Chapel: HPCC Benchmarks

Brad Chamberlain
Cray Inc.

CSEP 524
May 20, 2010
HPC Challenge (HPCC)

- **Class 2**: “most productive”
  - **Judged on**: 50% performance 50% elegance
  - **Four recommended benchmarks**: STREAM, RA, FFT, HPL
  - **Use of library routines**: discouraged
    (there’s also class 1: “best performance”; Cray won 3 of 4 this year)

- **Why you might care:**
  - many (correctly) downplay the top-500 as ignoring important things
  - HPCC takes a step in the right direction and subsumes the top 500

- **Historically**: the judges have “split the baby” for class 2
  - **2005**: *tie*: Cray (MTA-2) and IBM (UPC)
  - **2006**: *overall*: MIT (Cilk); *performance*: IBM (UPC); *elegance*: Mathworks (Matlab); *honorable mention*: Chapel and X10
  - **2007**: *research*: IBM (X10); *industry*: Int. Supercomp. (Python/Star-P)
  - **2008**: *performance*: IBM (UPC/X10);
    *productive*: Cray (Chapel), IBM (UPC/X10), Mathworks (Matlab)
  - **2009**: *performance*: IBM (UPC+X10);
    *elegance*: Cray (Chapel)
HPC Challenge: Chapel Entries (2008-2009)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>2008</th>
<th>2009</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global STREAM</td>
<td>1.73 TB/s (512 nodes)</td>
<td>10.8 TB/s (2048 nodes)</td>
<td>6.2x</td>
</tr>
<tr>
<td>EP STREAM</td>
<td>1.59 TB/s (256 nodes)</td>
<td>12.2 TB/s (2048 nodes)</td>
<td>7.7x</td>
</tr>
<tr>
<td>Global RA</td>
<td>0.00112 GUPs (64 nodes)</td>
<td>0.122 GUPs (2048 nodes)</td>
<td>109x</td>
</tr>
<tr>
<td>Global FFT</td>
<td>single-threaded single-node</td>
<td>multi-threaded multi-node</td>
<td>multi-node parallel</td>
</tr>
<tr>
<td>Global HPL</td>
<td>single-threaded single-node</td>
<td>multi-threaded single-node</td>
<td>single-node parallel</td>
</tr>
</tbody>
</table>

All timings on ORNL Cray XT4:
• 4 cores/node
• 8GB/node
• no use of library routines
HPCC STREAM and RA

- **STREAM Triad**
  - compute a distributed scaled-vector addition
    - $a = b + \alpha \cdot c$ where $a$, $b$, $c$ are vectors
  - embarrassingly parallel
  - stresses local memory bandwidth

- **Random Access (RA)**
  - make random xor-updates to a distributed table of integers
  - stresses fine-grained communication, updates (in its purest form)
Introduction to STREAM Triad

Given: \( m \)-element vectors \( A, B, C \)

Compute: \( \forall i \in 1..m, A_i = B_i + \alpha \cdot C_i \)

Pictorially:
Introduction to STREAM Triad

Given: $m$-element vectors $A$, $B$, $C$

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

Pictorially (in parallel):

\[ A \]
\[ B \]
\[ C \]
\[ \alpha \]

\[ = \quad = \quad = \quad = \quad = \quad = \]
\[ + \quad + \quad + \quad + \quad + \quad + \]
\[ * \quad * \quad * \quad * \quad * \quad * \]
STREAM Triad in Chapel

```chapel
const ProblemSpace: domain(1, int(64)) = [1..m];

var A, B, C: [ProblemSpace] real;

forall (a, b, c) in (A, B, C) do
  a = b + alpha * c;
```
STREAM Triad in Chapel

```plaintext
const BlockDist = new Block1D(bbox=[1..m], tasksPerLocale=...);

const ProblemSpace: domain(1, int(64)) dmapped BlockDist = [1..m];

var A, B, C: [ProblemSpace] real;

forall (a, b, c) in (A, B, C) do
    a = b + alpha * c;
```
EP-STREAM in Chapel

- Chapel’s *multiresolution design* also permits users to code in an SPMD style like the MPI version:

```chapel
var localGBs: [LocaleSpace] real;

coforall loc in Locales do
  on loc {
    const myProblemSpace: domain(1, int(64))
      = BlockPartition(ProblemSpace, here.id, numLocales);
    var myA, myB, myC: [myProblemSpace] real(64);
    const startTime = getCurrentTime();
    local {
      for (a, b, c) in (myA, myB, myC) do
        a = b + alpha * c;
      }
    const execTime = getCurrentTime() - startTime;
    localGBs(here.id) = timeToGBs(execTime);
  }

const avgGBs = (+ reduce localGBs) / numLocales;
```
## Experimental Platform

