Our submission at a glance

- **2 ½ programming languages**
  - UPC and Coarray Fortran; Spiral for FFT

- **Two platforms**
  - Power clusters (UPC + CAF)
  - Blue Gene/P (UPC only)

- **One completely rewritten benchmark**
  - HPL with tiled array library (no UPC language extensions)

- **One new benchmark**
  - K-means clustering
Machines and compilers

http://www.alphaworks.ibm.com/tech/upccompiler

**xIUPC**
- Status: alpha
- UPC moving towards standardization

**xICAf**
- Status: internal prototype
- Prioritized subsets in future Fortran releases

**Spiral**: academic project, supporting commercial company

http://spiralgen.com

<table>
<thead>
<tr>
<th>NCSA BluePrint cluster</th>
<th>IBM Poughkeepsie scaling cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 nodes x 16 threads</td>
<td>32 nodes x 128 SMT threads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IBM TJ Watson Res. Ctr. WatsonShaheen</th>
<th>Argonne Nat'l Labs Surveyor + Intrepid</th>
<th>Lawrence Livermore Nat'l Labs Dawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 racks x 4096 CPUs</td>
<td>32 of 40 racks</td>
<td>up to 16 racks</td>
</tr>
</tbody>
</table>
Benchmark descriptions 1/3: HPL, RA, Stream UPC and Coarray Fortran

- CAF versions of stream, RA

```fortran
do i=1,updates_per_image
  a     = stream_next(a)
  image = iand(ishft(a, shift), mask)
  idx   = iand(a, index_mask)
  T(idx)[image+1]=ieor(T(idx)[image+1], a)
end do
```

```fortran
subroutine stream_triad(a,b,c,alpha)
  real, intent(in) :: b(N)[*], c(N)[*]
  real, intent(out) :: a(N)[*]
  a(:) = b(:) + alpha * c(:)
end subroutine
```

- Tiled array library (UPC and CAF)
  - Designed for DLAs
  - Uses one-sided and collective communication primitives
  - UPC HPL implemented; CAF in progress
Benchmark descriptions 2/3: FFT
UPC, CAF and Spiral

- UPC, CAF: two variants of code
- Spiral: FFT in tensor language, generate code

- UPC and CAF: with Alltoall
  - local DFT (contiguous)
  - collect buffers into tiles
  - global transpose (using Alltoall)
  - local transpose tiles
  - un-tile buffers
  - twiddles etc.

- UPC and CAF: naive
  - local DFT (contiguous)
  - line-by-line global transpose
  - local transpose
  - twiddles etc.

- Spiral code:
  - DFT scrambles data to tiled format, performs local transpose, twiddles
  - global transpose (using Alltoall on contiguous data)
  - Use MPI + OpenMP (although have experimented with UPC also)
Benchmark descriptions 3/3: k-means clustering

- **Problem:** find few (K) representatives for large (N) set of points in (D)-dimensional Euclidian space
  - Iterative method: K x N Euclidian distances/iteration
  - Important B/A kernel (e.g. SPSS)
  - Almost Embarrassingly Parallel (no scaling issues)
  - Simple code: 3 loops (N, K and D)
    - Bandwidth-gated
    - Intractable by today's compilers
    - Worthy successor to Stream
Trouble in k-means land

```
do n1=1,N
  mindist0 = 1.0e99
  kmin0 = -1
  do k1=1,K
    d0 = 0.0
    do d1=1,D
      diff0 = pointv(n1,d1)-clusterv(k1,d1)
      d0 = d0 + diff0*diff0
    end do
    if (d0 .lt. mindist0) then
      kmin0 = k1
      mindist0 = d0
    end if
  end do
  lnearestv(n1) = kmin0
end do
```

Low D count throws off unrolling
Odd number of FP – no FMAs
Branches throw off vectorizers

Best manual solution we found:
• Lay out data in non-intuitive way (K dimension first, D dimension second)
• Unroll + fuse K-loop into D-loop
• Manually deploy vector select() statements where available
Performance summary 1/3: HPL, Stream, RA

<table>
<thead>
<tr>
<th></th>
<th>Efficiency</th>
<th>ESSL eff.</th>
<th>PF</th>
<th>Load Imb</th>
<th>Comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG/P</td>
<td>51%</td>
<td>65%</td>
<td>10%</td>
<td>5%</td>
<td>8%</td>
</tr>
<tr>
<td>Power5</td>
<td>57%</td>
<td>72%</td>
<td>10%</td>
<td>4%</td>
<td>5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>UPC naive</th>
<th>UPC opt</th>
<th>CAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG/P</td>
<td>32%</td>
<td>82%</td>
<td>N/A</td>
</tr>
<tr>
<td>Power7</td>
<td>50-60%</td>
<td>87%</td>
<td></td>
</tr>
</tbody>
</table>

UPC and CoArray Fortran RandomAccess on Power5

UPC RandomAccess on Blue Gene/P

HPL runtime composition on Blue Gene
Performance summary 2/3: FFT

6.4 TFlop/s
Performance summary 3/3: k-means

On Blue Gene/P:

- Peak BW: 3.4 GBytes/s
- Stream BW: 2.8
- K-means naive: 0.58
- K-means hand-opt: 1.22

