Question:

- **Suppose we want to compute using four decimal arithmetic:**
  - $S = 1.000 + 1.000 \times 10^4 - 1.000 \times 10^4$
  - What's the answer?

- **Ariane 5 rocket**
  - June 1996 exploded when a 64 bit fl pt number relating to the horizontal velocity of the rocket was converted to a 16 bit signed integer. The number was larger than 32,767, and thus the conversion failed.
  - $500M rocket and cargo
Two sources of numerical error

1) Round off error
2) Truncation error

Round off Error

- Caused by representing a number approximately

\[
\frac{1}{3} \equiv 0.333333
\]

\[
\sqrt{2} \equiv 1.4142...
\]
Problems created by round off error

- 28 Americans were killed on February 25, 1991 by an Iraqi Scud missile in Dhahran, Saudi Arabia.
- The patriot defense system failed to track and intercept the Scud. Why?

Problem with Patriot missile

- Clock cycle of 1/10 seconds was represented in 24-bit fixed point register created an error of $9.5 \times 10^{-8}$ seconds.

  \[
  0.1 \text{ (decimal)} = 00111101110011001100110011001101 \text{ (binary)}
  \]

- The battery was on for 100 consecutive hours, thus causing an inaccuracy of

  \[
  \frac{9.5 \times 10^{-8} \times 8 \text{ s}}{0.1 \text{ s}} \times 100 \text{ hr} \times \frac{3600 \text{ s}}{1 \text{ hr}} = 0.342 \text{ s}
  \]
Problem (cont.)

- The shift calculated in the ranging system of the missile was 687 meters.
- The target was considered to be out of range at a distance greater than 137 meters.

Truncation error

- Error caused by truncating or approximating a mathematical procedure.
Example of Truncation Error

Taking only a few terms of a Maclaurin series to approximate $e^x$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

If only 3 terms are used, 

$$Truncation \ Error = e^x - \left(1 + x + \frac{x^2}{2!}\right)$$

Defining Floating Point Arithmetic

♦ Representable numbers
  ➢ Scientific notation: $\pm \cdot d.d \ldots d \times r^{exp}$
  ➢ sign bit $\pm$/
  ➢ radix $r$ (usually 2 or 10, sometimes 16)
  ➢ significand $d.d \ldots d$ (how many base-$r$ digits $d$?)
  ➢ exponent $exp$ (range?)
  ➢ others?

♦ Operations:
  ➢ arithmetic: $+, -, \times, \div, \ldots$
    ➢ how to round result to fit in format
  ➢ comparison ($+, =, >$)
  ➢ conversion between different formats
    ➢ short to long FP numbers, FP to integer
  ➢ exception handling
    ➢ what to do for $0/0, 2^{\text{largest\_number}}$, etc.
  ➢ binary/decimal conversion
    ➢ for I/O, when radix not 10
IEEE Floating Point Arithmetic Standard 754 (1985) - Normalized Numbers

- Normalized Nonzero Representable Numbers: \( \pm 1.d...d \times 2^{\text{exp}} \)
  - Macheps = Machine epsilon = \( 2^{-\#\text{significand bits}} \) = relative error in each operation
  - smallest number \( e \) such that \( f(1 + e) > 1 \)
  - OV = overflow threshold = largest number
  - UN = underflow threshold = smallest number

<table>
<thead>
<tr>
<th>Format</th>
<th># bits</th>
<th>#significand bits</th>
<th>macheps</th>
<th>#exponent bits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>32</td>
<td>23+1</td>
<td>( 2^{-24} ) (( \approx 10^{-7} ))</td>
<td>8</td>
<td>( 2^{-126} - 2^{127} ) (( \approx 10^{-38} ))</td>
</tr>
<tr>
<td>Double</td>
<td>64</td>
<td>52+1</td>
<td>( 2^{-53} ) (( \approx 10^{-16} ))</td>
<td>11</td>
<td>( 2^{-1022} - 2^{1023} ) (( \approx 10^{-308} ))</td>
</tr>
<tr>
<td>Double &gt;=80</td>
<td>&gt;=64</td>
<td>&lt;=64</td>
<td>(&lt;2^{53}(\approx10^{-19}))</td>
<td>&gt;=15</td>
<td>(2^{16382} - 2^{16383} ) (( \approx 10^{-4932} ))</td>
</tr>
</tbody>
</table>