<table>
<thead>
<tr>
<th>machine characteristic</th>
<th>platform 1</th>
<th>platform 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>Cray XT4</td>
<td>Cray CX1</td>
</tr>
<tr>
<td>location</td>
<td>ORNL</td>
<td>Cray Inc.</td>
</tr>
<tr>
<td># compute nodes/locales</td>
<td>7,832</td>
<td>8</td>
</tr>
<tr>
<td>processor</td>
<td>2.1 GHz AMD Opteron</td>
<td>3 GHz Intel Xeon</td>
</tr>
<tr>
<td># cores per locale</td>
<td>4</td>
<td>2 × 4</td>
</tr>
<tr>
<td>total usable RAM per locale (as reported by /proc/meminfo)</td>
<td>7.68 GB</td>
<td>15.67 GB</td>
</tr>
<tr>
<td>STREAM Triad problem size per locale</td>
<td>85,985,408</td>
<td>175,355,520</td>
</tr>
<tr>
<td>STREAM Triad memory per locale</td>
<td>1.92 GB</td>
<td>3.92 GB</td>
</tr>
<tr>
<td>STREAM Triad percent of available memory</td>
<td>25.0%</td>
<td>25.0%</td>
</tr>
<tr>
<td>RA problem size per locale</td>
<td>$2^{28}$</td>
<td>$2^{29}$</td>
</tr>
<tr>
<td>RA updates per locale</td>
<td>$2^{19}$</td>
<td>$2^{24}$</td>
</tr>
<tr>
<td>RA memory per locale</td>
<td>2.0 GB</td>
<td>4.0 GB</td>
</tr>
<tr>
<td>RA percent of available memory</td>
<td>26.0%</td>
<td>25.5%</td>
</tr>
</tbody>
</table>
STREAM Performance: Chapel vs. MPI (2008)

Performance of HPCC STREAM Triad (Cray XT4)

- 2008 Chapel Global TPL=1
- 2008 Chapel Global TPL=2
- 2008 Chapel Global TPL=3
- 2008 Chapel Global TPL=4
- MPI EP PPN=1
- MPI EP PPN=2
- MPI EP PPN=3
- MPI EP PPN=4

Number of Locales

GB/s
STREAM Performance: Chapel vs. MPI (2009)

Performance of HPCC STREAM Triad (Cray XT4)

- 2008 Chapel Global TPL=1
- 2008 Chapel Global TPL=2
- 2008 Chapel Global TPL=3
- 2008 Chapel Global TPL=4
- MPI EP PPN=1
- MPI EP PPN=2
- MPI EP PPN=3
- MPI EP PPN=4
- Chapel Global TPL=1
- Chapel Global TPL=2
- Chapel Global TPL=3
- Chapel Global TPL=4
- Chapel EP TPL=4

Number of Locales

GB/s
Introduction to Random Access (RA)

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)
Compute: $N_U$ random updates to the table using bitwise-xor
Pictorially:
Introduction to Random Access (RA)

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)

Compute: $N_U$ random updates to the table using bitwise-xor

Pictorially:
Introduction to Random Access (RA)

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)

Compute: $N_U$ random updates to the table using bitwise-xor

Pictorially:

= 21 $\Rightarrow$ xor the value 21 into $T_{(21 \mod m)}$

repeat $N_U$ times
Introduction to Random Access (RA)

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)

Compute: $N_U$ random updates to the table using bitwise-xor

Pictorially (in parallel):
Introduction to Random Access (RA)

Given: \( m \)-element table \( T \) (where \( m = 2^n \) and initially \( T_i = i \))

Compute: \( N_U \) random updates to the table using bitwise-xor

Pictorially (in parallel):

Random Numbers
Not actually generated using lotto ping-pong balls!
Instead, implement a pseudo-random stream:
• \( k \)th random value can be generated at some cost
• given the \( k \)th random value, can generate the \((k+1)\)st much more cheaply
Introduction to Random Access (RA)

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)
Compute: $N_U$ random updates to the table using bitwise-xor

Pictorially (in parallel):

Conflicts
When a conflict occurs an update may be lost; a certain number of these are permitted
Introduction to Random Access (RA)

Given: $m$-element table $T$ (where $m = 2^n$ and initially $T_i = i$)

