Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures

Uday Bondhugula
Indian Institute of Science

Supercomputing 2013
Nov 16–22, 2013
Denver, Colorado

Distributed-memory compilation

- Manual parallelization for distributed-memory is extremely hard (even for affine loop nests)

Objectives

- Automatically generate MPI code from sequential C affine loop nests

Distributed-memory compilation

- Manual parallelization for distributed-memory is extremely hard (even for affine loop nests)

Objectives

- Automatically generate MPI code from sequential C affine loop nests

Distributed-memory compilation – why again?

- Large amount of literature already exists through early 1990s
  1. Past works: limited success
  2. *Still* no automatic tool has been available
  3. However, we now have new polyhedral libraries, transformation frameworks, code generators, and tools
  4. The same techniques are needed to compile for CPUs-GPU heterogeneous multicores
  5. Can be integrated with emerging runtimes
- Make a fresh attempt to solve this problem

Distributed-memory compilation – why again?

- Large amount of literature already exists through early 1990s
  1. Past works: limited success
  2. *Still* no automatic tool has been available
  3. However, we now have new polyhedral libraries, transformation frameworks, code generators, and tools
  4. The same techniques are needed to compile for CPUs-GPU heterogeneous multicores
  5. Can be integrated with emerging runtimes
- Make a fresh attempt to solve this problem

Distributed-memory compilation – why again?

- Large amount of literature already exists through early 1990s
  1. Past works: limited success
  2. *Still* no automatic tool has been available
  3. However, we now have new polyhedral libraries, transformation frameworks, code generators, and tools
  4. The same techniques are needed to compile for CPUs-GPU heterogeneous multicores
  5. Can be integrated with emerging runtimes
- Make a fresh attempt to solve this problem

Distributed-memory compilation – why again?

- Large amount of literature already exists through early 1990s
  1. Past works: limited success
  2. *Still* no automatic tool has been available
  3. However, we now have new polyhedral libraries, transformation frameworks, code generators, and tools
  4. The same techniques are needed to compile for CPUs-GPU heterogeneous multicores
  5. Can be integrated with emerging runtimes
- Make a fresh attempt to solve this problem

Distributed-memory compilation – why again?

- Large amount of literature already exists through early 1990s
  1. Past works: limited success
  2. *Still* no automatic tool has been available
  3. However, we now have new polyhedral libraries, transformation frameworks, code generators, and tools
  4. The same techniques are needed to compile for CPUs-GPU heterogeneous multicores
  5. Can be integrated with emerging runtimes
- Make a fresh attempt to solve this problem

Distributed-memory compilation – why again?

- Large amount of literature already exists through early 1990s
  1. Past works: limited success
  2. *Still* no automatic tool has been available
  3. However, we now have new polyhedral libraries, transformation frameworks, code generators, and tools
  4. The same techniques are needed to compile for CPUs-GPU heterogeneous multicores
  5. Can be integrated with emerging runtimes
- Make a fresh attempt to solve this problem

Why do we need communication?

- Communication during parallelization is a result of data dependences
- No data dependences $\Rightarrow$ ($\sim$) no communication
- Parallel loop implies no dependences satisfied by it
  - Communication is due to dependences that are satisfied outside but have (non-zero) components on the parallel loop

Why do we need communication?

- Communication during parallelization is a result of data dependences
- No data dependences ⇒ (∼) no communication
- Parallel loop implies no dependences satisfied by it
    - Communication is due to dependences that are satisfied outside but have (non-zero) components on the parallel loop

Why do we need communication?

- Communication during parallelization is a result of data dependences
- No data dependences $\Rightarrow$ ($\sim$) no communication
- Parallel loop implies no dependences satisfied by it
  - Communication is due to dependences that are satisfied outside but have (non-zero) components on the parallel loop

Dependences and Communication

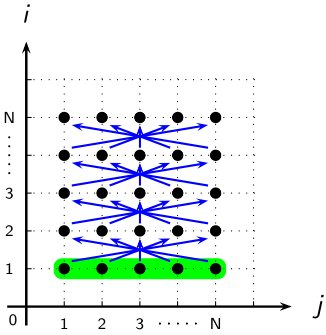


Figure : Inner parallel loop, j: hyperplane (0,1)

- The inner loop can be executed in parallel with communication for each iteration of the outer sequential loop

