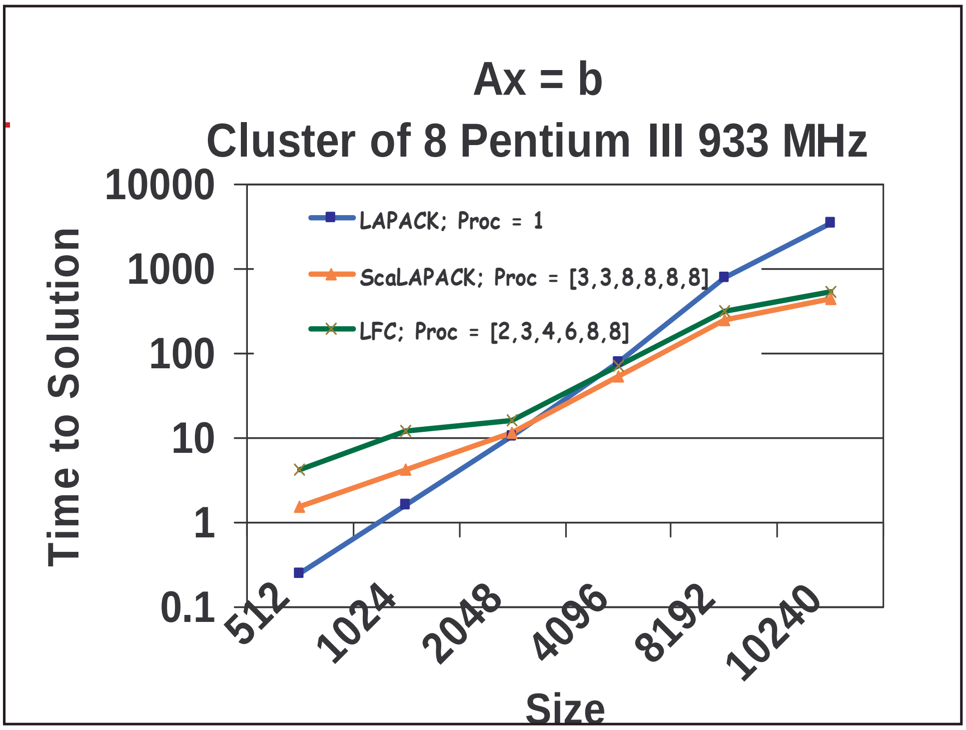


Uses Grid infrastructure, i.e.Globus/NWS, but doesn't have to.

Resource Selector

- Use information on Bandwidth/Latency/Load/Memory/CPU performance
  - 2 matrices (bw,lat) 3 arrays (load, cpu, memory available)
- Generated dynamically by library routine





LAPACK For Clusters (LFC)

- LFC will automate much of the decisions in the Cluster environment to provide best time to solution.
  - Adaptivity to the dynamic environment.
  - As the complexities of the Clusters and Grid increase need to develop strategies for self adaptability.
  - Handcrafted developed leading to an automated design.
- Developing a basic infrastructure for computational science applications and software in the Cluster and Grid environment.
  - Lack of tools is hampering development today.
- Plan to do suite: LU, Cholesky, QR, Symmetric eigenvalue, and Nonsymmetric eigenvalue
- Model for more general framework

FT-MPI

- Current MPI applications live under the MPI fault tolerant model of no faults allowed.
  - This is great on an MPP as if you lose a node you generally lose a partition/job anyway.
  - Makes reasoning about results easy. If there was a fault you might have received incomplete/incorrect values and hence have the wrong result anyway.
  - Planning a version of MPI with some extension which will all the user to recover from system errors, take corrective action, and carry one.
  - Plan to be finished by the end of summer with the beta release.

89

Fault Tolerance in the Message Passing

- Critical for many Grid and Cluster applications
- MPI wasn't designed to be fault tolerant
- Number of projects
  - FT-MPI at University of Tennessee

90

Algorithmic Fault Tolerance

- Important that this is built into the algorithms.
- Not good enough to have it in the message passing.
- Alpha version
  - Limited number of MPI functions supported
- Currently working on getting PETSc (The Portable, Extensible Toolkit for Scientific Computation from ANL) working in a FT mode
  - Target of 86 functions by end of summer 2002.
  - Covers all major classes of functions in MPI.
- Future work
  - Templates for different classes of MPI applications so users can build on our work
  - Some MPI-2 support (PIO?)
- Working on numerical library design for ScaLAPACK and PETSc that will be fault tolerant.

