LINPACK Benchmark Optimizations on a Virtual Processor Grid

Ed Anderson (eanderson@na-net.ornl.gov)
Maynard Brandt (retired)
Chao Yang (cwy@cray.com)
Outline

• Organization of the Cray LINPACK Benchmark code
• Kernel optimizations on the CRAY X1
• The virtual processor grid
The LINPACK benchmark

Solve a linear system

\[ Ax = b \]

using Gaussian elimination with partial pivoting.

The problem size and implementation are not specified.

The algorithm factors \( A = LU \) and solves

\[ y = L^{-1} b; \quad x = U^{-1} y \]

Performance results are specified in

- Gflop/s (billions of floating-point operations per second)
- Tflop/s (trillions of floating-point operations per second)
Block LU factorization

Right-looking algorithm:

I. Factor block column into $A = LU$

II. Exchange rows and update block row

III. Update the rest of the matrix using matrix multiplication
Optimizing the panel factorization

Sub-blocking or recursion is used within the block column so that more work is done in the optimized MM kernel.
Software choices

HPL (High Performance LINPACK)
http://www.netlib.org/benchmark/hpl
  • Block-cyclic distribution
  • Column-wise storage of blocks
  • MPI communication

Cray’s LINPACK Benchmark code
  • Block-cyclic distribution
  • Row-wise storage of blocks
  • MPI or SHMEM communications
2-D block cyclic data distribution

Example on a 2x3 processor grid:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Local view: ScaLAPACK/HPL

Blocks stored by columns, on processor 0:

\[
\begin{array}{ccc}
A_{1,1} & A_{1,4} & A_{1,7} \\
A_{3,1} & A_{3,4} & A_{3,7} \\
A_{5,1} & A_{5,4} & A_{5,7} \\
A_{7,1} & A_{7,4} & A_{7,7}
\end{array}
\]

\[A(mb*nrblks, nb*ncblks)\]

Advantage: With abstraction of BLAS and LAPACK routines, we can maintain much of the LAPACK design.

Disadvantage: As matrix size gets large, distribution blocks get spread out through memory.
## Local view: Cray LBM code

Blocks stored by rows, processor 0:

<table>
<thead>
<tr>
<th>A_{1,1}</th>
<th>A_{1,4}</th>
<th>A_{1,7}</th>
<th>A(mb, nb, ncb, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{3,1}</td>
<td>A_{3,4}</td>
<td>A_{3,7}</td>
<td>A(mb, nb, ncb, 2)</td>
</tr>
<tr>
<td>A_{5,1}</td>
<td>A_{5,4}</td>
<td>A_{5,7}</td>
<td>A(mb, nb, ncb, 3)</td>
</tr>
<tr>
<td>A_{7,1}</td>
<td>A_{7,4}</td>
<td>A_{7,7}</td>
<td>A(mb, nb, ncb, 4)</td>
</tr>
</tbody>
</table>

**Advantage:** Distribution blocks and block rows are contiguous.

**Disadvantage:** More indexing with the 4-D array.
Main computation kernel

After communication of the column and row blocks, each processor performs a matrix-matrix multiplication of the form:

In ScaLAPACK/HPL, one call to SGEMM is required for this operation.
Optimizing data layout for SGEMM

With row-wise storage of blocks, one call to SGEMM is needed to update each local block row.

\[
\begin{align*}
\text{[Blocks]} & = \text{[Blocks]} - \text{[X]} \\
\text{[Blocks]} & = \text{[Blocks]} - \text{[X]} \\
\text{[Blocks]} & = \text{[Blocks]} - \text{[X]}
\end{align*}
\]

(all the same block)
Internals of SGEMM on T3E

Basic operation: nb-by-nb matrix times nb-by-n matrix

\[
\begin{array}{ccc}
\text{nb} & \times & \text{nb} \\
\text{nb} & \text{nb} & \text{n} \\
A & \times & B \\
\end{array}
\]

Innermost kernel: 12×nb matrix-vector multiply, 12-element result
The nb-by-nb block of A is used repeatedly and will reside in cache.
The columns of B are streams (if LDB=nb, B is one long stream).
The result vector is held in registers until combined into C.
Transition of SGEMM: T3E $\rightarrow$ X1

- **T3E**: Result vector was 12 elements long
- **X1**: Result vector can be VL elements long (to 64)
Transition of SGEMM: T3E $\rightarrow$ X1

- **T3E**: nb x nb block constrained by size of 1 set of Scache ($4096W = 64x64$)
- **X1**: nb x nb block needs to fit in 2 MB cache ($256KW = 512x512$)
Transition of SGEMM: T3E $\rightarrow$ X1

T3E: Compute one column of C at a time
Transition of SGEMM: T3E $\rightarrow$ X1

T3E: Compute one column of C at a time
X1: Compute 4 columns of C at a time
Transition of SGEMM: T3E $\rightarrow$ X1

T3E: Compute $12 \times nb$ matrix-vector multiply in innermost kernel
Transition of SGEMM: T3E $\rightarrow$ X1

\[ \begin{align*}
A &\quad \text{nb} \quad \text{nb} \\
\downarrow &\quad \text{n} \quad \text{x} \\
\text{X1} &\quad \text{n} \quad \text{nb} \\
\downarrow &\quad \text{n} \\
C &\quad \text{nb} \quad \text{nb}
\end{align*} \]

T3E: Compute 12 x nb matrix-vector multiply in innermost kernel
X1: Each SSP computes a 64 x nb matrix-vector multiply concurrently
Internals of SGEMM on X1

