Lecture 6: ScaLAPACK & hybrid MPI/OpenMP programming

http://people.inf.ethz.ch/iyves/pnc11/

Peter Arbenz*, Andreas Adelmann**
*Computer Science Department, ETH Zürich,
  E-mail: arbenz@inf.ethz.ch
**Paul Scherrer Institut, Villigen
  E-mail: andreas.adelmann@psi.ch
Message passing is the common way of communicating on parallel computers with distributed memory.

The Message Passing Interface (MPI) has become a de facto standard for message passing on multicounters.

We learned some of the MPI basics: point-to-point communication (blocked and nonblocked) and collective communication.

Today we start investigating software that is built on top of MPI.

A number of software packages that are used on multicounters make use of MPI, but hide the actual communication from the user.
MPI (cont.)

- We will consider some of them
  - **ScalAPACK** (Scalable LAPACK) provides solvers for linear systems, least squares, and eigenvalue problems for dense matrices.
  - **PLASMA**: shared memory LA package.
  - **Trilinos** is a “framework” for solving all kinds of problems that involve unstructured problems with sparse matrices.
  - **IPPL** provides a “parallel particle layer” to handle simulations with large numbers of particles.

- There are software layers (e.g. UPC) that provide a **global address space**, that allow shared memory programming on distributed memory architectures.
Mapping vectors on processes: cyclic distribution

$p$: number of processes  
$x \in \mathbb{R}^n$: vector

We proceed like when dealing a deck of cards: The first number $x_0$ is stored on processor 0, the second number $x_1$ on processor 1, and so on, until the $p$-th number $x_{p-1}$ that is stored on processor $p - 1$. The next number $x_p$ is stored again on processor 0, etc. So,

$$x_i \text{ is stored at position } j \text{ on processor } k \text{ if } i = j \cdot p + k, \ 0 \leq k < p. \tag{1}$$

Similar to: omp schedule (static, 1)
Example of cyclic distribution

Let \( \mathbf{x} \) be a vector with 22 elements \( x_0, x_1, \ldots, x_{21} \). If \( \mathbf{x} \) is distributed cyclically, the 22 elements are distributed over the processors 0 to 4 like

\[
\begin{align*}
0 & : x_0 \quad x_5 \quad x_{10} \quad x_{15} \quad x_{20} \\
1 & : x_1 \quad x_6 \quad x_{11} \quad x_{16} \quad x_{21} \\
2 & : x_2 \quad x_7 \quad x_{12} \quad x_{17} \\
3 & : x_3 \quad x_8 \quad x_{13} \quad x_{18} \\
4 & : x_4 \quad x_9 \quad x_{14} \quad x_{19}
\end{align*}
\]

The cyclic distribution is displayed in Fig. 1. Each of the 22 vector elements is given a ‘color’ indicating on which processor it is stored.

**Figure 1:** Cyclic distribution of a vector
Mapping vectors on processes: block distribution

We may distribute the vector in $p$ big blocks of length $b = \lceil n/p \rceil$. For vectors of length $n$ we have

\[
 x_i \text{ is stored at position } k \text{ on processor } j \text{ if } i = j \cdot b + k, \ 0 \leq k < b. \tag{2}
\]

Similar to: omp schedule (static)
Example of block distribution

In the example above the first \( \lceil 22/5 \rceil = 5 \) elements of \( \mathbf{x} \) would go on processor 0, the next 5 elements on processor 1, and so on. With this block distribution the 22-vector of before is distributed in the following way on 5 processors.

\[
\begin{align*}
0 &: \ x_0 \ x_1 \ x_2 \ x_3 \ x_4 \\
1 &: \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \\
2 &: \ x_{10} \ x_{11} \ x_{12} \ x_{13} \ x_{14} \\
3 &: \ x_{15} \ x_{16} \ x_{17} \ x_{18} \ x_{19} \\
4 &: \ x_{20} \ x_{21}
\end{align*}
\]

**Figure 2:** Block distribution of a vector
Mapping vectors on processes: block-cyclic distribution

The block-cyclic distribution is a compromise between the two previous distributions. It alleviates the badly balanced distribution of the vector elements but compared with the cyclic distribution reduces the overhead of the memory accesses.