91

Fault Tolerance - Diskless (RAID) Checkpointing - Built into Software (J. Plank, J. Dongarra)

- Maintain a system checkpoint in memory
  - All processors may be roll back if necessary
  - Use m extra processors to encode checkpoints so that if up to m processors fail, their checkpoints may be restored
  - No reliance on disk
- Checksum and reverse communication
  - Checkpoint less frequently
  - Reverse the computation of the non-failed processors back to previous checkpoint
- Idea to build into library routines
  - System or user can dial it up
  - Working prototype for MM, LU, LL^T, QR, sparse solvers

92

Use Diskless Checkpointing (PL94b):

- The N application processors each maintain their own checkpoints locally.

- m extra processors maintain coding information so that if 1 or more processors die, they can be replaced.

- Will describe for m = 1 (parity)

93

What "Algorithm-based" means

Algorithm-based == non-transparent

Reasons against transparency:

- No synchronization worries
- Minimize checkpoint state
- Heterogeny

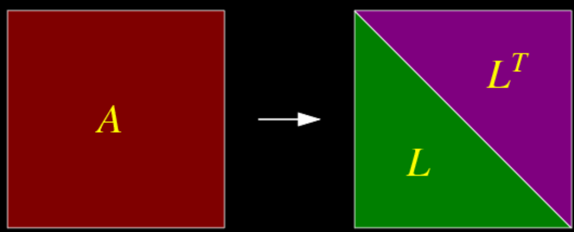
94

Cholesky Factorization

Factor a dense, symmetric, positive definite matrix A into two matrices:

$$A = LL^T$$

This is done in place:

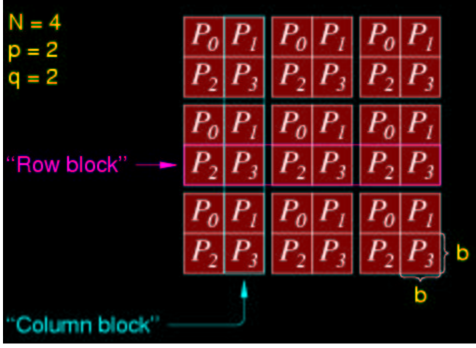


Before After

95

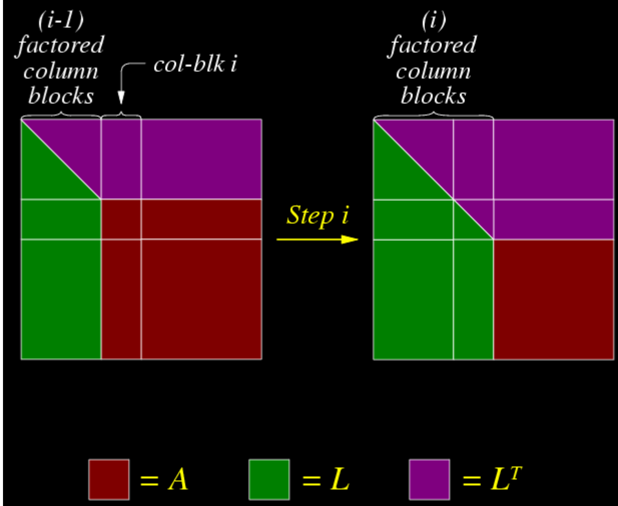
Blocking the Matrix

- The matrix is partitioned into square blocks of a specified block size b
- The processors are (logically) configured into a p by q mesh, and the blocks are doled among the processors in panels of p*q blocks.