Extended (80 bits on all Intel machines)

- +- Zero: \( \pm 0 \), significand and exponent all zero
  - Why bother with -0 later

IEEE Floating Point Arithmetic Standard 754 - "Denoms"

- Denormalized Numbers: \( \pm 0.d...d \times 2^{\text{min.exp}} \)
  - sign bit, nonzero significand, minimum exponent
  - Fills in gap between UN and 0

- Underflow Exception
  - occurs when exact nonzero result is less than underflow threshold UN
  - Ex: UN/3
  - return a denorm, or zero
IEEE Floating Point Arithmetic
Standard 754 - +- Infinity

◆ +- Infinity: Sign bit, zero significand, maximum exponent
◆ Overflow Exception
  ➢ occurs when exact finite result too large to represent accurately
  ➢ Ex: 2*OV
  ➢ return +- infinity
◆ Divide by zero Exception
  ➢ return +- infinity = 1/+-0
  ➢ sign of zero important!
◆ Also return +- infinity for
  ➢ 3+infinity, 2*infinity, infinity*infinity
  ➢ Result is exact, not an exception!

IEEE Floating Point Arithmetic
Standard 754 - NAN (Not A Number)

◆ NAN: Sign bit, nonzero significand, maximum exponent
◆ Invalid Exception
  ➢ occurs when exact result not a well-defined real number
  ➢ 0/0
  ➢ sqrt(-1)
  ➢ infinity-infinity, infinity/infinity, 0*infinity
  ➢ NAN + 3
  ➢ NAN > 3?
  ➢ Return a NAN in all these cases
◆ Two kinds of NANs
  ➢ Quiet - propagates without raising an exception
  ➢ Signaling - generate an exception when touched
    » good for detecting uninitialized data
Error Analysis

- **Basic error formula**
  - \( \text{fl}(a \text{ op } b) = (a \text{ op } b) \times (1 + d) \) where
    - \( \text{op one of } +, -, \times, / \)
    - \( |d| \leq \text{macheps} \)
    - assuming no overflow, underflow, or divide by zero

- **Example: adding 4 numbers**
  - \( \text{fl}(x_1 + x_2 + x_3 + x_4) = \left( (x_1 + x_2) \times (1 + d_1) + x_3 \times (1 + d_2) + x_4 \times (1 + d_3) \right) \times (1 + d_4) \)
  - where each \(|e_i| \leq 3 \times \text{macheps} \)

- Get exact sum of slightly changed summands \( x_i \times (1 + e_i) \)

- **Backward Error Analysis** - algorithm called numerically stable if it gives the exact result for slightly changed inputs

- Numerical Stability is an algorithm design goal

Backward error

- **Approximate solution is exact solution to modified problem.**
- **How large a modification to original problem is required to give result actually obtained?**
- **How much data error in initial input would be required to explain all the error in computed results?**
- **Approximate solution is good if it is exact solution to "nearby" problem.**

![Backward error diagram](image)
Sensitivity and Conditioning

- **Problem is insensitive or well conditioned** if relative change in input causes commensurate relative change in solution.
- **Problem is sensitive or ill-conditioned**, if relative change in solution can be much larger than that in input data.

\[
\text{Cond} = \frac{\text{Relative change in solution}}{\text{Relative change in input data}} = \frac{|f(x') - f(x)|/f(x)}{|x' - x|/x}
\]

- Problem is sensitive, or ill-conditioned, if cond $>> 1$.
- When function $f$ is evaluated for approximate input $x' = x + h$ instead of true input value of $x$.
- Absolute error $= f(x + h) - f(x) \approx h f'(x)$
- Relative error $= \frac{f(x + h) - f(x)}{f(x)} \approx h f'(x) / f(x)$