Let $b$ be a block size, usually a small integer like 2, 4, or 8. Then we partition the $n$-vector $\mathbf{x}$ in $n_b = \lceil n/b \rceil$ blocks the first $n_b - 1$ of which consist of $b$ elements while the last has length $n - (n_b - 1)b = n \mod b$. The blocks are now distributed cyclically over the $p$ processors. Let $i = j \cdot b + k$. Then from (2) we know that $x_i$ is the $k$-th element in block $j$. 
Block-cyclic distribution

Let now \( j = \ell \cdot p + m \). Then, interpreting (1) for blocks, we see that block \( j \) is the \( \ell \)-th block on processor \( m \). Thus,

\[
x_i \text{ is stored at position } \ell \cdot b + k \text{ on processor } m \\
\text{if } \quad i = j \cdot b + k, \quad 0 \leq k < b \\
\text{and } \quad j = \ell \cdot p + m, \quad 0 \leq m < p.
\]

The block-cyclic distribution generalizes both cyclic and block distribution. The cyclic distribution is obtained if \( b = 1 \); the block distribution is obtained if \( b = \lceil n/p \rceil \).

Similar to: omp schedule (static, b)
Example of block-cyclic distribution

With block size 2 and 5 processors, the elements of the 22 element vector $x$ are stored like

\[
\begin{align*}
0 & : \ x_0 \ x_1 \ x_{10} \ x_{11} \ x_{20} \ x_{21} \\
1 & : \ x_2 \ x_3 \ x_{12} \ x_{13} \\
2 & : \ x_4 \ x_5 \ x_{14} \ x_{15} \\
3 & : \ x_6 \ x_7 \ x_{16} \ x_{17} \\
4 & : \ x_8 \ x_9 \ x_{18} \ x_{19}
\end{align*}
\]

This distribution is displayed in Fig. 3.

Figure 3: Block distribution of a vector

The difference of the number of blocks per processor is at most 1. Therefore, the difference in the number of elements per processor cannot exceed $b$. 
Mapping matrices on processes

$p$: number of processes, matrix $A \in \mathbb{R}^{m \times n}$

Can apply vector mapping to matrix columns or rows.

**block column distribution** (with 5 processors)
Mapping matrices on processes: cyclic column distribution

Example with 5 processors:

Similarly: row cyclic, column / row block cyclic
Mapping matrices on procs: 2D block-cyclic distribution

We can mix row and column distributions.
Let $p = p_1 p_2$. We can then think of the processors being arranged in a $p_1$-by-$p_2$ process grid.

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

Figure 4: $2 \times 3$ process grid arranged in row-major order

Blocks of size $b_1$-by-$b_2$ are distributed over the processors. *The rules for the block-cyclic distribution for vectors is applied for rows and columns seperately*, using block size $b_1$ and $p_1$ processors for the rows and block size $b_2$ and $p_2$ processors for the columns.
2D block-cyclic distribution (global view)

Figure 5: Block-cyclic distribution of a $15 \times 20$ matrix on a $2 \times 3$ processor grid with blocks of $2 \times 3$ elements
2D block-cyclic distribution (local view)

Figure 6: Block-cyclic distribution of a $15 \times 20$ matrix on a $2 \times 3$ processor grid with blocks of $2 \times 3$ elements
LAPACK and the BLAS

LAPACK (Linear Algebra PACKage) is a library of Fortran 77 subroutines for solving the most commonly occurring problems in numerical linear algebra. It has been designed to be efficient on a wide range of modern high-performance computers.

LAPACK is written in a way that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). While LINPACK and EISPACK relied on the vector operations in BLAS-1, LAPACK calls BLAS-2 (matrix-vector operations) and BLAS-3 (matrix-matrix operations) to exploit the fast memories (caches) of today's high-performance computers.

Most of the work has been done at universities (U of TN at Knoxville, U of CA at Berkeley) and at NAG, Oxford.

