Team Collectives and BG/P Results

Rajesh Nishtala, Yili Zheng and Paul Hargrove

Future Technologies Group
Lawrence Berkeley National Laboratory
University of California, Berkeley
Observations
Observations

• Performance gains delivered through increasing concurrency rather than clock rates

• Application scalability is essential for future performance improvements

• 100,000s of processors will be the norm in the very near future

• Maximize the use of available resources
  • Leverage communication/communication and communication/computation overlap
Observations

- Performance gains delivered through increasing concurrency rather than clock rates

- Application scalability is essential for future performance improvements

- 100,000s of processors will be the norm in the very near future

- Maximize the use of available resources
  - Leverage communication/communication and communication/computation overlap

- Systems will favor many slower power efficient processors to fewer faster power inefficient one

- Light-weight communication and runtime system to minimize software overhead

- Close semantic match to underlying hardware
Overview
Overview

• Discuss our new port of GASNet, the communication subsystem for the Berkeley UPC compiler, to the BlueGene/P

• Outline the key differences found between one and two sided communication and their applicability to modern networks
Overview

• Discuss our new port of GASNet, the communication subsystem for the Berkeley UPC compiler, to the BlueGene/P

• Outline the key differences found between one and two sided communication and their applicability to modern networks

• Show how the microbenchmark performance advantages can translate to real applications

• Chose the communication bound NAS FT benchmark as the case study
Overview

• Discuss our new port of GASNet, the communication subsystem for the Berkeley UPC compiler, to the BlueGene/P

• Outline the key differences found between one and two sided communication and their applicability to modern networks

• Show how the microbenchmark performance advantages can translate to real applications

• Chose the communication bound NAS FT benchmark as the case study

• Thesis Statement:

  • The one-sided communication model found in GASNet is a better semantic fit to modern highly concurrent systems by better leveraging features such as RDMA and thus allowing applications to realize better scaling.
BlueGene/P Overview

- Representative example for future highly concurrent systems
- Compute node: 4 cores running at 850 MHz w/ 2GB of RAM and 13.6 GB/s between main memory and the cores
- Total cores = (32 nodes / node card) x (32 node cards / rack) x (upto 72 racks)
- Different networks for different tasks
  - 3D Torus for general point-to-point communication (5.1 GB/s per node)
  - Global Interrupt network for Barriers (1.3 us for 72 racks)
  - Global Collective Network for One-to-Many broadcast or Many-to-one reductions (0.85 GB/s per link)
One-Sided versus Two-Sided Communication

- **One-sided put (i.e. GASNet)**
  - dest addr
  - data payload

- **Two-sided send/recv (i.e. MPI)**
  - msg id
  - data payload

- **NIC**
  - pre-posted recv

- **Host cores**
- **Memory**

- **Advantages of One-sided put/get**
  - Directly transfer data without interrupting host cores
  - Message contains the information about the remote address to find out where to directly put the data
  - CPU need not be involved if NIC supports Remote Direct Memory Access (RDMA)
  - Synchronization is decoupled from the data movement.

- **Disadvantages of Two-sided send/recv**
  - Requires rendez-vous with host cores to agree where the data needs to be put before RDMA can be used
  - Bounce buffers can also be used for small enough message but slow serial can make it prohibitively expensive
  - Most modern networks provide RDMA functionality, so why not just use it directly?
GASNet Overview

• Portable and high performance runtime system for many different PGAS Languages

• Projects: Berkeley UPC, GCC-UPC, Titanium, Rice Co-Array Fortran, Cray Chapel, Cray UPC & Co-Array Fortran and many other experimental projects

• To the best of our knowledge, first PGAS compiler on BlueGene/P!

• Supported Networks: BlueGene/P (DCMF), Infiniband (VAPI and IBV), Cray XT (Portals), Quadrics (Elan), Myrinet (GM), IBM LAPI, SHMEM, SiCortex (soon to be released), UDP, MPI

• 100% open source and under BSD license

• Features:
  • Multithreaded (works on VN, Dual, or SMP modes)
  • Provides efficient nonblocking puts and gets
    • Often just a thin wrapper around hardware puts and gets
    • Also support for Vector, Index, and Strided (VIS) operations
  • Provides rich Active Messaging API
  • Provides Nonblocking Collective Communication
    • Collectives will soon be automatically tuned
GASNet Latency Performance