Dependences and Communication

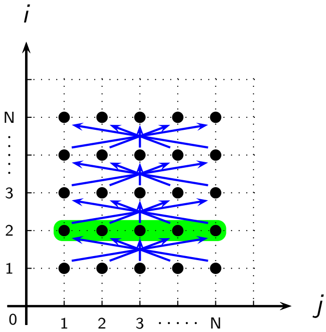


Figure : Inner parallel loop, j: hyperplane (0,1)

- The inner loop can be executed in parallel with communication for each iteration of the outer sequential loop

Dependences and Communication

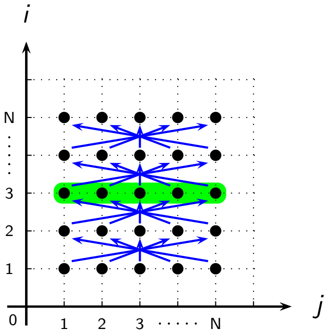


Figure : Inner parallel loop, j: hyperplane (0,1)

- The inner loop can be executed in parallel with communication for each iteration of the outer sequential loop

Dependences and Communication

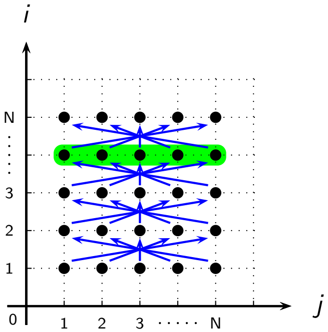


Figure : Inner parallel loop, *j*: hyperplane (0,1)

- The inner loop can be executed in parallel with communication for each iteration of the outer sequential loop

Dependences and Communication

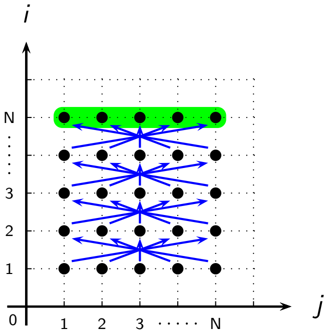
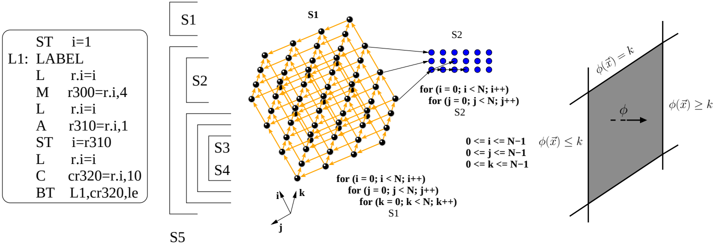


Figure : Inner parallel loop, *j*: hyperplane (0,1)

- The inner loop can be executed in parallel with communication for each iteration of the outer sequential loop

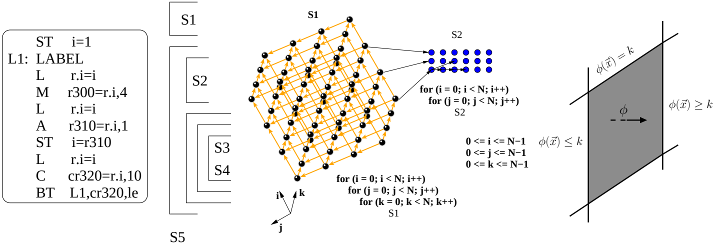
A polyhedral optimizer – various phases

1. Extracting a polyhedral representation (from sequential C)
2. Dependence analysis
3. Transformation and parallelization
4. Code generation (getting out of polyhedral extraction)



A polyhedral optimizer – various phases

1. Extracting a polyhedral representation (from sequential C)
2. Dependence analysis
3. **Transformation and parallelization**
4. Code generation (getting out of polyhedral extraction)



Distributed-memory parallelization

Involves a number of sub-problems

1. Finding the right computation partitioning
2. Data distribution and data allocation (weak scaling)
3. Determining communication sets given the above
4. Packing and unpacking data
5. Determining communication partners given the above

Distributed-memory parallelization

Involves a number of sub-problems

1. Finding the right computation partitioning
2. Data distribution and data allocation (weak scaling)
3. **Determining communication sets given the above**
4. **Packing and unpacking data**
5. **Determining communication partners given the above**

Distributed-memory code generation

- What to send?
- Whom to send to?

Difficulties

- For non-uniform dependences, not known how far dependences traverse
- Number of iterations (or tiles) is not known at compile time
- Number of processors may not be known at compile time (portability)
- Virtual to physical processor approach: are you sending to two virtual processors that are the same physical processor?