Basic operation: nb-by-nb matrix times nb-by-n matrix

\[ \begin{array}{c|c|c}
\text{SSP0} & \text{SSP1} & \text{SSP2} \\
\hline
\text{A} & \text{B} & \text{C} \\
\end{array} \]

The nb-by-nb block of A is used repeatedly and will reside in cache. B is read 4 columns at a time and is shared by the 4 SSPs. The result vector is held in registers until combined into C.
Prototype matrix multiply code

subroutine sgemmn( m, n, k, alpha, a, inra, b, inrb,
                 &                    beta, c, inrc )
    integer m, n, k, inra, inrb, inrc
    real alpha, beta, a(inra,*), b(inrb,*), c(inrc,*)
    integer i, js, m4
    cdir$ SSP_PRIVATE dmmnn
    m4 = (m+3)/4
    do i = 0, 3
        js = min( m4, max( m-i*m4, 0 ) )
        call dmmnn( js, n, k, alpha, a(1+i*m4,1), inra,
                 &                    b, inrb, beta, c(1+i*m4,1), inrc )
    end do
end

Compile with:

    ftn -Oaggress -O3 -s default64 -c sgemmn.f

LINPACK Benchmark Optimizations
on a Virtual Processor Grid
Optimizing the row exchanges

A row exchange is performed at each step of the block column factorization to put the largest element (in absolute value) on the diagonal. The vector IPIV records the exchanges.

Example:

IPIV(1) = 20
IPIV(2) =  6
IPIV(3) =  9
IPIV(4) = 31
IPIV(5) =  5
IPIV(6) = 22
IPIV(7) = 20
IPIV(8) = 20
Gather/scatter indices

We can avoid synchronizing after every exchange by translating IPIV into gather and scatter permutation vectors.

<table>
<thead>
<tr>
<th>scatter</th>
<th>gather</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -&gt; 7</td>
<td>1 &lt;-&gt; 20</td>
</tr>
<tr>
<td>2 -&gt; 22</td>
<td>2 &lt;-&gt; 6</td>
</tr>
<tr>
<td>3 -&gt; 9</td>
<td>3 &lt;-&gt; 9</td>
</tr>
<tr>
<td>4 -&gt; 31</td>
<td>4 &lt;-&gt; 31</td>
</tr>
<tr>
<td>5 -&gt; 5</td>
<td>5 &lt;-&gt; 5</td>
</tr>
<tr>
<td>6 -&gt; 2</td>
<td>6 &lt;-&gt; 22</td>
</tr>
<tr>
<td>7 -&gt; 8</td>
<td>7 &lt;-&gt; 1</td>
</tr>
<tr>
<td>8 -&gt; 20</td>
<td>8 &lt;-&gt; 7</td>
</tr>
</tbody>
</table>
Optimizing the communication

To optimize the communication, rows to be exchanged are first copied locally into a contiguous buffer.
When a $p \times q$ grid isn’t enough

Shortcomings of existing algorithm:

- Many processors are idle during column factorization.
- Not every processor count factors neatly into $N_p = p \times q$.

Example:

- $124 = 4 \times 31$
- $123 = 3 \times 41$
- $122 = 2 \times 61$
- $121 = 11 \times 11$

- On some systems it is better to leave one or two processors idle ($N_p - 1$ may not factor neatly).
The Virtual Processor Grid

Generalize the 2-D grid factorization to \( p \times q = k \times N_p \), where \( k \geq 1 \), \( p \leq N_p \), and \( \text{lcm}(p,N_p) = k \times N_p \).

Example: 6 processors in a 4\( \times \)3 virtual grid

```
  0 | 4 | 2
  1 | 5 | 3
  2 | 0 | 4
  3 | 1 | 5
```

This becomes the tiling pattern for the distributed 2-D matrix
Data structure for VPG

Recall the 4-D array used for row-wise storage of blocks:

\[ A(\text{mb, nb, ncblks, nrblks}) \]

Now add another dimension for the virtual processor index:

\[ A(\text{mb, nb, ncblks, nrblks, nvpi}) \]

Maintains contiguousness of distribution blocks and row blocks.
Need to add a loop over the virtual processor indices.
No extra storage except for some extra buffers for each virtual processor.
Coding issues for VPG

Loop doesn’t always go from 1 to nvpi.
Example: Send second column of following matrix across rows.

\[
\begin{array}{cccc}
0 & 4 & 2 & 0 \\
1 & 5 & 3 & 1 \\
2 & 0 & 4 & 2 \\
3 & 1 & 5 & 3 \\
0 & 4 & 2 & 0 \\
1 & 5 & 3 & 1 \\
\end{array}
\]

Send is initiated with virtual processor index 1 on \{4,5\}, with v.p. index 2 on \{0,1\}.

Recv is initiated with virtual processor index 2 on \{2,3,4,5\} and with v.p. index 1 on \{0,1\}

do i = 0, nvpi-1
   \{work on block numbered 1 + mod(start-1+i, nvpi)\}
end do
LINPACK Benchmark Performance
CRAY X1, 124 MSPs

Gflop/s vs. N for different grid configurations:
- 4x31 grid
- 31x4 grid
- 32x31 grid
Summary

• Storage order of distribution blocks was optimized for the cache.
• Leading dimension was padded from 256 to 260 to optimize the matrix-multiply kernel.
• Main computational kernel uses SSP parallelism.
• Communication was optimized using SHMEM.
• No barriers! Communication was extensively overlapped with computation.
• Virtual processor grid improves parallelism of column and row operations.