- GASNet implemented on top of Deep Computing Messaging Framework (DCMF)
  - Lower level than MPI
  - Provides Puts, Gets, AMSend, and Collectives
- Point-to-point ping-ack latency performance
  - N-byte transfer w/ 0 byte acknowledgement
  - GASNet takes advantage of DCMF remote completion notification
- Minimum semantics needed to implement the UPC memory model
  - Almost a factor of two difference until 32 bytes
  - Indication of better semantic match to underlying communication system
Each node has six 850MB/s* bidirectional link
Vary number of links used from 1 to 6
Initiate a series of nonblocking puts on the links (round-robin)
Communication/communication overlap
Both MPI and GASNet asymptote to the same bandwidth
GASNet outperforms MPI at midrange message sizes
Lower software overhead implies more efficient message injection
GASNet avoids rendezvous to leverage RDMA

*Kumar et. al showed the maximum achievable bandwidth for DCMF transfers is 748 MB/s per link so we use this as our peak bandwidth
See “The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer”, Kumar et al. ICS08
Case Study: NAS FT Benchmark

1D Partition (4 threads)

2D Partition (4x4 threads)

NY/TY
NZ/TZ
NX
NY
NZ/T
NX
Case Study: NAS FT Benchmark

• Perform a large 3D FFT
  • Used in many areas of computational science
    • Molecular dynamics, CFD, image processing, signal processing, astrophysics, etc.
  • Representative of a class of communication intensive algorithms
    • Requires parallel many-to-many communication
    • Stresses communication subsystem
    • Limited by bandwidth (namely bisection bandwidth) of the network
Case Study: NAS FT Benchmark

- Perform a large 3D FFT
  - Used in many areas of computational science
    - Molecular dynamics, CFD, image processing, signal processing, astrophysics, etc.
  - Representative of a class of communication intensive algorithms
    - Requires parallel many-to-many communication
    - Stresses communication subsystem
    - Limited by bandwidth (namely bisection bandwidth) of the network
- Building on our previous work, we perform a 2D partition of the domain
  - Requires two rounds of communication rather than one
  - Each processor communicates in two rounds with $O(\sqrt{\text{T}})$ threads in each
Our Terminology

- Domain is NX columns by NY rows by NZ planes
- We overlay a TY x TZ processor grid (i.e. X is only contiguous dimension)
- Plane: An NX columns by NY rows that is shared amongst a team of TY processors
- Slab: An NX columns by NY/TY rows of elements that is entirely on one thread
  - Each thread owns NZ/TZ slabs
- Packed Slab: An NX columns by NY/TY rows by NZ/TZ rows
  - All the data a particular thread owns
3D-FFT Algorithm

Each processor owns a row of 4 squares
(16 processors in example)
3D-FFT Algorithm

- Perform a 3D FFT (as part of NAS FT) across a large rectangular prism
- Perform an FFT in each of the 3 Dimensions
- Need to Team-Exchange for other 2/3 dimensions for a 2-D processor layout
- Performance limited by bisection bandwidth of the network

Each processor owns a row of 4 squares (16 processors in example)
3D-FFT Algorithm

- Perform a 3D FFT (as part of NAS FT) across a large rectangular prism
- Perform an FFT in each of the 3 Dimensions
- Need to Team-Exchange for other 2/3 dimensions for a 2-D processor layout
- Performance limited by bisection bandwidth of the network

Algorithm:
- Perform FFT across the rows

Each processor owns a row of 4 squares
(16 processors in example)
3D-FFT Algorithm

- Perform a 3D FFT (as part of NAS FT) across a large rectangular prism
- Perform an FFT in each of the 3 Dimensions
- Need to Team-Exchange for other 2/3 dimensions for a 2-D processor layout
- Performance limited by bisection bandwidth of the network

Algorithm:
- Perform FFT across the rows
- Do an exchange within each plane

Each processor owns a row of 4 squares
(16 processors in example)
3D-FFT Algorithm

- Perform a 3D FFT (as part of NAS FT) across a large rectangular prism
- Perform an FFT in each of the 3 Dimensions
- Need to Team-Exchange for other 2/3 dimensions for a 2-D processor layout
- Performance limited by bisection bandwidth of the network

Algorithm:
- Perform FFT across the rows
- Do an exchange within each plane
- Perform FFT across the columns

Each processor owns a row of 4 squares
(16 processors in example)
3D-FFT Algorithm