Distributed-memory code generation

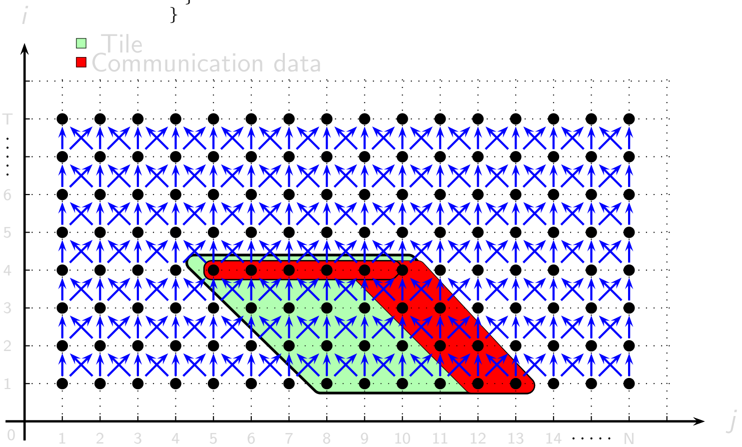
- What to send?
- Whom to send to?

Difficulties

- For non-uniform dependences, not known how far dependences traverse
- Number of iterations (or tiles) is not known at compile time
- Number of processors may not be known at compile time (portability)
- Virtual to physical processor approach: are you sending to two virtual processors that are the same physical processor?

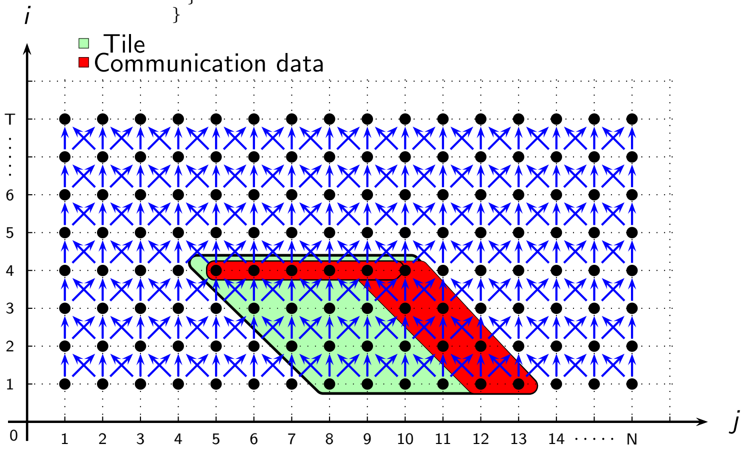
A near-neighbor computation example

```
for (t=1; t<=T−1; t++){
  for (j=1; j<=N−1; j++){
    u[t%2][j] = 0.333∗(u[(t−1)% 2][j−1]
        + u[(t−1)%2][j] + u[(t−1)%2][j+1]);
  }
}
```



A near-neighbor computation example

```
for (t=1; t<=T-1; t++){
  for (j=1; j<=N-1; j++){
    u[t%2][j]  = 0.333*(u[(t-1)% 2][j-1]
        + u[(t-1)%2][j] + u[(t-1)%2][j+1]);
  }
}
```



Floyd-Warshall example

Use to compute all-pairs shortest-paths in a directed graph

```
for (k=0; k < N; k++) {
 for (y=0; y < N; y++) {
  for (x=0; x < N; x++) {
    pathDistanceMatrix[y][x] = min(pathDistanceMatrix[y][k] +
             pathDistanceMatrix[k][x], pathDistanceMatrix[y][x]);
    }
  }
}
```