The software is available from http://www.netlib.org/lapack
Blocked Gaussian elimination with LAPACK

\[
P \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11} U_{11} & L_{11} U_{12} \\ L_{21} U_{11} & L_{21} U_{12} + L_{22} U_{22} \end{bmatrix}
\]
1. Gaussian elimination with column pivoting

\[
P \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}
\]

2. Apply \( P \) (row interchanges) on \( L_0 \) and \( \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} \)

3. Forward substitution:

\[
U_{12} = L_{11}^{-1} A_{12}
\]

4. Update rest of the matrix

\[
\tilde{A}_{22} = A_{22} - L_{21} U_{12} \quad ( = L_{22} U_{22})
\]
The computation of the above steps in the LAPACK routine, \texttt{DGETRF}, involves the following operations:

1. \texttt{DGETF2}: Apply the LU factorization on an $m \times b$ column panel of $A$ (i.e., $A_{11}$ and $A_{21}$)
   - [Repeat $b$ times ($i = 1, \ldots, b$)]
     - \texttt{IDAMAX}: find the (absolute) maximum element of the $i$-th column and its location
     - \texttt{DSWAP}: interchange $i$-th row with row that holds maximum
     - \texttt{DSCAL}: scale the $i$-th column of the matrix
     - \texttt{DGER}: update the trailing submatrix

2. \texttt{DLASWP}: Apply row interchanges to the left / right of panel.

3. \texttt{DTRSM}: Compute the $b \times (n - b)$ row panel of $U$

   $$U_{12} \leftarrow (L_{11})^{-1}A_{12}.$$ 

4. \texttt{DGEMM}: Update the rest of the matrix, $A_{22}$,

   $$\hat{A}_{22} \leftarrow A_{22} - L_{21}U_{12} (= L_{22}U_{22})$$
Complexity of blocked LU factorization

\[
\begin{align*}
\frac{n}{b} & \sum_{i=1}^{n/b} \frac{2}{3} (b)^3 \\
& + \sum_{i=1}^{n/b-1} (i \cdot b)(b)^2 \\
& + \sum_{i=1}^{n/b-1} (i \cdot b)(b)^2 \\
& + \sum_{i=1}^{n/b-1} 2 \ b(i \cdot b)^2 \\
& = (b)^3 \left[ \frac{2}{3} \left( \frac{n}{b} \right)^3 + \mathcal{O} \left( \frac{n}{b} \right)^2 \right] \text{ flop} \\
& = \frac{2}{3} n^3 + \mathcal{O}(n^2) \text{ flop}
\end{align*}
\]

\[A_{11} = L_{11} U_{11}\]

\[A_{21} U^{-1}_{11}\]

\[L^{-1}_{11} A_{12}\]

\[A_{22} - A_{21} A_{12}\]

The blocked algorithm costs as much as the unblocked algorithm.
Accesses to main memory

Here we assume that $3(b)^2$ numbers can be stored in cache.

$$2(b)^2 + \sum_{i=1}^{n/b-1} 2b(i \cdot b) + \sum_{i=1}^{n/b-1} 2b(i \cdot b) + \sum_{i=1}^{n/b-1} 3ib(i \cdot b) = (b)^2 \left[ \left( \frac{n}{b} \right)^3 + O\left( \frac{n}{b} \right)^2 \right] = \frac{1}{b} \left( n^3 + O(n^2) \right)$$

Blocked algorithm executes $\frac{3}{2}b$ flops per memory access (i.e. access to the (slow) main memory). Note: $b$ is bounded by cache size.

In PLASMA each of the blocked operations is a task.
## Performance of DGETRF

<table>
<thead>
<tr>
<th>$n$</th>
<th>Time</th>
<th>Speed</th>
<th>$n$</th>
<th>Time</th>
<th>Speed</th>
<th>$n$</th>
<th>Time</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.27 sec</td>
<td>308</td>
<td>1000</td>
<td>2.17 sec</td>
<td>307</td>
<td>2000</td>
<td>16.57 sec</td>
<td>322</td>
</tr>
<tr>
<td>1000</td>
<td>0.18 sec</td>
<td>463</td>
<td>1000</td>
<td>1.44 sec</td>
<td>463</td>
<td>2000</td>
<td>11.06 sec</td>
<td>482</td>
</tr>
<tr>
<td>1000</td>
<td>0.13 sec</td>
<td>641</td>
<td>1000</td>
<td>1.08 sec</td>
<td>617</td>
<td>2000</td>
<td>8.16 sec</td>
<td>654</td>
</tr>
<tr>
<td>1000</td>
<td>0.11 sec</td>
<td>757</td>
<td>1000</td>
<td>0.89 sec</td>
<td>749</td>
<td>2000</td>
<td>6.73 sec</td>
<td>792</td>
</tr>
<tr>
<td>1000</td>
<td>0.10 sec</td>
<td>833</td>
<td>1000</td>
<td>0.80 sec</td>
<td>833</td>
<td>2000</td>
<td>6.42 sec</td>
<td>831</td>
</tr>
<tr>
<td>1000</td>
<td>0.09 sec</td>
<td>926</td>
<td>1000</td>
<td>0.78 sec</td>
<td>855</td>
<td>2000</td>
<td>7.74 sec</td>
<td>689</td>
</tr>
<tr>
<td>1000</td>
<td>0.10 sec</td>
<td>833</td>
<td>1000</td>
<td>0.96 sec</td>
<td>694</td>
<td>2000</td>
<td>10.16 sec</td>
<td>525</td>
</tr>
</tbody>
</table>