- Perform a 3D FFT (as part of NAS FT) across a large rectangular prism
- Perform an FFT in each of the 3 Dimensions
- Need to Team-Exchange for other 2/3 dimensions for a 2-D processor layout
- Performance limited by bisection bandwidth of the network

Algorithm:
- Perform FFT across the rows
- Do an exchange within each plane
- Perform FFT across the columns
- Do an exchange across planes

Each processor owns a row of 4 squares (16 processors in example)
3D-FFT Algorithm

- Perform a 3D FFT (as part of NAS FT) across a large rectangular prism
- Perform an FFT in each of the 3 Dimensions
- Need to Team-Exchange for other 2/3 dimensions for a 2-D processor layout
- Performance limited by bisection bandwidth of the network

Algorithm:
- Perform FFT across the rows
- Do an exchange within each plane
- Perform FFT across the columns
- Do an exchange across planes
- Perform FFT across the last dimension

Each processor owns a row of 4 squares (16 processors in example)
3D FFT: Packed Slabs

2D Partition
(4x4 threads)

NZ/TZ

NY/TY

NX
3D FFT: Packed Slabs

- Perform communication and computation in two distinct phases
  - First perform the computation for all the rows in X-dimension
    - Communication system is idle during this time
  - Perform a Transpose to relocalize the Y-dimension
    - Requires Packing and Unpacking
    - Performed across all the processors with the same color
  - Perform the FFT for all the columns
  - Perform a transpose to relocalize the Z-dimension
  - Perform the final set of FFTs

<table>
<thead>
<tr>
<th>Message Size Round 1</th>
<th>(NZ/TZ) × (NY/TY) × (NX/TY) elements</th>
</tr>
</thead>
<tbody>
<tr>
<td># Messages in Round 1</td>
<td>TY</td>
</tr>
<tr>
<td>Message Size Round 2</td>
<td>(NZ/TZ) × (NX/TY) × (NY/TZ) elements</td>
</tr>
<tr>
<td># Messages in Round 2</td>
<td>TZ</td>
</tr>
</tbody>
</table>
3D FFT: Packed Slabs

- Perform communication and computation in two distinct phases
  - First perform the computation for all the rows in X-dimension
    - Communication system is idle during this time
  - Perform a Transpose to relocalize the Y-dimension
    - Requires Packing and Unpacking
    - Performed across all the processors with the same color
  - Perform the FFT for all the columns
  - Perform a transpose to relocalize the Z-dimension
  - Perform the final set of FFTs
- As per conventional wisdom, data is packed to increase message size
  - Only exploits communication/communication overlap during the transpose
  - MPI implements transpose as in memory data movement plus one call to MPI_Alltoall() for each round
    - Minimum number of calls to MPI

<table>
<thead>
<tr>
<th>Message Size</th>
<th>(NZ/TZ) × (NY/TY) × (NX/TY) elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td></td>
</tr>
<tr>
<td># Messages in Round 1</td>
<td>TY</td>
</tr>
<tr>
<td>Message Size</td>
<td>(NZ/TZ) × (NX/TY) × (NY/TZ) elements</td>
</tr>
<tr>
<td>Round 2</td>
<td></td>
</tr>
<tr>
<td># Messages in Round 2</td>
<td>TZ</td>
</tr>
</tbody>
</table>
3D FFT: Slabs

- Observation:
  - After one of the NZ/TZ planes of row FFTs is done we can start transferring the data
  - Allows communication/communication overlap and communication/computation overlap

Algorithm sketch:
1. for each of the NZ/TZ planes
   1.1. perform all NY/TY row FFTs (len NX)
   1.2. pack data for this plane
   1.3. Initiate nonblocking all-to-all
2. wait for all all-to-alls to finish
3. unpack data
4. for each of the NZ/TZ planes
   4.1. perform all NX/TY row FFTs (len NY)
   4.2. pack data for this plane
   4.3. Initiate nonblocking all-to-all
5. wait for all all-to-alls to finish
6. unpack data
7. perform last round of (NY/TZ) (NX/TY) FFTs (len NZ)

- Without nonblocking collectives in MPI we implement this through point-to-point operations
- UPC and MPI versions have the same communication schedules
Strong Scaling

- Fix problem size at 2k x 1k x 1k and run in VN mode
- Upto 4 racks of BG/P with 4 processes per node
- Analytic upper bound calculates megaflop rate based on time needed to transfer domain across the bisection
- Kink at 2048 cores indicates where 3D Torus is completed
- MPI Packed Slabs scales better than MPI Slabs
- Benefit of comm/comp. overlap outweighed by extra messages
- UPC (i.e. GASNet) Slabs consistently outperforms MPI
- Lower software overhead enables better overlap
- Outperforms Slabs by mean of 63% and Packed Slabs by mean of 37%
Weak Scaling