Compute: $N_U$ random updates to the table using bitwise-xor

Pictorially (in parallel):

Batching
To amortize communication overheads at lower node counts, up to 1024 updates may be precomputed per process before making any of them
RA Declarations in Chapel

```chapel
const TableDist = new Block1D(bbox=[0..m-1], tasksPerLocale=...),
    UpdateDist = new Block1D(bbox=[0..N_U-1], tasksPerLocale=...);

var T: [TableSpace] uint(64);
```
RA Computation in Chapel

```plaintext
const TableSpace: domain(1, uint(64)) dmapped TableDist = [0..m-1],
Updates: domain(1, uint(64)) dmapped UpdateDist = [0..N_U-1];

var T: [TableSpace] uint(64);

forall (_ , r) in (Updates, RAStream()) do
  on T(r&indexMask) do
    T(r&indexMask) ^= r;
```

RAComputation:

![Diagram of RA Computation with variables and flow of updates]

RAStream(): \(r_0, r_1, r_2, r_3 \ldots, r_9, \ldots, r_{17}, r_{23}, \ldots, r_{N_U-1}\)
RA Performance: Chapel (2009)

Performance of HPCC Random Access (Cray XT4)

- Chapel TPL=1
- Chapel TPL=2
- Chapel TPL=4
- Chapel TPL=8

Number of Locales

GUP/s

2048
RA Efficiency: Chapel vs. MPI (2009)

Efficiency of HPCC Random Access on 32+ Locales (Cray XT4)

% Efficiency (of scaled Chapel TPL=4 local GUP/s)

Number of Locales

Chapel TPL=1
Chapel TPL=2
Chapel TPL=4
Chapel TPL=8
MPI PPN=4
MPI No Buckets PPN=4
MPI+OpenMP TPN=4
HPL Notes
Block-Cyclic Distribution

BlockCyclic(start=(1,1), blksize=4)
Block-Cyclic Distribution

BlockCyclic(start=(1,1), blksize=4)
Block-Cyclic Distribution

BlockCyclic\(\text{start}=(1,1), \text{blksize}=4\)
Block-Cyclic Distribution

**Notes:**
- at extremes, Block-Cyclic is:
  - the same as Cyclic (when blkSize == 1)
  - similar to Block
    - the same when things divide evenly
    - slightly different when they don’t (last locale will own more or less than blkSize)

**Benefits relative to Block and Cyclic:**
- if work isn’t well load-balanced across a domain (and is spatially-based), likely to result in better balance across locales than Block
- provides nicer locality than Cyclic (locales own blocks rather than singletons)

**Also:**
- a good match for algorithms that are block-structured in nature
  - like HPL
  - typically the distribution’s blocksize will be set to the algorithm’s
HPL Overview

- **Category**: dense linear-algebra
- **Computation**:
  - compute L-U factorization of a matrix $A$
    - $L = \text{lower-triangular matrix}$
    - $U = \text{upper-triangular matrix}$
    - $LU = A$
  - in order to solve $Ax = b$
  - solving $Ax = b$ is easier using these triangular matrices
HPL Overview (continued)

- **Approach:** block-based recursive algorithm
- **Details:**
  - pivot (swap rows of matrix and vectors) to maintain numerical stability
  - store $b$ adjacent to $A$ for convenience, ease-of-pivoting
  - reuse $A$’s storage to represent $L$ and $U$
HPL Configs

// matrix size and blocksize
config const n = computeProblemSize(numMatrices, elemType, rank=2, memFraction=2, retType=indexType), blkSize = 5;

// error tolerance for verification
config const epsilon = 2.0e-15;

// standard random initialization stuff
config const useRandomSeed = true,
    seed = if useRandomSeed then SeedGenerator.currentTime
    else 31415;