96

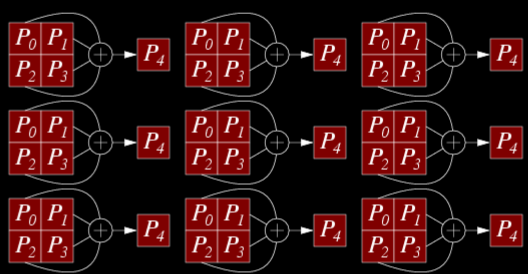
Top-looking Cholesky Factorization



97

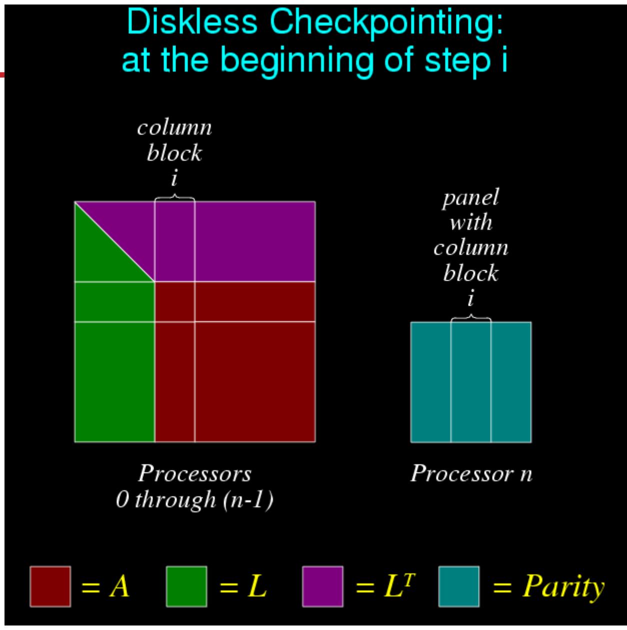
Diskless Checkpointing: Starting State

For each panel of the matrix, maintain a block in the checkpointing processor that holds the bitwise parity of all blocks in that panel

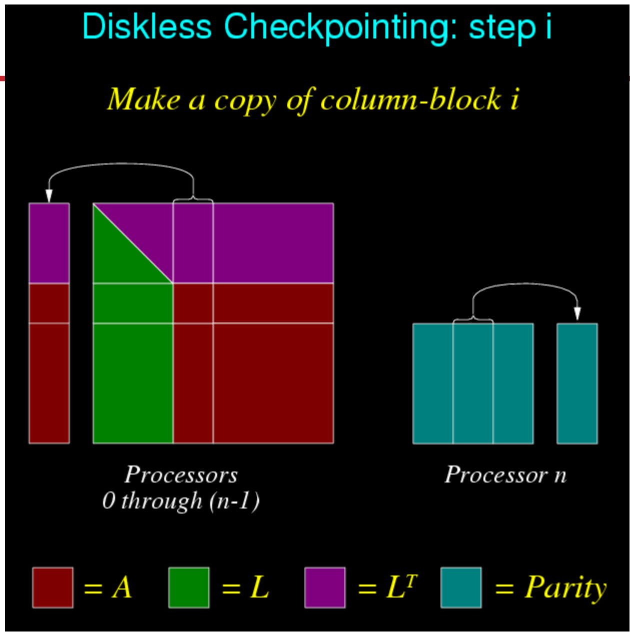


If a single processor fails, then its state may be restored from the remaining live ones

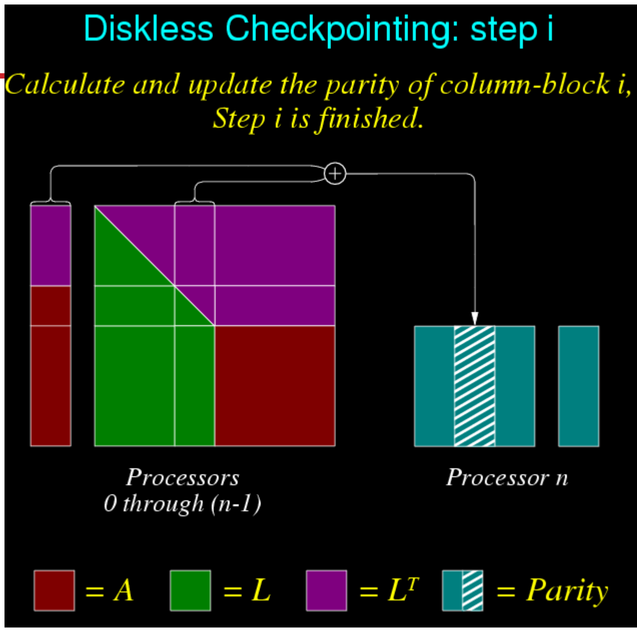
98


Diskless Checkpointing:
at the beginning of step i
column
block
i
panel
with
column
block
i
Processors
0 through (n-1)
Processor n
= A
= L
= L^T
= Parity