- Scale problem size with the number of cores
- computation for FFT scales as $O(N \log N)$ so thus flops don’t scale linearly
- UPC Slabs scales better than strong scaling benchmark
- Message size gets too small at high concurrency for strong scaling and becomes hard to utilize overlap
- MPI Packed Slabs outperforms MPI Slabs (most of the time)
- Again indicates that overlapping communication/computation is not a fruitful optimization for MPI
- UPC achieves **1.93** Teraflops while best MPI achieves **1.37** Teraflops
- 40% improvement in performance at 16k cores.
Performance Comparison

For a 4k x 2k x 2k cube on 128 x 128 processor grid:
- Packed Slabs Message Size: 128kB
- Slabs Message Size: 8kB
Performance Comparison

For a 4k x 2k x 2k cube on 128 x 128 processor grid:
Packed Slabs Message Size: 128kB
Slabs Message Size: 8kB

Both Asymptote to the same bandwidth for 128kB messages
Performance Comparison

For a 4k x 2k x 2k cube on 128 x 128 processor grid:
- Packed Slabs Message Size: 128kB
- Slabs Message Size: 8kB

GASNet gets 24% higher bandwidth for two links and 39% higher for four links (six link is not applicable because of thread layout)

Both Asymptote to the same bandwidth for 128kB messages
Performance Breakdown

• Performance breakdown for weak scaling at 16k cores
• Major difference in performance is from synchronous communication time
• Lower bandwidth for smaller messages is offset by effectively overlapping communication/computation
  • Key performance tradeoff: Higher communication/computation overlap potential for lower message bandwidth
    • Until nonblocking collectives are found in MPI we also give up the use of collective operations
• Results show cumulative effects of allowing communication/computation overlap and one-sided communication through GASNet
Conclusions

- We have ported GASNet and Berkeley UPC to the BlueGene/P
  - Uses native DCMF for communication
  - Microbenchmarks show better performance than MPI for both latency and bandwidth
  - One-sided communication model is a better semantic fit to the network
- Use NAS FT benchmark as a case-study
  - Represent a class of communication-bound problems
  - Compare two algorithms:
    - Packed Slabs (only comm./comm. overlap)
    - Slabs (both comm./comp. overlap and comm./comm. overlap)
  - UPC (GASNet) consistently outperforms MPI versions
    - Best UPC benchmark achieves 1.93 Teraflops across 16k cores
    - Best MPI achieves 1.37 Teraflops (a 40% improvement in performance)
Collective Communication

• Commonly used communication patterns
• Improve productivity and optimize performance
• Team -- group of participants
• Types
  – Broadcast (one to many)
  – Gather (many to one)
  – Exchange (many to many)
Parallel Matrix Multiplication

\[
\begin{pmatrix}
    c_{11} & \cdots & c_{1n} \\
    \vdots & \ddots & \vdots \\
    c_{m1} & \cdots & c_{mn}
\end{pmatrix}
= \begin{pmatrix}
    a_{11} & \cdots & a_{1n} \\
    \vdots & \ddots & \vdots \\
    a_{m1} & \cdots & a_{mn}
\end{pmatrix}
\times \begin{pmatrix}
    b_{11} & \cdots & b_{1n} \\
    \vdots & \ddots & \vdots \\
    b_{m1} & \cdots & b_{mn}
\end{pmatrix}
\]
Data Partitioning

Global Matrix View

Distributed Matrix Storage

Computer I

Computer II

Computer III

Computer IV
# Team Collective Communication

- **Row team broadcast**
  
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

- **Column team broadcast**
  
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>
Software Organization

- UPC Collective API
- Other PGAS Collective API
- GASNet Collective Communication API
- Portable Implementation
- Native Implementation
- Point-to-point Comm. driver
- Collective Comm. driver
- Network Hardware
Barrier Performance on BG/P

- Use BG/P global barrier hardware
- Speed:
  - GI > Torus > Binomial
- Applicability:
  - GI < Torus < Binomial
Broadcast Performance on BG/P

- Use BG/P special broadcast networks
- Speed:
  - Tree > Torus > Point2Point
- Applicability:
  - Tree < Torus < Point2Point

(256 cores / 64 nodes, virtual node mode)
All-to-all Performance on BG/P

Latency (micro sec.)

