HPC Challenge 2006 Awards Competition: xlUPC on BlueGene/L

IBM Research, IBM SWG Toronto, and LLNL
Environment

- **Benchmarks:**
  - HPL, FFT, Random Access and EP STREAM Triad

- **Software**
  - An experimental version of the IBM xlUPC compiler
  - An experimental version of the BG/L communication library

- **Blue Gene characteristics & installations**
  - BG nodes (2 procs. each) have 4M L3 cache, 512 MB local memory; connected by a 3D torus, 175 MB/s/link
  - Blue Gene/X – 1 rack, 2048 procs., 512 GB mem.
  - Blue Gene/W – 20 racks, 40K procs., 10 TB mem.
  - Blue Gene/L – 64 racks, 128K procs., 32 TB mem.
Global HPL

- **UPC naïve version** – nice and simple code, low performance

- **Optimizations:**
  - Calls to BLAS (ESSL in IBM speak) – we introduced multi-dimensional blocking of shared arrays
  - Collective communication – critically needed when scaling to thousands of processors
    - Added when UPC collectives supported (e.g., broadcast in backsolve)
    - update requires broadcast on subset of threads which is not supported in the UPC specification

<table>
<thead>
<tr>
<th>Lines</th>
<th>Cmnts</th>
<th>NCSL File</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>11</td>
<td>30 backsolve.upc</td>
</tr>
<tr>
<td>89</td>
<td>26</td>
<td>48 main.upc</td>
</tr>
<tr>
<td>52</td>
<td>12</td>
<td>35 matgen.upc</td>
</tr>
<tr>
<td>43</td>
<td>25</td>
<td>24 panel.upc</td>
</tr>
<tr>
<td>50</td>
<td>13</td>
<td>30 pivot.upc</td>
</tr>
<tr>
<td>45</td>
<td>16</td>
<td>23 swap.upc</td>
</tr>
<tr>
<td>45</td>
<td>24</td>
<td>15 tri_solve.upc</td>
</tr>
<tr>
<td>101</td>
<td>49</td>
<td>55 update.upc</td>
</tr>
<tr>
<td>63</td>
<td>22</td>
<td>28 hpl.h</td>
</tr>
</tbody>
</table>

536 198 288 Total
Multidimensional blocked data distribution in UPC

- **Syntax:**
  
  ```
  shared [B][B] double A [M][N];
  ```

- **Thread assigned to** \( a[i][j] \):
  
  \[
  p(i, j) = \left( \left\lfloor \frac{i}{B} \right\rfloor \times \left\lceil \frac{N}{B} \right\rceil + \left\lfloor \frac{j}{B} \right\rfloor \right) \mod \text{THREADS}
  \]

- Blocks are assigned sequentially, not in a checkerboard layout.
Performance bottlenecks

- Comp/comm ratio low
  - `upc_memget()` calls overload the CPU that owns A[ii][k]

- Calls for collective communication (and subsets of threads)
  - No appropriate collective,
  - No communicators in UPC

- Collectives should also be used in: `backsolve`, `triangular_solve`, `outer_product`, `max_pivot`
Performance

<table>
<thead>
<tr>
<th>BlueGene Procs</th>
<th>Matrix Size</th>
<th>Gflops</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td>1.47</td>
<td>52.50%</td>
</tr>
<tr>
<td>64</td>
<td>44000</td>
<td>47.17</td>
<td>26.32%</td>
</tr>
<tr>
<td>256</td>
<td>85000</td>
<td>117.87</td>
<td>16.44%</td>
</tr>
</tbody>
</table>

• Remaining issues
  • Load balancing
  • Communication overhead (collectives)
Global FFT - Complex 1-D Discrete Fourier Transform (DFT)

Conventional algorithm: two-dimensional index mapping
- compute DFT of N columns
- multiply element (i,j) by $W_{N^*N}^i$ (twiddle factors)
- compute DFT of N rows

DFTs can be done independently (in parallel)
- Matrix transpose may be needed to make DFTs local
- FFTW library computes local DFTs
Global FFT – UPC code

<table>
<thead>
<tr>
<th>Lines</th>
<th>Blank</th>
<th>Cmnts</th>
<th>NCSL</th>
<th>TP tok</th>
<th>fftp.code</th>
</tr>
</thead>
<tbody>
<tr>
<td>151</td>
<td>18</td>
<td>43</td>
<td>100</td>
<td>1018</td>
<td>fftp.upc</td>
</tr>
<tr>
<td>59</td>
<td>14</td>
<td>23</td>
<td>22</td>
<td>160</td>
<td>fftp.h</td>
</tr>
<tr>
<td>210</td>
<td>32</td>
<td>66</td>
<td>122</td>
<td>1168</td>
<td>total</td>
</tr>
<tr>
<td>(121</td>
<td>24</td>
<td>23</td>
<td>75</td>
<td>637</td>
<td>verify.upc)</td>
</tr>
</tbody>
</table>