**Table 1:** Times [sec] and speed in Mflop/s of DGESV on a P IV (2.4 GHz, 1 GB)
Scalable parallel solution of systems of equations with ScaLAPACK

Scalable parallel solution of systems of equations with ScaLAPACK

SeaLAPACK Software Hierarchy

SeaLAPACK

PBLAS

LAPACK

BLACS

BLAS

Message Passing Primitives (MPI, PVM, MPL, GAM, etc.)

Global

Local
Building blocks of ScaLAPACK

- **LAPACK, BLAS**
- **BLACS**: Basic Linear Algebra Communication Subroutines
  - Kernels based on MPI (or other message passing libraries) for communication as needed in ScaLAPACK.
  - Goals according to good experience with BLAS: easy to use, highly optimized, portable
  - Assumes matrices to be distributed in a 2D block cyclic layout.
- **PBLAS**: Parallel BLAS
  - Similar interfaces as BLAS.
  - PBLAS operate on matrices distributed in a 2D block cyclic layout.
  - (Additional parallel operation: matrix transposition)
Four Basic Steps Required to Call a ScaLAPACK Routine

1. Initialize the process grid ($\equiv$ BLACS)
2. Distribute data (matrices/vectors) on the process grid
3. Call ScaLAPACK routine
4. Release the process grid ($\equiv$ BLACS)
Process grid and scoped operations.

- Commonly, processes in a parallel environment are presented to the user as a linear array of processes numbered 0 to \( p - 1 \).
- BLACS allows to arrange processes in a process grid with \( R \) process rows and \( C \) process columns, \( R \cdot C \leq p \). The processes are then labeled \((i, j)\) with \( 0 \leq i < R \) and \( 0 \leq j < C \).

![Figure 7: Eight processes mapped to a 2 × 4 process grid](image)

- A scoped operation corresponds to collective communication in MPI. Natural scopes within a process grid are process row, process column, and all processes.

Context.

Each process grid is enclosed in a context. A context has the same meaning as a communicator in MPI.
Distribute the Matrix on the Process Grid

All global matrices must be distributed on the process grid prior to the invocation of a ScaLAPACK routine. It is the user’s responsibility to perform this task.

This can in fact be very cumbersome and is very error-prone.

Each global matrix that is to be distributed across the process grid must be assigned an array descriptor.

Matrices are distributed in a 2D block cyclic layout.
Block Gaussian elimination (redux)

1. Gaussian elimination with column pivoting

\[
P \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}
\]

2. Apply \(P\) (row interchanges) to

\[
L_0 \quad \text{and} \quad \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}
\]

3. Forward substitution:

\[
U_{12} = L_{11}^{-1} A_{12}
\]

4. Update rest of the matrix

\[
\tilde{A}_{22} = A_{22} - L_{21} U_{12} (= L_{22} U_{22})
\]
Parallel blocked LU factorization

We assume a $P \times Q$ process grid.

The matrix is distributed on processes using a 2D block cyclic layout with square $(b \times b)$ blocks.

1. Gaussian elimination in panel ($b$ columns)

$$
P \begin{bmatrix}
A_{11} \\
A_{21}
\end{bmatrix} = 
\begin{bmatrix}
L_{11} \\
L_{21}
\end{bmatrix} U_{11}
$$

We neglect the arithmetic operations (flops) in this step.

1.1 Determine pivot row

Determine maximal element in column and make it available to all processes in process column (allreduce).