Message Size (Bytes per node)

(256 cores / 64 nodes, virtual node mode)
Dense Linear Algebra Performance on BG/P

Parallel Matrix Multiplication
(256 core BlueGene/P)

- PBLAS (MPI): 458 GFlops
- UPC hand-roll: 580 GFlops
- UPC collective: 625 GFlops

Parallel Cholesky Factorization
(256 core BlueGene/P)

- ScaLapack (MPI): 202 GFlops
- UPC hand-roll: 212 GFlops
- UPC collective: 220 GFlops
FFT Performance on BG/P (weak scaling)

<table>
<thead>
<tr>
<th></th>
<th>Slabs</th>
<th>Slabs (Collective)</th>
<th>Packed Slabs (Collective)</th>
<th>MPI Packed Slabs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFlops</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D/8</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>D/4</td>
<td>256</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>D/2</td>
<td>1024</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>D</td>
<td>2048</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
</tr>
<tr>
<td>D*2</td>
<td>4096</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
</tr>
<tr>
<td>D*4</td>
<td>8192</td>
<td>4096</td>
<td>8192</td>
<td>16384</td>
</tr>
<tr>
<td>D*8</td>
<td>16384</td>
<td>8192</td>
<td>16384</td>
<td>32768</td>
</tr>
<tr>
<td>D*16</td>
<td>32768</td>
<td>16384</td>
<td>32768</td>
<td></td>
</tr>
</tbody>
</table>
Summary

• **High performance and enhanced productivity**
  • Beat standard numerical linear algebra package
    – Parallel matrix multiplication: **36%** faster (256 cores)
    – Parallel Cholesky factorization: **9%** faster (256 cores)
  • Improved FFT performance
    – Weak scaling: **38%** over MPI (16K cores)
    – Strong Scaling: **20%** over MPI (16K cores)
  • Native GASNet collective implementation for large scale BlueGene systems
    – Broadcast: **3-4** times speedup
    – Exchange: **2-3** times speedup
Backup Slides
## Comparison of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Packed Slabs</th>
<th>Slabs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message Size in Round 1</strong></td>
<td>$(NZ/TZ) \times (NY/TY) \times (NX/TY)$ elements</td>
<td>$(NY/TY) \times (NX/TY)$ elements</td>
</tr>
<tr>
<td><strong># Messages in Round 1</strong></td>
<td>TY</td>
<td>$(NZ/TZ) \times TY$</td>
</tr>
<tr>
<td><strong>Message Size in Round 2</strong></td>
<td>$(NZ/TZ) \times (NX/TY) \times (NY/TZ)$ elements</td>
<td>$(NX/TY) \times (NY/TZ)$ elements</td>
</tr>
<tr>
<td><strong># Messages in Round 2</strong></td>
<td>TZ</td>
<td>$(NZ/TZ) \times TZ$</td>
</tr>
</tbody>
</table>
Appendix (Packed Slabs)

Algorithm 1 FFT Packed Slabs

1: Let myPlane = MYTHREAD / TY
2: Let myRow = MYTHREAD % TY
3: Let teamY = all threads who have same value of myPlane
4: Let teamZ = all threads who have same value of myRow
5: for plane = 0 to \(\frac{NZ}{TY}\) do
6:     for row = 0 to \(\frac{NX}{TY}\) do
7:         do 1D FFT of length NX
8:     end for
9: end for
10: Pack the slabs together
11: Do Alltoall on teamY
12: Unpack the slabs to make Y dimension contiguous
13: for plane = 0 to \(\frac{NZ}{TZ}\) do
14:     for row = 0 to \(\frac{NY}{TY}\) do
15:         do 1D FFT of length NY
16:     end for
17: end for
18: Pack the slabs together
19: Do Alltoall on teamZ
20: Unpack the slabs to make the Z dimension contiguous
21: for plane = 0 to \(\frac{NY}{TZ}\) do
22:     for row = 0 to \(\frac{NX}{TY}\) do
23:         do 1D FFT of length NZ
24:     end for
25: end for
Algorithm 2 FFT Slabs