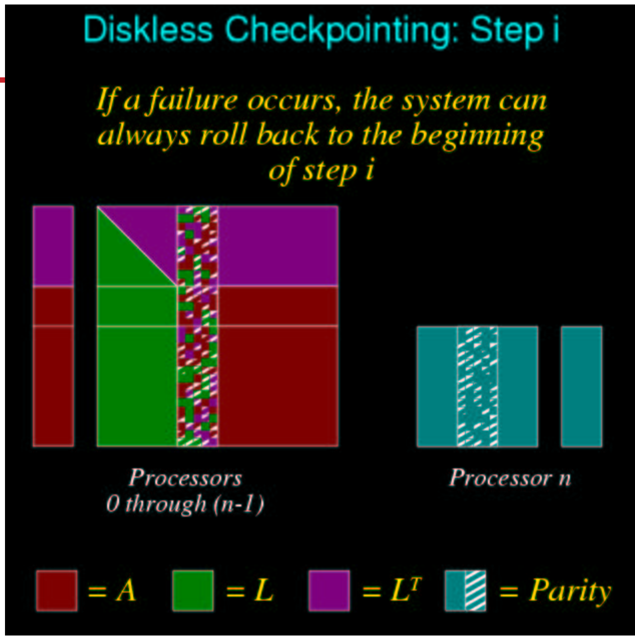
99


Diskless Checkpointing: step i
Make a copy of column-block i
Processors
0 through (n-1)
Processor n
= A
= L
= L^T
= Parity

100


Diskless Checkpointing: step i
Calculate and update the parity of column-block i,
Step i is finished.
Processors
0 through (n-1)
Processor n
= A
= L
= L^T
= Parity

101


Diskless Checkpointing: Step i
If a failure occurs, the system can
always roll back to the beginning
of step i
Processors
0 through (n-1)
Processor n
= A
= L
= L^T
= Parity

102

Tools for Performance Evaluation

- Timing and performance evaluation has been an art
  - Resolution of the clock
  - Issues about cache effects
  - Different systems
  - Can be cumbersome and inefficient with traditional tools
- Situation about to change
  - Today's processors have internal counters

103

Performance Counters

- Almost all high performance processors include hardware performance counters.
- Some are easy to access, others not available to users.
- On most platforms the APIs, if they exist, are not appropriate for the end user or well documented.
- Existing performance counter APIs
  - Compaq Alpha EV 6 & 6/7
  - SGI MIPS R10000
  - IBM Power Series
  - CRAY T3E
  - Sun Solaris
  - Pentium Linux and Windows
  - IA-64
  - HP-PA RISC
  - Hitachi
  - Fujitsu
  - NEC

Performance Data That May Be Available

- Cycle count
- Floating point instruction count
- Integer instruction count
- Instruction count
- Load/store count
- Branch taken / not taken count
- Branch mispredictions
- Pipeline stalls due to memory subsystem
- Pipeline stalls due to resource conflicts
- I/D cache misses for different levels
- Cache invalidations
- TLB misses
- TLB invalidations

105

Low Level API

- Increased efficiency and functionality over the high level PAPI interface
- There's about 40 functions
- Obtain information about the executable and the hardware.
- Thread safe

106

High Level API

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Not thread safe at the moment
- Only allows PAPI Presets events

107

High Level Functions

- PAPI_flops()
- PAPI_num_counters()
  - Number of counters in the system
- PAPI_start_counters()
- PAPI_stop_counters()
  - Enable counting of events and describes what to count
- PAPI_read_counters()
  - Returns event counts

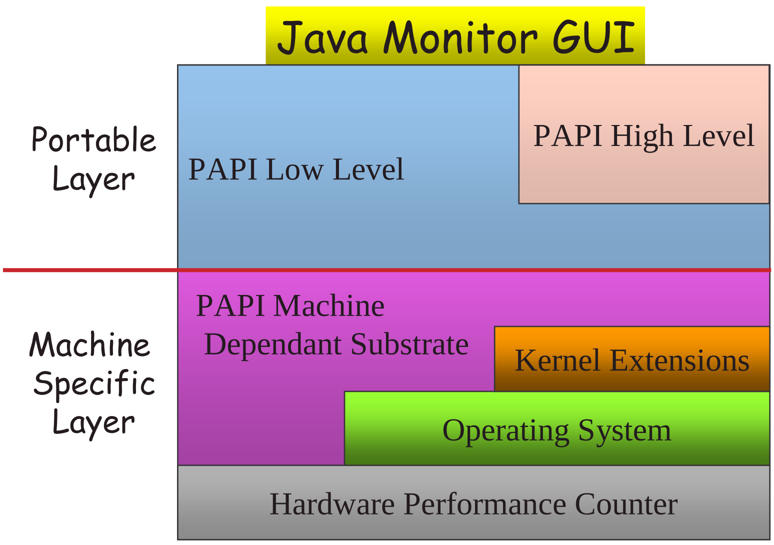
108

Perfometer Features

- Platform independent visualization of PAPI metrics
- Flexible interface
- Quick interpretation of complex results
- Small footprint
  - (compiled code size < 15k)
- Color coding to highlight selected procedures
- Trace file generation or real time viewing.