1.2 Swap rows in current process column (send / receive)
2. Forward substitution:

\[ U_{12} = L_{11}^{-1} A_{12} \]

2.3 Broadcast pivot information on process row
2.4 Swap remaining rows
2.5 Broadcast lower trapezoid factor \( \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \) on process row
2.6 Triangular solve \( L_{11}^{-1} A_{12} \): BLAS-3 TRSM

3. Update rest of the matrix

\[ \tilde{A}_{22} = A_{22} - L_{21} U_{12} (= L_{22} U_{22}) \]

3.7 Broadcast upper trapezoid factor \( U_{12} \) on process columns
3.8 Rank-\( b \) update: BLAS-3 GEMM
Complexity

In summary: \( p = p_r \cdot p_c \)

Time:
\[
T_{LU}(p_r, p_c, b) \approx \left( 2n \log p_r + \frac{2n}{b} \log p_c \right) t_{\text{startup}} \\
+ \frac{n^2}{2p} (4p_r + p_c \log p_c + p_r \log p_r) t_{\text{word}} \\
+ \frac{2n^3}{3p} t_{\text{flop}}
\]

Speedup:
\[
S_{LU}(p_r, p_c, b) = \frac{T_{LU}(1, 1, b)}{T_{LU}(p_r, p_c, b)} \\
= \frac{p_r \cdot p_c}{1 + \frac{3p}{n^2} \left( \log p_r + \frac{\log p_c}{b} \right) \frac{t_{\text{startup}}}{t_{\text{flop}}} + \frac{3}{4n} (4p_r + p_c \log p_c + p_r \log p_r) \frac{t_{\text{word}}}{t_{\text{flop}}}}
\]
Discussion

- Isoefficiency holds approximately if \( n^2 \sim p \log_2 p \).
- The number of process columns \( p_c \) has a slightly smaller weight in the \( t_{\text{startup}} \) and in the \( t_{\text{word}} \) term than the number of process rows \( p_r \) (pivoting!). Therefore it might be advisable to choose \( p_c > p_r \).
- We know that \( t_{\text{flop}} \) depends heavily on the block size \( b \). The performance prediction for small \( b \) is probably too good.
- Big \( b \) on the other side means that at the end of the factorization only a few processors have work (load imbalance). Therefore \( b \) should not be too big. (From our earlier consideration we remember that \( b \) is also bounded by the cache size.)
Timings of the ScaLAPACK system solver pdgesv

<table>
<thead>
<tr>
<th>$p_r$</th>
<th>$p_c$</th>
<th>$b$</th>
<th>time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1440</td>
<td>2880</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>27.54</td>
<td>347.6</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>2.98</td>
<td>17.5</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>3.04</td>
<td>12.1</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>3.49</td>
<td>11.2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5.45</td>
<td>14.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>54.60</td>
<td>534.4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4.69</td>
<td>26.3</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>3.43</td>
<td>18.7</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>4.11</td>
<td>15.7</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>5.07</td>
<td>15.9</td>
</tr>
</tbody>
</table>

Timings of the ScaLAPACK system solver pdgesv on one processor and on 36 processors with varying dimensions of the process grid.
Times and speedups for various problem sizes

<table>
<thead>
<tr>
<th></th>
<th>$n = 500$</th>
<th></th>
<th>$n = 1000$</th>
<th></th>
<th>$n = 2000$</th>
<th></th>
<th>$n = 5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$t$</td>
<td>$S(p)$</td>
<td>$t$</td>
<td>$S(p)$</td>
<td>$t$</td>
<td>$S(p)$</td>
<td>$t$</td>
</tr>
<tr>
<td>1</td>
<td>0.959</td>
<td>1</td>
<td>8.42</td>
<td>1.0</td>
<td>121</td>
<td>1</td>
<td>2220</td>
</tr>
<tr>
<td>2</td>
<td>0.686</td>
<td>1.4</td>
<td>4.92</td>
<td>1.7</td>
<td>47.3</td>
<td>2.7</td>
<td>1262</td>
</tr>
<tr>
<td>4</td>
<td>0.788</td>
<td>1.2</td>
<td>3.16</td>
<td>2.7</td>
<td>17.7</td>
<td>6.9</td>
<td>500</td>
</tr>
<tr>
<td>8</td>
<td>0.684</td>
<td>1.4</td>
<td>2.31</td>
<td>3.7</td>
<td>10.8</td>
<td>11</td>
<td>303</td>
</tr>
<tr>
<td>16</td>
<td>1.12</td>
<td>0.86</td>
<td>2.45</td>
<td>3.4</td>
<td>7.43</td>
<td>16</td>
<td>141</td>
</tr>
<tr>
<td>32</td>
<td>1.12</td>
<td>0.86</td>
<td>2.27</td>
<td>3.7</td>
<td>6.53</td>
<td>19</td>
<td>48</td>
</tr>
</tbody>
</table>