Sensitivity: 2 Examples

**cos(\pi/2)** and 2-d System of Equations

- Consider problem of computing cosine function for arguments near \(\pi/2\).
- Let $x \approx \pi/2$ and let $h$ be small perturbation to $x$. Then

\[
\begin{align*}
\text{Abs: } f(x + h) - f(x) &\approx h f'(x) \\
\text{Rel: } \frac{f(x + h) - f(x)}{f(x)} &\approx h f'(x) / f(x)
\end{align*}
\]

absolute error $= \cos(x+h) - \cos(x) \approx -h \sin(x) \approx -h$,
relative error $\approx -h \tan(x) \approx \infty$

- So small change in $x$ near $\pi/2$ causes large relative change in $\cos(x)$ regardless of method used.
- $\cos(1.57079) = 0.63267949 \times 10^{-5}$
- $\cos(1.57078) = 1.64267949 \times 10^{-5}$
- Relative change in output is a quarter million times greater than relative change in input.
Sensitivity: 2 Examples

\[ \cos(\pi/2) \] and 2-d System of Equations

- Consider problem of computing cosine function for arguments near \( \pi/2 \).
- Let \( x \approx \pi/2 \) and let \( h \) be small perturbation to \( x \). Then

\[
\text{absolute error} = \cos(x+h) - \cos(x) \\
\approx -h \sin(x) \approx -h,
\]

\[
\text{relative error} \approx -h \tan(x) \approx \infty
\]

- So small change in \( x \) near \( \pi/2 \) causes large relative change in \( \cos(x) \) regardless of method used.
- \( \cos(1.57079) = 0.63267949 \times 10^{-5} \)
- \( \cos(1.57078) = 1.64267949 \times 10^{-5} \)
- Relative change in output is a quarter million times greater than relative change in input.

Exception Handling

- What happens when the “exact value” is not a real number, or too small or too large to represent accurately?
- 5 Exceptions:
  - Overflow - exact result > OV, too large to represent
  - Underflow - exact result nonzero and < UN, too small to represent
  - Divide-by-zero - nonzero/0
  - Invalid - 0/0, sqrt(-1), ...
  - Inexact - you made a rounding error (very common!)

- Possible responses
  - Stop with error message (unfriendly, not default)
  - Keep computing (default, but how?)
Summary of Values
Representable in IEEE FP

- **+ - Zero**
- **Normalized nonzero numbers**
- **Denormalized numbers**
- **+ - Infinity**
- **NANs**
  - Signaling and quiet
  - Many systems have only quiet

More on the In-Class Presentations

- Start at 10:00 on Monday, 5/4/09
- Present your report to class (~20 mins)
- Use powerpoint or pdf
- Load on laptop in room
- Describe your project, perhaps motivate via application
- Describe your method/approach
- Provide comparison and results
- See me about your topic
- Topic of general interest to the course.
- The idea is to read three or four papers from the literature (references will be provided)
- Implement the application on the cluster you build
- Synthesize them in terms of a report (~20 pages)
- Turn in reports (email them) on Monday, 5/4/09 before class
- New ideas and extensions are welcome, as well as implementation prototype if needed.
- Go to class webpage for ideas
Adaptive Approach for Level 3

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

Optimizing in practice

- Tiling for registers
  - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache
- Exploiting fine-grained parallelism within the processor
  - super scalar
  - pipelining
- Complicated compiler interactions
- Hard to do by hand (but you’ll try)
- Automatic optimization an active research area
- PHIPAC: www.icsi.berkeley.edu/~bilmes/phipac
- www.cs.berkeley.edu/~iyer/asci_slides.ps
- ATLAS: www.netlib.org/atlas/index.html
ATLAS

- Keep a repository of kernels for specific machines.
- Develop a means of dynamically downloading code
- Extend work to allow sparse matrix operations
- Extend work to include arbitrary code segments
- See: http://www.netlib.org/atlas/

Implementation

**Installation:** self-tuning

- Code generation (C)
- Hand Code (assembly)

**Timing**

- Algorithm selection

**Runtime:** decision based on data

- Small, contiguous data
  - No-copy code
- Otherwise
  - Code with data copy

Performance does not depend on data, unless:
- Special numerical properties exist
- Diagonal dominance for LU factorization (5%-10% speed improvement)
- NaNs, INFs in the vector/matrix
Code Generation Strategy

- On-chip multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory fetch
  - Register reuse
  - Loop overhead minimization
- Takes a couple of hours to run.