109

PAPI Implementation



110

PAPI - Supported Processors

- Intel Pentium,II,III, 4, Itanium,
  - Linux 2.4, 2.2, 2.0 and perf kernel patch
- IBM Power 3,604,604e (Power 4 coming)
  - For AIX 4.3 and pmtoolkit (in 4.3.4 available)
  - (laderose@us.ibm.com)
- Sun UltraSparc I, II, & III
  - Solaris 2.8
- SGI IRIX/MIPS
- AMD Athlon
  - Linux 2.4 and perf kernel patch
- Cray T3E, SV1, SV2
- Windows 2K and XP
- To download software see:
  http://icl.cs.utk.edu/papi/

Work in progress on Compaq Alpha
Fortran, C, and MATLAB bindings

111

Early Users of PAPI

- DEEP/PAPI (Pacific Sierra)
  http://www.psrv.com/deep_papi_top.html
- TAU (Allen Mallony, U of Oregon)
  http://www.cs.uoregon.edu/research/paracomp/tau/
- SvPablo (Dan Reed, U of Illinois)
  http://vibes.cs.uiuc.edu/Software/SvPablo/svPablo.htm
- Cactus (Ed Seidel, Max Plank/U of Illinois)
  http://www.aei-potsdam.mpg.de
- Vprof (Curtis Janssen, Sandia Livermore Lab)
  http://aros.ca.sandia.gov/~cljanss/perf/vprof/
- Cluster Tools (Al Geist, ORNL)
- DynaProf

112

What is DynaProf?

- *A* portable tool to dynamically instrument a running executable with *Probes* that monitor application performance.
- Simple command line interface.
- Java based GUI interface.
- Open Source Software.
- Built on and in collaboration with Bart Miller and Jeff Hollingsworth Paradyn project at U. Wisconsin and Dyninst project at U. Maryland

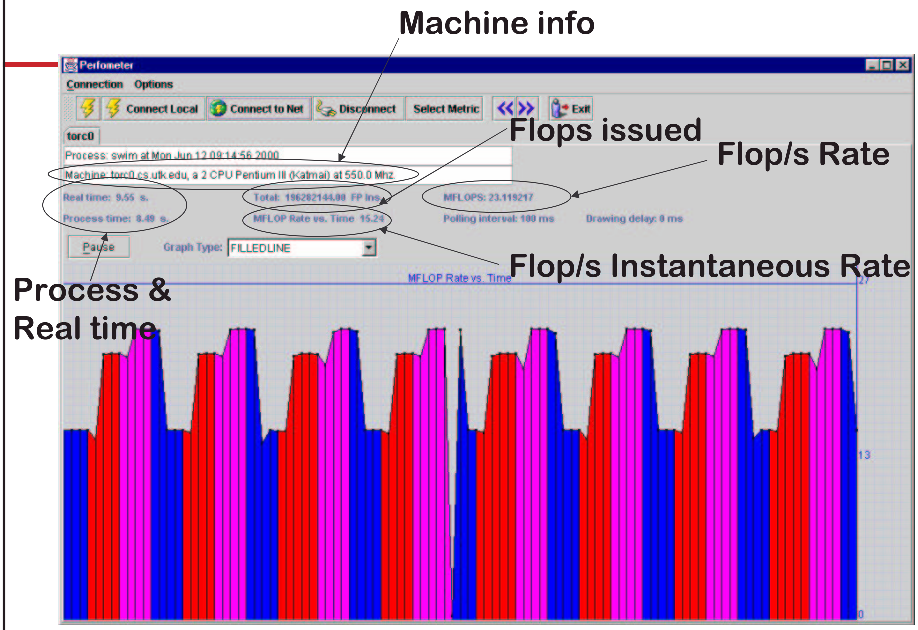
113

Dynamic Instrumentation:

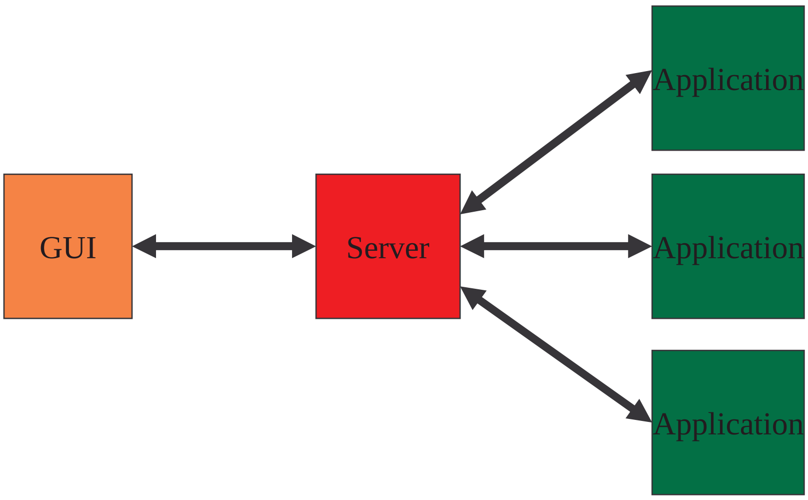
- Operates on a running executable.
- Identifies instrumentation *points* where code can be inserted.
- Inserts code *snippets* at selected *points*.
- *Snippets* can collect and monitor performance information.
- *Snippets* can be removed and reinserted dynamically.
- Source code not required, just executable

114

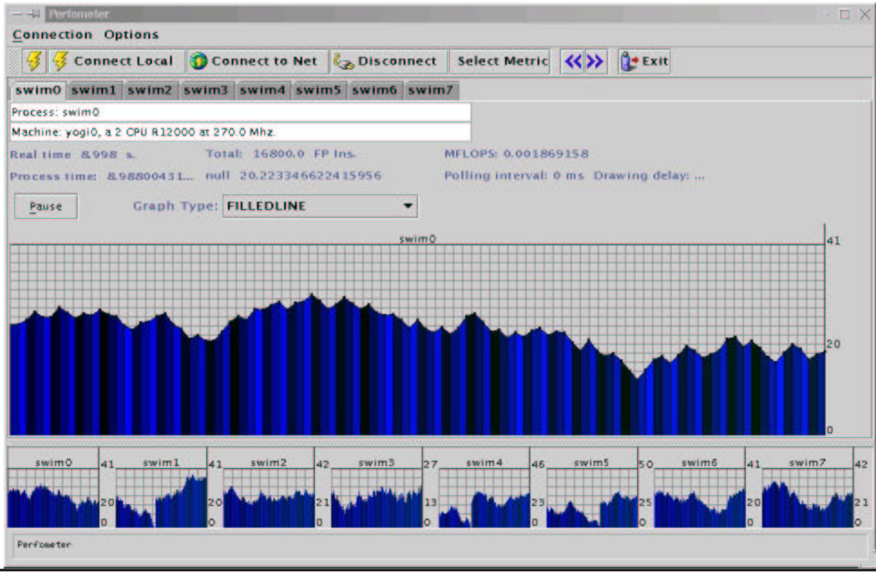
Perfometer/ DynaProf



Next Version of Perfometer Implementation



PAPI's Parallel Interface



117

Futures for Numerical Algorithms and Software on Clusters and Grids

- Retargetable Libraries - Numerical software will be adaptive, exploratory, and intelligent
- Determinism in numerical computing will be gone.
  - After all, its not reasonable to ask for exactness in numerical computations.
  - Auditability of the computation, reproducibility at a cost
- Importance of floating point arithmetic will be undiminished.
  - 16, 32, 64, 128 bits and beyond.
- Reproducibility, fault tolerance, and auditability
- Adaptivity is a key so applications can effectively use the resources.

118

Contributors to These Ideas

- Top500
  - Erich Strohmaier, LBL
  - Hans Meuer, Mannheim U
- ATLAS
  - Antoine Petitet, UTK
  - Clint Whaley, UTK
- Recursive factorization
  - Piotr Luszczek,UTK
  - Victor Eijkhout, UTK
- PAPI
  - Shirley Browne, UTK
  - Kevin London, UTK
  - Phil Mucci, UTK
  - Keith Seymour, UTK
  - Dan Terpstra, UTK

For additional information see…

www.netlib.org/top500/
icl.cs.utk.edu/atlas/
icl.cs.utk.edu/papi/
www.cs.utk.edu/~dongarra/

Many opportunities within the group at Tennessee

119