Optimizing Data Re-allocation Via Communication Aggregation in Chapel

Alberto Sanz, Rafael Asenjo, Juan López, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi and Bradford L. Chamberlain
Introduction

• Parallel frameworks
  ▫ Distributed memory
    • Message passing: MPI, SHMEM, GASNet, …
  ▫ Shared memory
    • Pthreads, OpenMP
    • Task frameworks:
      • Intel TBB, Cilk, Intel CnC, MS TPL, Java Concurrency

• Parallel Languages
  • Partitioned Global Address Space (PGAS)
    • UPC, Co-array Fortran (CAF), Titanium (Parallel Java)
  • High Performance Computing Systems (HPCS)
    • Chapel (Cray), X10 (IBM), Fortress (Sun/Oracle)

• Heterogeneous: CUDA, OpenCL, OpenACC
Chapel Motivation

- Emerging parallel language
  - Open source initiative under development
  - Pioneered by Cray Inc.
    - In the context of DARPA’s High Productivity Computing Systems (HPCS).

- Goals:
  - Productivity: performance and programmability
  - Portability
  - Robustness
  - Multiresolution philosophy
Chapel Motivation

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

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

**In pictures:**

![Diagram showing vector operations and scalar multiplication](image_url)
#include <hpcc.h>

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d). \n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 3.0;
    }

    scalar = 3.0;

    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];

    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);

    return 0;
}
#include <hpcc.h>
#endif _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 3.0;
    }

    scalar = 3.0;

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];

    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);

    return 0;
}
MPI + OpenMP vs CUDA

#include <hpcc.h>
#define OPENMP
#include <omp.h>

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &commSize);
  MPI_Comm_rank(comm, &myRank);
  rv = HPCC_Stream(params, 0 == myRank);
  MPI_Reduce(&rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm);
  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  double scalar;
  VectorSize = HPCC_LocalVectorSize(params, 3, sizeof(double), 0);
  a = HPCC_XMALLOC(double, VectorSize);
  b = HPCC_XMALLOC(double, VectorSize);
  c = HPCC_XMALLOC(double, VectorSize);
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf(outFile, "Failed to allocate memory (%d)\n", VectorSize);
      fclose(outFile);
      return 1;
    }
  }
  #ifdef _OPENMP
  #pragma omp parallel for
  #endif
  for (j = 0; j < VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 3.0;
  }
  scalar = 3.0;
  #ifdef _OPENMP
  #pragma omp parallel for
  #endif
  for (j = 0; j < VectorSize; j++)
    a[j] = b[j] + scalar * c[j];
  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);
  return 0;
}

#include <hpcc.h>
#define CUDA

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;
  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x);
  if (N % dimBlock.x != 0) dimGrid.x++;
  cudaMemcpy(d_b, 2.0f, N);
  cudaMemcpy(d_c, 3.0f, N);
  cudaMemcpy<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
  cudaMemcpy(d_a, d_b);
  cudaMemcpy(d_b, d_c);
  cudaMemcpy(d_c, d_a);
  cudaThreadSynchronize();
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a, float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad(float *a, float *b, float *c,
                             float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx] + scalar * b[idx];
}
MPI + OpenMP vs Cuda

```c
#include "hpcc.h"
#define N 2000000
int main() {
    float *d_a, *d_b, *d_c;
    float scalar;
    cudaMalloc((void**)&d_a, sizeof(float)*N);
    cudaMalloc((void**)&d_b, sizeof(float)*N);
    cudaMalloc((void**)&d_c, sizeof(float)*N);
    dim3 dimBlock(128);
    dim3 dimGrid(N/dimBlock.x);
    if (N % dimBlock.x != 0) dimGrid.x+=1;
    set_array<<<dimGrid,dimBlock>>>(d_b, 2.0f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, 3.0f, N);
    scalar=3.0f;
    STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
    cudaThreadSynchronize();
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

**Philosophy:** Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert each to focus on their strengths.

Global data re-allocation via communication aggregation in Chapel. University of Málaga – Spain
Chapel Background

- **Domains:** index space
  
  ```chapel
  var DA = [1..8];
  var DB = [1..8 by 2];
  var A:[DA] real;
  var B:[DB] real;
  ```

- **Data distributions:**
  - User defined distributions
  - Standard distributions: Block, Cyclic, Block-Cyclic, ...