// standard knobs for controlling printing
config const printParams = true,
    printArrays = false,
    printStats = true;
HPL Distributions and Domains

```javascript
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
                                blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
                      = [1..n, 1..n+1],
                          MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType, // the matrix A and vector b
        piv: [1..n] indexType,    // a vector of pivot values
        x  : [1..n] elemType;    // the solution vector, x

var A => Ab[MatrixSpace],       // an alias for the Matrix part of Ab
       b => Ab[.., n+1];        // an alias for the last column of Ab
```
```javascript
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
                              blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
                      = [1..n, 1..n+1],
                      MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType,
        piv: [1..n] indexType,
        x  : [1..n] elemType;

var A => Ab[MatrixSpace],
        b => Ab[.., n+1];
```
HPL Distributions and Domains

```
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
                              blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
                      = [1..n, 1..n+1],
                         MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType,
       piv: [1..n] indexType,
       x : [1..n] elemType;

var A => Ab[MatrixSpace],
       b => Ab[.., n+1];
```
HPL Distributions and Domains

```javascript
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
    blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
    = [1..n, 1..n+1],
    MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType,
    piv: [1..n] indexType,
    x  : [1..n] elemType;

var A => Ab[MatrixSpace],
    b => Ab[.., n+1];
```
HPL Distributions and Domains

```javascript
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
                        blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
    = [1..n, 1..n+1],
    MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType,
    piv: [1..n] indexType,
    x : [1..n] elemType;

var A => Ab[MatrixSpace],
    b => Ab[.., n+1];
```
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1), blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst = [1..n, 1..n+1], MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType, piv: [1..n] indexType, x : [1..n] elemType;

var A => Ab[MatrixSpace], b => Ab[.., n+1]
HPL Distributions and Domains

```javascript
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
    blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
    = [1..n, 1..n+1],
    MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType,
    piv: [1..n] indexType,
    x : [1..n] elemType;

var A => Ab[MatrixSpace],
    b => Ab[.., n+1];
```
HPL Distributions and Domains

```javascript
const BlkCycDst = new dmap(new BlockCyclic(start=(1,1),
    blkSize=blkSize));

const MatVectSpace: domain(2, indexType) dmapped BlkCycDst
    = [1..n, 1..n+1],
    MatrixSpace = MatVectSpace[.., ..n];

var Ab : [MatVectSpace] elemType,
    piv: [1..n] indexType,
    x  : [1..n] elemType;

var A => Ab[MatrixSpace],
    b => Ab[.., n+1];
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

piv
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

x
```
HPL Callgraph

- main()
  - initAB()
  - LUFactorize()
    - panelSolve()
    - updateBlockRow()
    - schurComplement()
      - dgemm()
  - backwardSub
  - verifyResults()
HPL Callgraph

- main()
main()

initAB(Ab);

const startTime = getCurrentTime();

LUFactorize(n, Ab, piv);

x = backwardSub(n, A, b);

const execTime = getCurrentTime() - startTime;

const validAnswer = verifyResults(Ab, MatrixSpace, x);
printResults(validAnswer, execTime);
HPL Callgraph

- main()
  - initAB()
  - LUFactorize()
  - backwardSub
  - verifyResults()
HPL Callgraph

- main()
  - initAB()
  - LUFactorize()
  - backwardSub
  - verifyResults()
LUFactorize

- main loop marches down block diagonal

- each iteration views matrix as follows:

- as computation proceeds, since these four areas shrink
  ⇒ Block-Cyclic more appropriate than Block
def LUFactorize(n: indexType, Ab: [1..n, 1..n+1] elemType, 
    piv: [1..n] indexType) {
    const AbD = Ab.domain; // alias Ab.domain to save typing
    piv = 1..n;

    for blk in 1..n by blkSize {
        const tl = AbD[blk..#blkSize, blk..#blkSize],
            tr = AbD[blk..#blkSize, blk+blkSize..],
            bl = AbD[blk+blkSize.., blk..#blkSize],
            br = AbD[blk+blkSize.., blk+blkSize..],
            l = AbD[blk.., blk..#blkSize];

        panelSolve(Ab, l, piv);
        if (tr.numIndices > 0) then
            updateBlockRow(Ab, tl, tr);

        if (br.numIndices > 0) then
            schurComplement(Ab, blk);
    }
}
What does each kernel use?

- `panelSolve()`
  - $l$
  - $Ab$

- `updateBlockRow()`
  - $tl$
  - $tr$

- `schurComplement()`
  - $bl$
  - $br$
HPL Callgraph

main()
  • initAB()
  • LUFactorize()
    ▪ panelSolve()
    ▪ updateBlockRow()
    ▪ schurComplement()
  • backwardSub
  • verifyResults()
def panelSolve(Ab: [], t, panel: domain(2, indexType), piv: [] indexType) {

    const pnlRows = panel.dim(1),
    pnlCols = panel.dim(2);

    assert (piv.domain.dim(1) == Ab.domain.dim(1));

    if (pnlCols.length == 0) then return;

    for k in pnlCols {
        // iterate through the columns of the panel
    }
}
panelSolve