1: Let myPlane = MYTHREAD / TY
2: Let myRow = MYTHREAD % TY
3: For MPI Prepost all recvs for First Communication Round
4: BARRIER
5: for plane = 0 to $\frac{NZ}{TY}$ do
6: for row = 0 to $\frac{NY}{TY}$ do
7: do 1D FFT of length NX
8: end for
9: Pack the data for this plane
10: for $t = 1; t \leq TY; t = t + 1$ do
11: initiate communication to thread $myPlane \times TY + (t + myRow) \% TY$
12: end for
13: end for
14: Wait for all communication to finish
15: Unpack all the data to make Y dimension contiguous
16: For MPI Prepost all recvs for Second Communication Round
17: BARRIER
18: for plane = 0 to $\frac{NZ}{TY}$ do
19: for row = 0 to $\frac{NY}{TY}$ do
20: do 1D FFT of length NY
21: end for
22: Pack the data for this plane
23: for $t = 1; t \leq TZ; t = t + 1$ do
24: initiate communication to thread $((t + myPlane) \% TZ) \times TY + myRow$
25: end for
26: end for
27: Wait for all communication to finish
28: Unpack all the data to make Z dimension contiguous
29: for plane = 0 to $\frac{NY}{TY}$ do
30: for row = 0 to $\frac{NZ}{TY}$ do
31: do 1D FFT of length NZ
32: end for
33: end for
## Node Configurations

<table>
<thead>
<tr>
<th>Node Count</th>
<th>Core Count</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>T</th>
<th>TY x TZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>256</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16 x 16</td>
</tr>
<tr>
<td>128</td>
<td>512</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>16 x 32</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>32 x 32</td>
</tr>
<tr>
<td>512</td>
<td>2048</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>64 x 32</td>
</tr>
<tr>
<td>1024</td>
<td>4096</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>64 x 64</td>
</tr>
<tr>
<td>2048</td>
<td>8192</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>4</td>
<td>64 x 128</td>
</tr>
<tr>
<td>4096</td>
<td>16384</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>4</td>
<td>128 x 128</td>
</tr>
</tbody>
</table>
# UPC Semaphore (P2P sync)

```c
#include <upc.h>

void upc_sem_post(upc_sem_t *s);
void upc_sem_postN(upc_sem_t *s, size_t n); /* only valid for INTEGER sems */

void upc_sem_wait(upc_sem_t *s);
void upc_sem_waitN(upc_sem_t *s, size_t n); /* only valid for INTEGER sems */

int upc_sem_try(upc_sem_t *s);
int upc_sem_tryN(upc_sem_t *s, size_t n); /* only valid for INTEGER sems */
```

Description:

`upc_sem_post(N)`: atomically increment the logical value of semaphore `s` by 1 (N)

`upc_sem_wait(N)`: suspend the calling thread until the logical value of semaphore `s` is \( v = 1 \) (N), then atomically decrement the value by that amount and return. If multiple threads are simultaneously blocked inside `wait`, (only valid for `UPC_SEM_MCONSUMER`) then it is undefined the order in which they will be serviced (no fairness guarantees)

`upc_sem_try(N)`: A non-blocking variant of `upc_sem_wait(N)`. Attempt to perform a `upc_sem_wait(N)` on `s`. If the operation can succeed immediately, perform it and return non-zero to indicate success. Otherwise, return zero to indicate failure.

`upc_sem_post(N)` implies a `upc_fence` operation upon entry to the function, `upc_sem_wait(N)` implies a `upc_fence` operation immediately before exiting, and `upc_sem_try(N)` implies a `upc_fence` operation immediately before a successful completion.
UPC Semaphore (Memput)

#include <upc.h>

void upc_memput_signal (shared void *dst, const void *src, size_t nbytes, upc_sem_t *s, size_t n);
void upc_memput_signal_async(shared void *dst, const void *src, size_t nbytes, upc_sem_t *s, size_t n);

Description:

Perform a memput with the same data movement semantics as upc_memput, and increment s by n when the transfer is complete. Requires upc_threadof(s) == upc_threadof(dst).

Both functions MAY return on the initiator before the transfer is complete at the target — the semaphore on the target will be atomically incremented by n when the transfer is globally complete.

No explicit notifications or guarantees are provided to the initiator regarding the completion of the transfer at the target (remote completion).

bupc_memput_signal returns as soon as the source memory is safe to overwrite (ie it blocks for local completion of the transfer), whereas bupc_memput_signal_async MAY return earlier, while the source memory is still in use (and therefore not safe to overwrite). Callers of bupc_memput_signal_async are responsible for enforcing their own synchronization from the target thread to the initiator thread, in order to decide when the source memory is safe to overwrite.