Times $t$ in seconds and speedups $S(p)$ for various problem sizes $n$ and processor numbers $p$ for solving a random system of equations with the general solver $\text{pdgesv}$ of ScaLAPACK on the Beowulf cluster. The block size for $n \leq 1000$ is $b = 32$, the block size for $n > 1000$ is $b = 16$. 
Timings on Gonzales for fixed blocksize, $n = 1k, 2k, 5k$

<table>
<thead>
<tr>
<th>$b$</th>
<th>$N = 1000$</th>
<th></th>
<th>$N = 2000$</th>
<th></th>
<th>$N = 5000$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time/s</td>
<td>S</td>
<td>Time/s</td>
<td>S</td>
<td>Time/s</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>0.141</td>
<td>1.0</td>
<td>0.643</td>
<td>1.0</td>
<td>10.124</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>0.082</td>
<td>1.7</td>
<td>0.290</td>
<td>2.2</td>
<td>3.320</td>
<td>3.0</td>
</tr>
<tr>
<td>8</td>
<td>0.076</td>
<td>1.9</td>
<td>0.253</td>
<td>2.5</td>
<td>2.221</td>
<td>4.6</td>
</tr>
<tr>
<td>16</td>
<td>0.078</td>
<td>1.8</td>
<td>0.218</td>
<td>2.9</td>
<td>1.674</td>
<td>6.0</td>
</tr>
<tr>
<td>32</td>
<td>0.077</td>
<td>1.8</td>
<td>0.223</td>
<td>2.9</td>
<td>1.568</td>
<td>6.5</td>
</tr>
<tr>
<td>64</td>
<td>0.085</td>
<td>1.7</td>
<td>0.249</td>
<td>2.6</td>
<td>1.663</td>
<td>6.1</td>
</tr>
</tbody>
</table>

Timings and speedups $S$ for `pdgesv` on Gonzales with fixed number of processors $p = 32$ for various block sizes $b$. 
MPI virtual topologies

- A **virtual topology** describes a mapping/ordering of MPI processes into a geometric “shape”.
- Two main types of MPI topologies: **cartesian** (grid) and **graph**.
- MPI topologies are **virtual**: no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and **groups**. (Groups: ordered set of processes)
- Must be ‘programmed’ by the application developer.
MPI virtual topologies (cont.)

- MPI virtual topologies are convenient
  - Useful for applications with specific communication patterns – patterns that match an MPI topology structure.
  - Example: a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

- MPI virtual topologies may lead to efficient communication
  - May support process mapping based upon the physical characteristics of a given parallel machine.
  - Depends on MPI implementation
MPI virtual topologies (Example)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>(0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>(0,2)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>(0,3)</td>
</tr>
<tr>
<td>4</td>
<td>(1,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>(1,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>(1,2)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>(1,3)</td>
</tr>
<tr>
<td>8</td>
<td>(2,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>(2,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>(2,2)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>(2,3)</td>
</tr>
<tr>
<td>12</td>
<td>(3,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>(3,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>(3,2)</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>(3,3)</td>
</tr>
</tbody>
</table>
MPI virtual topologies (code)

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 16

int main(int argc, char *argv[])
{
    int numtasks, cartrank, colrank, rank, reorder=0, count=10,
    int buffer;

    MPI_Comm cartcomm, colcomm;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```
MPI virtual topologies (more code)

```c
if (numtasks == SIZE) {
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
    MPI_Comm_rank(cartcomm, &cartrank);
    MPI_Cart_coords(cartcomm, cartrank, 2, coords);

    MPI_Cart_sub(cartcomm, belongs, &colcomm);
    MPI_Comm_rank(colcomm, &colrank);

    buffer = rank;
    MPI_Allreduce(MPI_IN_PLACE, &buffer, 1, MPI_INT, MPI_SUM, colcomm);

    printf("%d %d [%d,%d] %d:\n", cartrank, colrank, coords[0],
            coords[1], buffer);
}
else
    printf("Must specify %d processors. Terminating.\n", SIZE);
```
MPI virtual topologies (output)

ineichen@obiwan:~$ mpirun -np 16 ./topo
2 0 [0,2] 32:
6 1 [1,2] 32:
9 2 [2,1] 28:
10 2 [2,2] 32:
12 3 [3,0] 24:
14 3 [3,2] 32:
1 0 [0,1] 28:
4 1 [1,0] 24:
5 1 [1,1] 28:
8 2 [2,0] 24:
3 0 [0,3] 36:
...
Hybrid MPI/OpenMP programming