- Code is iteratively generated & timed until optimal case is found. We try:
  - Differing NBs
  - Breaking false dependencies
  - M, N and K loop unrolling

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
Gaussian Elimination Basics

Solve $Ax = b$

**Step 1**

$A = LU$

**Step 2**  Forward Elimination

Solve $Ly = b$

**Step 3**  Backward Substitution

Solve $Ux = y$

Note: Changing RHS does not imply to recompute LU factorization

---

Gaussian Elimination

Standard Way

- Subtract a multiple of a row

Overwrite $A$ with $L$ and $U$

The lower part of $A$ has a representation of “L”
Gaussian Elimination (GE) for Solving $Ax=b$

- Add multiples of each row to later rows to make $A$ upper triangular
- Solve resulting triangular system $Ux = c$ by substitution

... for each column $i$
... zero it out below the diagonal by adding multiples of row $i$ to later rows
for $i = 1$ to $n-1$
... for each row $j$ below row $i$
for $j = i+1$ to $n$
... add a multiple of row $i$ to row $j$
\[
\text{tmp} = A(j,i);
\text{for k = i to n}
A(j,k) = A(j,k) - (\text{tmp}/A(i,i)) \times A(i,k)
\]

Refine GE Algorithm (1)

- Initial Version
  ... for each column $i$
  ... zero it out below the diagonal by adding multiples of row $i$ to later rows
for $i = 1$ to $n-1$
... for each row $j$ below row $i$
for $j = i+1$ to $n$
... add a multiple of row $i$ to row $j$
\[
\text{tmp} = A(j,i);
\text{for k = i to n}
A(j,k) = A(j,k) - (\text{tmp}/A(i,i)) \times A(i,k)
\]

- Remove computation of constant $\text{tmp}/A(i,i)$ from inner loop.

\[
\text{for } i = 1 \text{ to } n-1
\text{for } j = i+1 \text{ to } n
m = A(j,i)/A(i,i)
\text{for k = i to n}
A(j,k) = A(j,k) - m \times A(i,k)
\]
Refine GE Algorithm (2)

- Last version

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n-1 \\
\text{for } j &= i+1 \text{ to } n \\
m &= A(j,i)/A(i,i) \\
\text{for } k &= i \text{ to } n \\
A(j,k) &= A(j,k) - m \times A(i,k)
\end{align*}
\]

- Don’t compute what we already know: zeros below diagonal in column i

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n-1 \\
\text{for } j &= i+1 \text{ to } n \\
m &= A(j,i)/A(i,i) \\
\text{for } k &= i \text{ to } n \\
A(j,k) &= A(j,k) - m \times A(i,k)
\end{align*}
\]

- Do not compute zeros

Refine GE Algorithm (3)

- Last version

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n-1 \\
\text{for } j &= i+1 \text{ to } n \\
m &= A(j,i)/A(i,i) \\
\text{for } k &= i+1 \text{ to } n \\
A(j,k) &= A(j,k) - m \times A(i,k)
\end{align*}
\]

- Store multipliers m below diagonal in zeroed entries for later use

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n-1 \\
\text{for } j &= i+1 \text{ to } n \\
A(j,i) &= A(j,i)/A(i,i) \\
\text{for } k &= i+1 \text{ to } n \\
A(j,k) &= A(j,k) - A(j,i) \times A(i,k)
\end{align*}
\]

- Store m here
Refine GE Algorithm (4)

- Last version

```
for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i) / A(i,i)
  for k = i+1 to n
    A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Split Loop

```
for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i) / A(i,i)
  for j = i+1 to n
    for k = i+1 to n
      A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

Store all m's here before updating rest of matrix

---

Refine GE Algorithm (5)

- Last version

```
for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i) / A(i,i)
  for j = i+1 to n
    for k = i+1 to n
      A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Express using matrix operations (BLAS)