**fftv1:**

```c
shared [N*N/THREADS] complex_t ComplexArray_t [N*N];
```

**fftv1**

```c
transpose(X,A);
fftw_on_columns(A);
mult_by_twiddle(A);
transpose(A,Z);
fftw_on_rows(Z);
transpose(Z,A);
```

**fftv2**

```c
local_copy_input(X,A);
fftw_on_columns(A);
mult_by_twiddle(A);
transpose(A,Z);
fftw_on_rows(Z);
```
**Performance analysis**

*On 64 racks FFT performance is limited by the cost of transposes*

Array size: 64 MBytes/thread

Data sent through cross-section each transpose: 32 MBytes/thread

\[ \text{cpabytes} = 32 \text{ MBytes} / \eta \quad = \quad 80 \text{ MBytes} \]

\[ \text{totalbytes} = \text{cpabytes} \times \text{threads} = \quad = 5120 \text{ GBytes} \]

Cross-section BW (64 x 32 x 32 torus)
2 wires/link x 32 x 32 x 2 links

Bandwidth =
4096 x 0.25 Bytes/cycle x 700MHz =
667 GBytes/s

### fftv1 (3 transposes)

\[
\begin{align*}
T_{\text{transpose}} & = \frac{\text{totalbytes}}{\text{BW}} = 7.68\text{s} \\
T_{\text{fftw}} & = 1.8\text{s} \\
T_{\text{twiddle}} & = 1.7\text{s} \\
T_{\text{total}} & = 3 \cdot T_{\text{transpose}} + 2 \cdot T_{\text{fftw}} + T_{\text{twiddle}} \\
\text{Performance} & = \frac{5 \times n \times \log(n)}{T_{\text{total}}} \leq 1843 \text{ GFlops}
\end{align*}
\]

### fftv2 (1 transpose)

\[
\begin{align*}
T_{\text{transpose}} & = \frac{\text{totalbytes}}{\text{BW}} = 7.68\text{s} \\
T_{\text{fftw}} & = 3.4\text{s} \\
T_{\text{twiddle}} & = 2.2\text{s} \\
T_{\text{total}} & = T_{\text{transpose}} + 2 \cdot T_{\text{fftw}} + T_{\text{twiddle}} \\
\text{Performance} & = \frac{5 \times n \times \log(n)}{T_{\text{total}}} \leq 3131 \text{ GFlops}
\end{align*}
\]
### FFT Performance

<table>
<thead>
<tr>
<th>BlueGene Racks</th>
<th>Procs</th>
<th>Array Elements</th>
<th>Arrays TBytes</th>
<th>fftv1 Gflops</th>
<th>fftv2 GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2048</td>
<td>$2^{32}$</td>
<td>0.13</td>
<td>51.29</td>
<td>54.29</td>
</tr>
<tr>
<td>4</td>
<td>8192</td>
<td>$2^{34}$</td>
<td>0.5</td>
<td>124.81</td>
<td>198.93</td>
</tr>
<tr>
<td>16</td>
<td>16384</td>
<td>$2^{36}$</td>
<td>2</td>
<td>512.70</td>
<td>742.90</td>
</tr>
<tr>
<td>64</td>
<td>65536</td>
<td>$2^{38}$</td>
<td>8</td>
<td>1115.00</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Random Access

- Each update is a packet
  - Performance limited by latency, cross-section bandwidth

- Compiler optimization:
  - Identify remote update operations

- Verification: run the algorithm twice

- Changes since last year: optimized packet size

- Lines of code: 107

```c
u64Int ran = starts(NUPDATE/THREADS * MYTHREAD);
upc_forall (i = 0; i < NUPDATE; i++; i) {
    ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
    Table[ran & (TableSize-1)] ^= ran;
}
```
Theoretical GUPS limit on 64 rack BlueGene system

One packet per update (naïve algorithm!)

**Update packets:**
- 12 Byte header
- 4 Bytes opcode, type
- 4 Bytes target SVD
- 4 Bytes offset
- 8 Bytes update value
- 10 Bytes CRC + CF

**42 Bytes on wire**

\[
\left\{ \begin{array}{l}
  \text{Packet size: } 42 \text{ Bytes} \\
  \text{Wire speed: } 4 \frac{\text{cycles}}{\text{Byte}} \\
  \text{CPU speed: } 700 \text{ MHz } = 1.4 \frac{\text{ns}}{\text{cycle}} \\
\end{array} \right\} \rightarrow P = 4.16 \cdot 10^6 \frac{\text{packets}}{\text{second} \cdot \text{link}}
\]

Cross-section bandwidth:
- 64 x 32 x 32 torus:
- 2 wires/link x 32 x 32 x 2 (torus) = 4096 links
- \( BW = 4096 \cdot 4.16 \cdot 10^6 = 17 \cdot 10^9 \text{ packets/s} \)