Figure : Floyd-warshall algorithm

Floyd-Warshall communication pattern

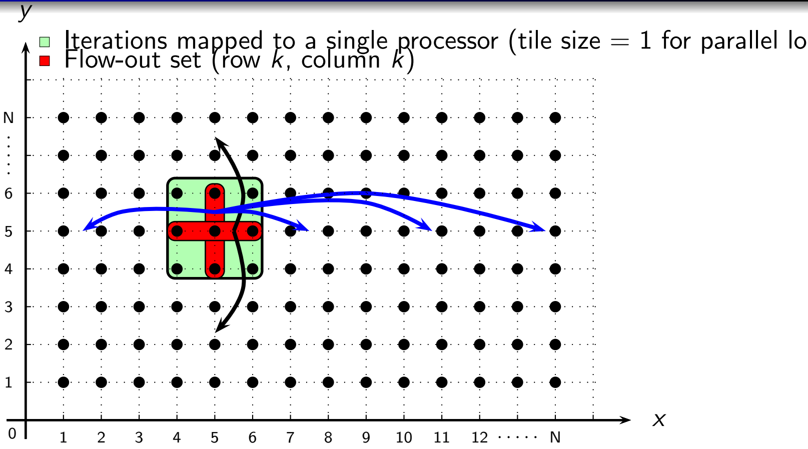


Figure : Communication for Floyd-Warshall: at outer loop iteration $k-1$, processor(s) updating the k^{th} row and k^{th} column broadcast them to processors along their column and row respectively.

Code generation after transformation: example – 2-d seidel

- Performing distributed memory code generation after transformation

```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```

- Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

Code generation after transformation: example – 2-d seidel

- Performing distributed memory code generation after transformation

```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
       a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                  a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```

- Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

Code generation after transformation: example – 2-d seidel

- Performing distributed memory code generation after transformation

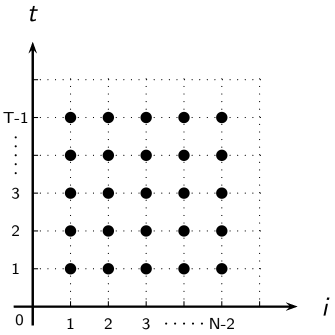
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
       a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
               a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```

- Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

Code generation after transformation

- Performing distributed memory code generation on transformed code

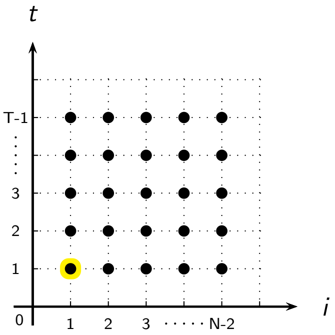
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

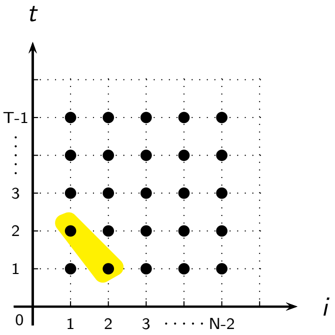
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
       a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                  a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

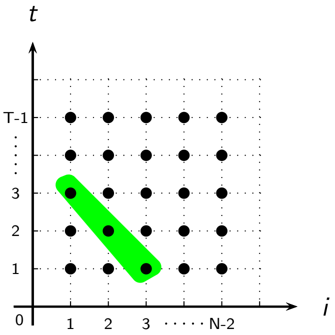
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

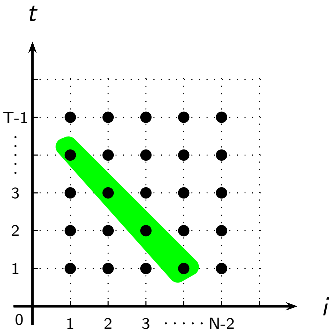
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

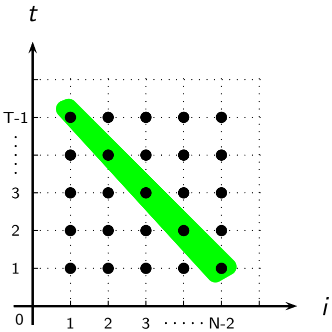
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

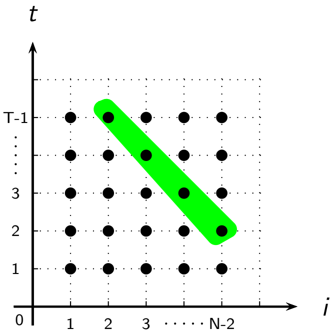
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
       a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