- iterate over the columns of the panel, serially
- find the value with the largest magnitude in the column (the *pivot value*
- swap that row with the top in that column *for the whole Ab matrix*
- scale the rest of that column by the pivot value
panelSolve

```plaintext
var col = panel[k.., k..k];

if col.dim(1).length == 0 then return;

const (, (pivotRow, )) = maxloc reduce(abs(Ab(col)), col),
pivot = Ab[pivotRow, k];

piv[k] <=> piv[pivotRow];

Ab[k, ..] <=> Ab[pivotRow, ..];

if (pivot == 0) then
    halt("Matrix can not be factorized");

if k+1 <= pnlRows.high then
    Ab(col)[k+1.., k..k] /= pivot;

if k+1 <= pnlRows.high && k+1 <= pnlCols.high then
    forall (i,j) in panel[k+1.., k+1..] do
        Ab[i,j] -= Ab[i,k] * Ab[k,j];
```
HPL Callgraph

- main()
  - initAB()
  - LUFactorize()
    - panelSolve()
    - updateBlockRow()
    - schurComplement()
  - backwardSub
  - verifyResults()
updateBlockRow

- iterate over the rows of $tr$, serially

- accumulate into each value the product of its predecessors from $tl$ and previous rows
if (tr.numIndices > 0) then
  updateBlockRow(Ab, tl, tr);

def updateBlockRow(Ab: [], tl, tr: domain(2)) {
  const tlRows = tl.dim(1),
    tlCols = tl.dim(2),
    trRows = tr.dim(1),
    trCols = tr.dim(2);

  assert(tlCols == trRows);

  for i in trRows do
    forall j in trCols do
      for k in tlRows.low..i-1 do
        Ab[i, j] -= Ab[i, k] * Ab[k, j];
  }
}
updateBlockRow w/ distribution
updateBlockRow w/ distribution
updateBlockRow w/ distribution

Ab

TL (replicated, logically)
updateBlockRow w/ distribution

Ab

TL (replicated, physically)
HPL Callgraph

- main()
  - initAB()
  - LUFactorize()
    - panelSolve()
    - updateBlockRow()
    - schurComplement()
  - backwardSub
  - verifyResults()
schurComplement

- accumulate into each block in br the product of its corresponding blocks from bl and tr
updateBlockRow w/ distribution
schurComplement w/ distribution

replicated col, logical view

replicated row, logical view
schurComplement w/ distribution

replicated col, physical view

replicated row, physical view
if (br.numIndices > 0) then
    schurComplement(Ab, blk);

def schurComplement(Ab: [1..n, 1..n+1] elemType, ptOp: indexType) {
    const AbD = Ab.domain;

    const ptSol = ptOp+blkSize;

    const replAD: domain(2) = AbD[ptSol.., ptOp..#blkSize],
        replBD: domain(2) = AbD[ptOp..#blkSize, ptSol..];

    const replA : [replAD] elemType = Ab[ptSol.., ptOp..#blkSize],
        replB : [replBD] elemType = Ab[ptOp..#blkSize, ptSol..];

    forall (row,col) in AbD[ptSol.., ptSol..] by (blkSize, blkSize) {
        local {
            const aBlkD = replAD[row..#blkSize, ptOp..#blkSize],
                bBlkD = replBD[ptOp..#blkSize, col..#blkSize],
                cBlkD = AbD[row..#blkSize, col..#blkSize];

            dgemm(aBlkD.dim(1).length, aBlkD.dim(2).length, bBlkD.dim(2).length,
                replA(aBlkD), replB(bBlkD), Ab(cBlkD));
        } } }
HPL Callgraph

- **main()**
  - `initAB()`
  - `LUFactorize()`
    - `panelSolve()`
    - `updateBlockRow()`
    - `schurComplement()`
      - `dgemm()`
  - `backwardSub`
  - `verifyResults()`
def dgemm(p: indexType,                // number of rows in A
    q: indexType,                // number of cols in A, number of rows in B
    r: indexType,                // number of cols in B
    A: [1..p, 1..q] ?t,
    B: [1..q, 1..r] t,
    C: [1..p, 1..r] t) {

    for i in 1..p do
        for j in 1..r do
            for k in 1..q do
                C[i, j] -= A[i, k] * B[k, j];
            }

}
HPL Callgraph

- main()
  - initAB()
  - LUFactorize()
    - panelSolve()
    - updateBlockRow()
    - schurComplement()
      - dgemm()
  - backwardSub
  - verifyResults()