- Why should we care about combining OpenMP and MPI?
- Common lore:
  “An OpenMP+MPI hybrid code is never faster than a pure MPI code on the same hybrid hardware, except for obvious cases”
- Data placement in MPI is (hopefully) optimal. How can OpenMP improve?
- Need not be true.
  - Many MPI processes need memory space, in particular on many-many core nodes.
- Overlapping of computation and communication via non-blocking MPI often does not work.
**Taxonomy of hybrid “modes”**

**Vector mode:** MPI is called only outside OpenMP parallel regions. This is what many people mean when they say “hybrid”
- Similar to what we did on vector-parallel machines

**Task mode:** One or more threads in the parallel region are dedicated to special tasks, like doing communication in the background
- This is functional parallelism on the thread level

Slide by G. Hager, see p. 51
OpenMP multithreading in MPI process

- **MPI_THREAD_SINGLE**: There is no OpenMP multithreading in the program.
- **MPI_THREAD_FUNNELED**: All of the MPI calls are made by the master thread. This will happen if all MPI calls are outside OpenMP parallel regions or are in master regions.
- **MPI_THREAD_SERIALIZED**: Multiple threads make MPI calls, but only one at a time.
- **MPI_THREAD_MULTIPLE**: Any thread may make MPI calls at any time.
Topology ("mapping") choices with MPI+OpenMP

- One MPI process per node
  - Diagram showing different configurations

- One MPI process per socket
  - Diagram showing different configurations

- OpenMP threads pinned "round robin" across cores in node
  - Diagram showing different configurations

- Two MPI processes per node
  - Diagram showing different configurations

Slide by G. Hager, see p. 51
**MPI Initialization**

Instead of starting MPI by `MPI_Init`

```c
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int *provided)
```

required: the desired level of thread support.
provided: the actual level of thread support provided by the system.

Thread support at levels `MPI_THREAD_FUNNELED` or higher allows potential overlap of communication and computation.
Algorithmic reason for hybrid programming

- Let’s assume a cubical domain Ω with an overlaid $n \times n \times n$ grid.

- We make a domain decomposition approach. Each subdomain consists of

  $$\frac{n}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$$

  grid points

- To exchange data at the interfaces, a halo or ghost layer is needed.
Algorithmic reason for hybrid programming (cont.)

Situation in 2D:

12 × 12 grid, 16 subdomains with 3 × 3 subgrids
Algorithmic reason for hybrid programming (cont.)

Simple 3D numerical example:

(1) Let’s assume that we have 8 cores on a shared-memory node.
(2) Let’s assume that the subdomains are $10 \times 10 \times 10$.

In a MPI-only computation there are 8 MPI processes on a node. For each of the subdomains we have to store $(10 + 2)^3 = 12^3 = 1'728$ grid points.

Altogether, on a node, we store $8 \times 1'728 = 13'824$ grid points.

A hybrid MPI-OpenMP computations stores a $20^3$ grid plus halo: In this way we have to store $(20 + 2)^3 = 22^3 = 10'648$ grid points.

The MPI-only program needs 30% more memory space.
How do you distribute loop iterations if one thread of your team is missing?

Straightforward answer: Use nested parallelism

```c
#pragma omp parallel num_threads(2)
{
    if(!omp_get_thread_num()) {
        // do comm thread stuff here
    }
    else {
        #pragma omp parallel num_threads(7)
        {
            #pragma omp for
            // do work threads stuff here
        }
    }
}
```
Reference on hybrid OpenMP/MPI programming


- Georg Hager et al.: MPI+OpenMP hybrid computing (on modern multicore systems)

  Talk given at the 39th Speedup Workshop on High-Performance Computing
  ETH Zürich, September 6-7, 2010.

  Slides available from http://www.speedup.ch/workshops/w39_2010.html
  There are further references on the slides.