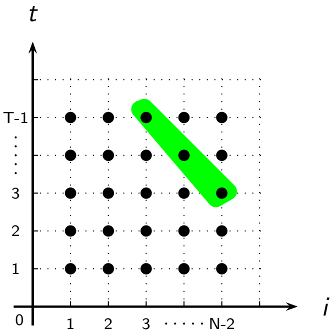
```
for (t=0; t<=T−1; t++) {
  for (i=1; i<=N−2; i++) {
    for (j=1; j<=N−2; j++) {
      a[i][j] = (a[i−1][j−1] + a[i−1][j] + a[i−1][j+1] + a[i][j−1] +
                a[i][j] + a[i][j+1] + a[i+1][j−1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

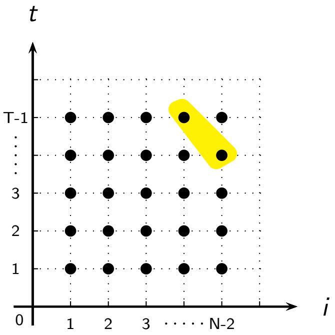
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
      a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                 a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

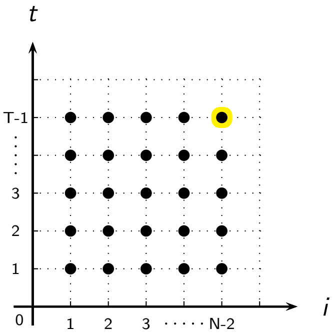
```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
       a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                  a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

```
for (t=0; t<=T−1; t++) {
  for (i=1; i<=N−2; i++) {
    for (j=1; j<=N−2; j++) {
        a[i][j] = (a[i−1][j−1] + a[i−1][j] + a[i−1][j+1] + a[i][j−1] +
                   a[i][j] + a[i][j+1] + a[i+1][j−1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```



Code generation after transformation

- Performing distributed memory code generation on transformed code

```
for (t=0; t<=T−1; t++) {
  for (i=1; i<=N−2; i++) {
    for (j=1; j<=N−2; j++) {
       a[i][j] = (a[i−1][j−1] + a[i−1][j] + a[i−1][j+1] + a[i][j−1] +
              a[i][j] + a[i][j+1] + a[i+1][j−1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```

- Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)
- $T(t,i,j) = (t, t+i, 2t+i+j)$
- Tile all dimensions
- Create a tile schedule, and identify loop to be parallelized
- Generate communication primitives on this code

Code generation after transformation

- Performing distributed memory code generation on transformed code

```
for (t=0; t<=T-1; t++) {
  for (i=1; i<=N-2; i++) {
    for (j=1; j<=N-2; j++) {
       a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
    }
  }
}
```

- Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)
- $T(t, i, j) = (t, t + i, 2t + i + j)$
- Tile all dimensions
- Create a tile schedule, and identify loop to be parallelized
- Generate communication primitives on this code

Computing data accessed

```
if ((N >= 3) && (T >= 1)) {
  for (t1=0;t1<=floord(N+2*T−4,32);t1++) {
    lbp=max(ceild(t1,2), ceild (32*t1−T+1,32));
    ubp=min(min(floord(N+T−3,32),floord(32*t1+N+29,64)),t1);
#pragma omp parallel for
    for (t2=lbp;t2<=ubp;t2++) {
      for (t3=max(ceild(64*t2−N−28,32),t1);t3<=min(min(min(min(floord(N+T−3,16),floord(32*t1−32*t2+N+29,16)
        for (t4=max(max(max(32*t1−32*t2,32*t2−N+2),16*t3−N+2),−32*t2+32*t3−N−29);t4<=min(min(min(min
          for (t5=max(max(32*t2,t4+1),32*t3−t4−N+2);t5<=min(min(32*t2+31,32*t3−t4+30),t4+N−2);t5++) {
            for (t6=max(32*t3,t4+t5+1);t6<=min(32*t3+31,t4+t5+N−2);t6++) {
              a[−t4+t5][−t4−t5+t6]=(a[−t4+t5−1][−t4−t5+t6−1]+a[−t4+t5−1][−t4−t5+t6]+a[−t4+t5−1][−t4−
            }
          }
        }
      }
    }
    / * communication code should go here */
  }
}
```

- Image of $(-t4 + t5, -t4 - t5 + t6)$ over an integer set
- Straightforward to accomplish via polyhedral libraries
    - ISL: just create an isl map
    - Polylib: use polylib image function or projections