```
for i = 1 to n-1
  for j = i+1 to n
    A(j,i+1:n) = A(j,i+1:n) * (1 / A(i,i))
  for j = i+1 to n
    A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) * A(i,i+1:n)
```

Work at step i of Gaussian Elimination

- Finished part of U
- Finished multipliers

---

18
What GE really computes

Call the strictly lower triangular matrix of multipliers M, and let L = I+M

Call the upper triangle of the final matrix U

**Lemma (LU Factorization):** If the above algorithm terminates (does not divide by zero) then A = L*U

Solving A*x=b using GE

- Factorize A = L*U using GE (cost = 2/3 n^3 flops)
- Solve L*y = b for y, using substitution (cost = n^2 flops)
- Solve U*x = y for x, using substitution (cost = n^2 flops)

Thus A*x = (L*U)*x = L*(U*x) = L*y = b as desired

Pivoting in Gaussian Elimination

- A = [ 0 1 ] fails completely because can’t divide by A(1,1)=0
- But solving Ax=b should be easy!

- When diagonal A(i,i) is tiny (not just zero), algorithm may terminate but get completely wrong answer
  - **Numerical instability**
  - Roundoff error is cause

- **Cure:** Pivot (swap rows of A) so A(i,i) large
Gaussian Elimination with Partial Pivoting (GEPP)

• Partial Pivoting: swap rows so that A(i,i) is largest in column

for i = 1 to n-1
    find and record k where |A(k,i)| = max{i <= j <= n} |A(j,i)|  ... i.e. largest entry in rest of column i
    if |A(k,i)| = 0
        exit with a warning that A is singular, or nearly so
    elseif i != k
        swap rows i and k of A
    end if
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)  ... each quotient lies in [-1,1]
    A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) * A(i,i+1:n)

• **Lemma**: This algorithm computes A = P*L*U, where P is a permutation matrix.
• This algorithm is numerically stable in practice
• For details see LAPACK code at http://www.netlib.org/lapack/single/sgetrf2.f

Problems with basic GE algorithm

- What if some A(i,i) is zero? Or very small?
  - Result may not exist, or be “unstable”, so need to pivot
- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)

for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)  ... BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) * A(i,i+1:n)  ... BLAS 2 (rank-1 update)

IBM RS/6000 Power 3 (200 MHz, 800 Mflops Peak)

Level 3 BLAS
Level 2 BLAS
Level 1 BLAS
Converting BLAS2 to BLAS3 in GEPP

- Blocking
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP
- BIG IDEA: Delayed Updates
  - Save updates to “trailing matrix” from several consecutive BLAS2 updates
  - Apply many updates simultaneously in one BLAS3 operation
- BIG IDEA: Delayed Updates
  - Same idea works for much of dense linear algebra
  - Open questions remain
- First Approach: Need to choose a block size \( b \)
  - Algorithm will save and apply \( b \) updates
  - \( b \) must be small enough so that active submatrix consisting of \( b \) columns of \( A \) fits in cache
  - \( b \) must be large enough to make BLAS3 fast

**Blocked GEPP** *(www.netlib.org/lapack/single/sgetrf.f)*