  ```chapel
  var DC = DA dmapped Block(DA);
  var C:[DC] real;
  ```
Global data re-allocation via communication aggregation in Chapel. University of Málaga – Spain

BBox = [1..10, 1..10];
Dom = [3..10 by 3, 2..10 by 2];
const myDist = new dmap(new Block(BBox));
const myDom = Dom dmapped myDist;
var A:[myDom] real(64) = 1..;

myDist

Block
rank = 2;
boundingBox = [1..10, 1..10];
targetLocDom = [0..1, 0..1];
targetLocales = L0 L1 L2 L3
locDist[0..1, 0..1];

myDom

BlockDom
rank = 2;
stridable = true;
dist
whole = [3..10 by 3, 2..10 by 2];
locDoms[0..1, 0..1]

A

BlockArr
rank = 2;
eltType = real(64);
stridable = true;
(dom)
locArr[0..1, 0..1]

LocBlock

myChunk = [-∞..5, -∞..5]

L0

LocBlockDom

myBlock = [3..5 by 3, 2..5 by 2];
L0

myBlock = [3..5 by 3, 6..10 by 2];
L1

myBlock = [6..10 by 3, 2..5 by 2];
L2

myBlock = [6..10 by 3, 6..10 by 2];
L3

myElems = 1.0 2.0

LocBlockArr

myElems = 3.0 4.0 5.0

L0

myElems = 6.0 7.0 11.0 12.0

L1

myElems = 8.0 9.0 10.0 13.0 14.0 15.0

L3

L0

L1

L2

L3

Global data re-allocation via communication aggregation in Chapel. University of Malaga – Spain

SBAC-PAU'2012
GASNet

[1..4, 1..4 by 3, 2..4]

3D Representation

```
void gasnet_gets_bulk (DstAddr, DstStrides, SrcNode, SrcAddr, SrcStrides, Count, StrideLevels)
```

```
void gasnet_puts_bulk(DstNode, DstAddr, DstStrides, SrcAddr, SrcStrides, Count, StrideLevels)
```

1D Representation

```
void gasnet_gets_bulk (DstAddr, DstStrides, SrcNode, SrcAddr, SrcStrides, Count, StrideLevels)
```

```
void gasnet_puts_bulk(DstNode, DstAddr, DstStrides, SrcAddr, SrcStrides, Count, StrideLevels)
```

StrideLevels = 2  Count=(3,2,4)  SrcStrides= (12, 16)
Data aggregation implementation

\[ A[D1] = B[D2] \]

- **DR = DR**
  - A and B are DR arrays, A allocated on locale 0 and B on locale 1.
  - One call to `gasnet_gets_bulk` (if executed on locale 0) or `gasnet_puts_bulk` (if executed on locale 1)
Data aggregation implementation

A[D1] = B[D2]

- DR = BD

```chapel
forall i in 0..3 do
  on A do { //co-locate with A
    // Run DR=DR assignments in parallel
    A[Slice_i] = B.locArr[i].myElems
  }
```

BD = Block Distributed Array (rely on several DR arrays)
Data aggregation implementation

A[D1] = B[D2]

- BD = BD

forall i in 0..3 do
  on A.locArr[i] do { // co-locate with locArr[i]
    // Run DR=BD assignments in parallel
    A.locArr[i].myElems[dest] = B[Slice_i];
  }

Global data re-allocation via communication aggregation in Chapel. University of Málaga – Spain
Results

Cray HECToR supercomputer:
- Cray XE6, 90,112 cores - AMD Opteron 2.3 GHz.
- Cray Gemini interconnection network.
- Peak capacity: 800 Tflop/s.
- Top500.org: 19

Cray Jaguar supercomputer:
- Cray XK6, 298,592 cores - AMD Opteron 2.2 GHz.
- Gemini 3D torus interconnection network.
- Peak capacity: 2.6 Pflop/s.
- Top500.org: 6
Results
(BlockDist re-allocation)