UPC and CoArray Fortran K-means on Power7

Peak perf. around 26 GFlop/s
Productivity and elegance

- Spiral elegant and concise; generates impenetrable code
- Split-phase UPC memget caused a lot of trouble
- Coarray Fortran much less verbose than UPC
  - UPC drawback: abuse of pointers to remote objects
  - Except for operators
- UPC not quite equipped for writing libraries
  - Need to fix

Lines of code summary:

<table>
<thead>
<tr>
<th></th>
<th>HPL</th>
<th>FFT</th>
<th>RA</th>
<th>Stream</th>
<th>k-means</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPC</td>
<td>1012+595</td>
<td>369</td>
<td>165</td>
<td>186</td>
<td>~ 600</td>
</tr>
<tr>
<td>CAF</td>
<td>N/A+ 192</td>
<td>304</td>
<td>134</td>
<td>58</td>
<td>361</td>
</tr>
</tbody>
</table>
Conclusion

- **CoArray Fortran and UPC:**
  - Under development at IBM; teams work closely together
  - Emphasis on PGAS languages as library builders; interoperability with MPI a goal
  - Upcoming milestones for PERCS, Blue Waters

- **We know how to program for scalability**
  - But do not forget single/multicore performance

- **Spiral exploits domain knowledge**
  - Very concise expression (1 tensor equation)
  - Potential for absolute best performance on many architectures
Backup
## Performance summary – Blue Gene/P (UPC/Spiral)

<table>
<thead>
<tr>
<th></th>
<th>HPL GFlop/s</th>
<th>RA GUP/s</th>
<th>FFT GFlops/s</th>
<th>Spiral GFlops/s</th>
<th>Stream GBytes/s</th>
<th>k-means GBytes/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 NC</td>
<td>246</td>
<td>0.038</td>
<td>11</td>
<td>14.2</td>
<td>1.21</td>
<td>2.81</td>
</tr>
<tr>
<td>4 NC</td>
<td>1017</td>
<td>0.153</td>
<td>31</td>
<td>39.8</td>
<td></td>
<td>5.62</td>
</tr>
<tr>
<td>1 MP</td>
<td>4061</td>
<td>0.61</td>
<td>191</td>
<td>276</td>
<td></td>
<td>12.07</td>
</tr>
<tr>
<td>1 rack</td>
<td>8115</td>
<td>1.21</td>
<td></td>
<td></td>
<td></td>
<td>24.12</td>
</tr>
<tr>
<td>2 racks</td>
<td>16214</td>
<td>2.41</td>
<td>550</td>
<td>625</td>
<td></td>
<td>48.30</td>
</tr>
<tr>
<td>4 racks</td>
<td>31626</td>
<td>4.82</td>
<td></td>
<td>691</td>
<td></td>
<td>96.49</td>
</tr>
<tr>
<td>8 racks</td>
<td></td>
<td></td>
<td>1479</td>
<td>1370</td>
<td></td>
<td>193.09</td>
</tr>
<tr>
<td>16 racks</td>
<td>112904</td>
<td></td>
<td></td>
<td>2890</td>
<td></td>
<td>360.10</td>
</tr>
</tbody>
</table>

**Summary:** 52% flat linear

**IBM TJ Watson Res. Ctr.**
**WatsonShaheen**
All 4 racks

**Argonne Nat'l Labs**
**Surveyor + Intrepid**
up to 16 racks

**Lawrence Livermore Nat'l Labs**
**Dawn**
up to 16 racks

82%
APGAS: one library to run them all

- **Support for UPC and CAF**
  - shared arrays; pointers-to-shared; locks; optimized collectives

- **Support for X10**
  - Asyncs & activities; remote references

- **Multiplatform**
  - Power, BG, Intel, Sun etc.
  - LAPI (IB, HPS), DCMF (BG), MX (Myrinet), TCP/IP sockets

- **Interoperable**
  - MPI
Bisection bandwidth calculation (Blue Gene/P)

<table>
<thead>
<tr>
<th># Nodes</th>
<th>Torus</th>
<th>Bisection (links)</th>
<th>BW (GB/s)</th>
<th>GUPS limit</th>
<th>FFT limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>4x4x2</td>
<td>32</td>
<td>13.6</td>
<td>0.32</td>
<td>39</td>
</tr>
<tr>
<td>128</td>
<td>4x8x4</td>
<td>128</td>
<td>55</td>
<td>1.30</td>
<td>176</td>
</tr>
<tr>
<td>1024</td>
<td>8x8x16</td>
<td>256</td>
<td>109</td>
<td>2.6</td>
<td>870</td>
</tr>
<tr>
<td>2048</td>
<td>8x16x16</td>
<td>512</td>
<td>217</td>
<td>5.2</td>
<td>1741</td>
</tr>
<tr>
<td>4096</td>
<td>16x16x16</td>
<td>1024</td>
<td>434</td>
<td>10.3</td>
<td>3482</td>
</tr>
</tbody>
</table>

Torus bisection = smallest diameter x 2 (torus) x 2 (half traffic)
Bisection Bandwidth = Bisection x 0.42 GB/s/link
GUPS limit = Bisection bandwidth / 42 bytes/packet
FFT Gflops = flops * BW / Bytes
FFT Gflops = 5* log(N) * N * N * Bandwidth / 3 * N * N * sizeof(cplx)