```
for ib = 1 to n-1 step b     ... Process matrix \( b \) columns at a time
  end = ib + b-1              ... Point to end of block of \( b \) columns
  apply BLAS2 version of GEPP to get \( A(ib:n, ib:end) = P' \cdot L' \cdot U' \)
  ... let \( LL \) denote the strict lower triangular part of \( A(ib:end, ib:end) + I \)
  \( A(ib:end, end+1:n) = LL^{-1} \cdot A(ib:end, end+1:n) \) ... update next \( b \) rows of \( U \)
  \( A(end+1:n, end+1:n) = A(end+1:n, end+1:n) \) - \( A(end+1:n, ib:end) \cdot A(ib:end, end+1:n) \)
  ... apply delayed updates with single matrix-multiply
  ... with inner dimension \( b \)
```
Review: BLAS 3 (Blocked) GEPP

for \( ib = 1 \) to \( n-1 \) step \( b \) \( \quad \) ... Process matrix \( b \) columns at a time
  end = \( ib + b - 1 \) \( \quad \) ... Point to end of block of \( b \) columns
  apply BLAS2 version of GEPP to \( A(ib:n, ib:ib) = P^* L^* U^* \)
  ... let \( LL \) denote the strict lower triangular part of \( A(ib:ib, ib:ib) + I \)
  \( A(ib:ib, end+1:n) = LL^{-1} \cdot A(ib:ib, end+1:n) \) \( \quad \) ... update next \( b \) rows of \( U \)
  - \( A(end+1:n, ib:ib) = A(end+1:n, end+1:n) \)
  - apply delayed updates with single matrix-multiply
    ... with inner dimension \( b \)

Gaussian Elimination

- **Standard Way**
  - subtract a multiple of a row

- **LINPACK**
  - apply sequence to a column

- **LAPACK**
  - apply sequence to \( nb \)
  - then apply \( nb \) to rest of matrix
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.

Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or $Q^T C$

- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form

- Block QR iteration for nonsymmetric eigenvalue problems
**ScaLAPACK**

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

### Different Data Layouts for Parallel GE

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) 1D Column Blocked Layout</td>
<td>Bad load balance: P0 idle after first n/4 steps</td>
</tr>
<tr>
<td>2) 1D Column Cyclic Layout</td>
<td>Load balanced, but can't easily use BLAS2 or BLAS3</td>
</tr>
<tr>
<td>3) 1D Column Block Cyclic Layout</td>
<td>Can trade load balance and BLAS2/3 performance by choosing b, but factorization of block column is a bottleneck</td>
</tr>
<tr>
<td>4) Block Skewed Layout</td>
<td>Complicated addressing</td>
</tr>
<tr>
<td>5) 2D Row and Column Blocked Layout</td>
<td>Bad load balance: P0 idle after first n/2 steps</td>
</tr>
<tr>
<td>6) 2D Row and Column Block Cyclic Layout</td>
<td>The winner!</td>
</tr>
</tbody>
</table>

![Diagram](image.png)
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

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
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,... )
  ```

ScalAPACK Overview

ScalAPACK SOFTWARE HIERARCHY
Software/Algorithms follow hardware evolution in time

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINPACK (70's)</strong></td>
</tr>
<tr>
<td>(Vector operations)</td>
</tr>
<tr>
<td>Rely on</td>
</tr>
<tr>
<td>- Level-1 BLAS operations</td>
</tr>
<tr>
<td><strong>LAPACK (80's)</strong></td>
</tr>
<tr>
<td>(Blocking, cache friendly)</td>
</tr>
<tr>
<td>Rely on</td>
</tr>
<tr>
<td>- Level-3 BLAS operations</td>
</tr>
</tbody>
</table>

Those new algorithms
- have a very low granularity, they scale very well (multicore, petascale computing, …)
- remove a lot of dependencies among the tasks, (multicore, distributed computing)
- avoid latency (distributed computing, out-of-core)
- rely on fast kernels

Those new algorithms need new kernels and rely on efficient scheduling algorithms.
**A New Generation of Software:**
Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70's) (Vector operations)</td>
</tr>
<tr>
<td>LAPACK (80's) (Blocking, cache friendly)</td>
</tr>
<tr>
<td>ScalAPACK (90's) (Distributed Memory)</td>
</tr>
<tr>
<td>PLASMA (00's) New Algorithms (many-core friendly)</td>
</tr>
</tbody>
</table>

**Those new algorithms**
- have a very **low granularity**, they scale very well (multicore, petascale computing, ...)  
- **removes a lots of dependencies** among the tasks, (multicore, distributed computing)  
- **avoid latency** (distributed computing, out-of-core)  
- rely on fast kernels

**Those new algorithms need new kernels and rely on efficient scheduling algorithms.**
**Fork-Join vs. Dynamic Execution**

Fork-Join – parallel BLAS

DAG-based – dynamic scheduling

Experiments on Intel's Quad Core Clovertown with 2 Sockets w/ 8 Treads

Time saved
Provide Highest Performance

LU -- octa-socket, dual-core Opteron

- PLASMA & ACML BLAS
- ACML LU
- MKL LU
- LAPACK & ACML BLAS

multi-core - x86

- PLASMA
- MKL
- ACML
- LAPACK