```chapel
config const n = 500;

var Dist1 = new dmap(new Block([1..n,1..n,1..n]));
var Dist2 = new dmap(new Block([1..2*n,1..2*n,1..2*n]));

var Dom1: domain(3, int) dmapped Dist1 = [1..n,1..n,1..n];
var Dom2: domain(3, int) dmapped Dist2 = [1..2*n,1..2*n,1..2*n];

var A:[Dom1] real(64);
var B:[Dom2] real(64);

var D=[1..n,1..n by 4,1..n];
A[D]=B[D]; // Assignment
```

- Two Block distributions
- Two 3D distributed domains
- Two 3D distributed ARRAYS
- Slice domain: 500x125x500 indices
- Array assignment: 250 MBytes
Results
(BlockDist re-allocation)

- wA - with Aggregation optimization, speedup vs. woA up to 1,000x
- Gemini conduit is faster than MPI conduit on each machine
### Results
*(FFT, Block to Cyclic redistribution)*

<table>
<thead>
<tr>
<th>Loc</th>
<th>T. BtoC</th>
<th>T. CtoB</th>
<th>T. BtoC</th>
<th>T. CtoB</th>
<th>Comm. Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,850</td>
<td>6,6653</td>
<td>0,0109</td>
<td>0,0099</td>
<td><strong>361,31</strong></td>
</tr>
<tr>
<td>4</td>
<td>0,944</td>
<td>5,4038</td>
<td>0,3032</td>
<td>1,3381</td>
<td><strong>4,63</strong></td>
</tr>
<tr>
<td>16</td>
<td>2,143</td>
<td>5,1227</td>
<td>0,2454</td>
<td>0,4146</td>
<td><strong>11,00</strong></td>
</tr>
</tbody>
</table>

FFT on **HECToR**, Gemini conduit, n=2\(^{20}\)

<table>
<thead>
<tr>
<th>Loc</th>
<th>T. BtoC</th>
<th>T. CtoB</th>
<th>T. BtoC</th>
<th>T. CtoB</th>
<th>Comm. Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,0175</td>
<td>0,0761</td>
<td>0,00495</td>
<td>0,03987</td>
<td><strong>2,09</strong></td>
</tr>
<tr>
<td>4</td>
<td>3,1547</td>
<td>4,6204</td>
<td>0,01095</td>
<td>0,08519</td>
<td><strong>80,88</strong></td>
</tr>
<tr>
<td>16</td>
<td>1,3238</td>
<td>1,5435</td>
<td>0,03310</td>
<td>0,05919</td>
<td><strong>31,07</strong></td>
</tr>
</tbody>
</table>

FFT on **Jaguar**, Gemini conduit, n=2\(^{20}\)

- Part of HPC Challenge (HPCC) suite
- Aggregation speedup up to 80x
Results
(PARACR, Block to Cyclic redistribution)

- PARACR is an algorithm for solving tridiagonal systems of equations
- Block distribution better for the first steps, Cyclic for the last steps
- Aggregation speedup for Block-to-Cyclic redistribution up to 1,000x
Conclusions

• Chapel is an emerging parallel programming language
• This work explores the aggregation of communications in Block and Cyclic Distribution
• We take advantage of GASNet one-sided bulk communication routines
• The results show significant speedups for communications times
• Included in the new release: Chapel 1.6

• Future Work:
  ▫ Generalize this optimization to other distributions
  ▫ Improve scheduling of communications
For More Information

Chapel project page:
http://chapel.cray.com
  ▫ Overview, papers, presentations, spec, …

Chapel SourceForge page:
https://sourceforge.net/projects/chapel/
  ▫ Release downloads, public mailing lists, code repository, …

Research group page:
http://www.ac.uma.es/~asenjo/research/
Parallel programming models, Parallel languages (Chapel, UPC, X10), Parallel libraries (TBB), …