Half of all packets travel across the cross-section

Theoretical limit = 34 GUPS
## Random Access: Performance Results

<table>
<thead>
<tr>
<th>BlueGene Racks</th>
<th>Mem TB</th>
<th>GUPS 2005</th>
<th>GUPS 2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2048</td>
<td>0.25</td>
<td>0.56</td>
</tr>
<tr>
<td>2</td>
<td>4096</td>
<td>0.5</td>
<td>1.11</td>
</tr>
<tr>
<td>4</td>
<td>8192</td>
<td>1</td>
<td>1.70</td>
</tr>
<tr>
<td>8</td>
<td>16384</td>
<td>2</td>
<td>3.36</td>
</tr>
<tr>
<td>16</td>
<td>32768</td>
<td>4</td>
<td>6.10</td>
</tr>
<tr>
<td>32</td>
<td>65536</td>
<td>8</td>
<td>11.54</td>
</tr>
<tr>
<td>64</td>
<td>131072</td>
<td>16</td>
<td>16.72</td>
</tr>
</tbody>
</table>
Thank you!
Backup
Global HPL Basics (Panel Factorization)

- **Code:**
  - Follow `dgetrf()` floor plan
  - blocked factorization
  - Parallelize inner loops
  - blocks local to threads
  - Comm. granularity: **block**

- **Data:**
  - Need 2-D blocked distribution
  - Block locality, load balance
  - UPC syntax doesn't allow it!
  - ... so we extended UPC
FFT – Matrix transpose

All-to-all communication pattern
- bottleneck for Blue Gene

Blocked transpose
- blocksize $B = \frac{N}{\text{THREADS}}$

Each thread gets one $B \times B$ block from each other threads using upc_memgets
- no strided access with upc_memget
- we need $B$ memgets for each block

Each block is transposed in place at the destination
FFT – Matrix transpose: the code

```c
upc_fornall (i = 0; i < N; i += bsize; &B[i*N])
  for (j = 0; j < N; j += bsize) {
    // copy block to dest row by row
    complex_t * lb = (complex_t *)&B[i*N+j];
    for (unsigned k = 0; k < bsize; k++ )
      upc_memget( lb + k*N, &A[(j+k)*N + i], sizeof(complex_t) * bsize );
    // transpose block in place
    for (unsigned k = 0; k < bsize - 1; k++ )
      for (unsigned l = k + 1; l < bsize; l++ ) {
        complex_t c = lb[k*N+l];
        lb[k*N+l] = lb[l*N+k];
        lb[l*N+k] = c;
      }
  }
```

Transpose of A->B, shared arrays of N*N interpreted as (N, N) matrices
FFT – Multiplication by twiddle factors

- \( Z \) : shared array of \( N \times N \) interpreted as (\( N, N \)) matrix
- multiplication of element (\( i,j \)) by \( W_{N \times N}^{ij} \), where \( W_{N \times N}^{ij} = e^{-2\pi i j / N \times N} \)

```c
void multByTwiddleFactors(ComplexArray_t Z)
{
    for (ArrayIndex_t i = 0; i < N; i++)
        upc_forall (ArrayIndex_t j = 0; j < N; j++) &Z[ i*N+j ]
    {
        double x = ( 2 * M_PI * i * j ) / ( N * N );
        double tw_re = cos(x), tw_im = -sin(x);
        Z[ i*N+j ].re = tw_re * Z[ i*N+j ].re - tw_im * Z[ i*N+j ].im;
        Z[ i*N+j ].im = tw_im * Z[ i*N+j ].re + tw_re * Z[ i*N+j ].im;
    }
}
```
EP Stream Triad

```c
upc_forall (i = 0; i < VectorSize; i++; i) {
    a[i] = b[i] + alpha * c[i];
}
```

- **Embarrassingly parallel**: performance is gated by the individual node memory bandwidth
- **Important compiler optimization**:
  - Identify shared array accesses that have affinity to the accessing thread; transform them into local accesses
- **Verification**: random sampling
- **Lines of code**: 90
# EP STREAM Triad – Performance Results

<table>
<thead>
<tr>
<th>Processors</th>
<th>Problem Size</th>
<th>Memory Used</th>
<th>GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>11,453,246,122</td>
<td>256 GB</td>
<td>1432.70</td>
</tr>
<tr>
<td>4096</td>
<td>22,906,492,245</td>
<td>500 GB</td>
<td>2865.35</td>
</tr>
<tr>
<td>8192</td>
<td>45,812,984,490</td>
<td>1 TB</td>
<td>5730.41</td>
</tr>
<tr>
<td>16384</td>
<td>91,625,968,981</td>
<td>2 TB</td>
<td>11460.65</td>
</tr>
<tr>
<td>32768</td>
<td>183,251,937,962</td>
<td>4 TB</td>
<td>22920.70</td>
</tr>
<tr>
<td>131072</td>
<td>733,007,751,850</td>
<td>16 TB</td>
<td>91627.49</td>
</tr>
</tbody>
</table>