Computing data accessed

```
if ((N >= 3) && (T >= 1)) {
  for (t1=0;t1<=floord(N+2*T-4,32);t1++) {
    lbp=max(ceild(t1,2), ceild (32*t1-T+1,32));
    ubp=min(min(floord(N+T-3,32),floord(32*t1+N+29,64)),t1);
#pragma omp parallel for
    for (t2=lbp;t2<=ubp;t2++) {
      for (t3=max(ceild(64*t2-N-28,32),t1);t3<=min(min(min(min(floord(N+T-3,16),floord(32*t1-32*t2+N+29,16)
        for (t4=max(max(max(32*t1-32*t2,32*t2-N+2),16*t3-N+2),-32*t2+32*t3-N-29);t4<=min(min(min(min
          for (t5=max(max(32*t2,t4+1),32*t3-t4-N+2);t5<=min(min(32*t2+31,32*t3-t4+30),t4+N-2);t5++) {
            for (t6=max(32*t3,t4+t5+1);t6<=min(32*t3+31,t4+t5+N-2);t6++) {
              a[-t4+t5][-t4-t5+t6]=(a[-t4+t5-1][-t4-t5+t6-1]+a[-t4+t5-1][-t4-t5+t6]+a[-t4+t5-1][-t4-
            }
          }
        }
      }
    }
    / * communication code should go here */
  }
}
```

- Image of $(-t4 + t5, -t4 - t5 + t6)$ over an integer set
- Straightforward to accomplish via polyhedral libraries
  - ISL: just create an isl map
  - Polylib: use polylib image function or projections

Computing data accessed

```
if ((N >= 3) && (T >= 1)) {
  for (t1=0;t1<=floord(N+2*T-4,32);t1++) {
    lbp=max(ceild(t1,2), ceild (32*t1-T+1,32));
    ubp=min(min(floord(N+T-3,32),floord(32*t1+N+29,64)),t1);
#pragma omp parallel for
    for (t2=lbp;t2<=ubp;t2++) {
      for (t3=max(ceild(64*t2-N-28,32),t1);t3<=min(min(min(min(floord(N+T-3,16),floord(32*t1-32*t2+N+29,16)
        for (t4=max(max(max(32*t1-32*t2,32*t2-N+2),16*t3-N+2),-32*t2+32*t3-N-29);t4<=min(min(min(min
          for (t5=max(max(32*t2,t4+1),32*t3-t4-N+2);t5<=min(min(32*t2+31,32*t3-t4+30),t4+N-2);t5++) {
            for (t6=max(32*t3,t4+t5+1);t6<=min(32*t3+31,t4+t5+N-2);t6++) {
              a[-t4+t5][-t4-t5+t6]=(a[-t4+t5-1][-t4-t5+t6-1]+a[-t4+t5-1][-t4-t5+t6]+a[-t4+t5-1][-t4-
            }
          }
        }
      }
    }
    / * communication code should go here */
  }
}
```

- Image of $(-t4 + t5, -t4 - t5 + t6)$ over an integer set
- Straightforward to accomplish via polyhedral libraries
  - ISL: just create an isl map
  - Polylib: use polylib image function or projections

Computing data accessed – parametric

- What we are interested in: data accessed for a given t_1, t_2 for example
- Parametric in t_1, t_2, N (don't eliminate t_1, t_2 from the system)
- Yields data written to or being read in a given iteration

  For previous code, given t_1, t_2, N, we get:

  $1 \leq d_2 \leq N - 2$
  $max(1, 32t_2 - 31) \leq d_1 \leq min(T - 2, 32t_2 + 31)$
  $64t_2 - 32t_1 - 31 \leq d_1 \leq 64t_2 - 32t_1 + 31$
  $-31 \leq 32t1 - 32t2 \leq N - 1$
- d_1 can be bounded

Computing data accessed – parametric

- What we are interested in: data accessed for a given t_1, t_2 for example
- Parametric in t_1, t_2, N (don't eliminate t_1, t_2 from the system)
- Yields data written to or being read in a given iteration

  For previous code, given t_1, t_2, N, we get:

  $1 \leq d_2 \leq N - 2$
  $max(1, 32t_2 - 31) \leq d_1 \leq min(T - 2, 32t_2 + 31)$
  $64t_2 - 32t_1 - 31 \leq d_1 \leq 64t_2 - 32t_1 + 31$
  $-31 \leq 32t1 - 32t2 \leq N - 1$
- d_1 can be bounded

Past approaches

1. Access function based [dHPF PLDI'98, Griebl-Classen IPDPS'06]
2. Dependence-based [Amarasinghe-Lam PLDI'93]

Our approach is dependence-based

- + Dependence information is already available (last writer property would mean some of the analysis need not be redone)
- + Natural
- − May not be the right granularity

Past approaches

1. Access function based [dHPF PLDI'98, Griebl-Classen IPDPS'06]
2. Dependence-based [Amarasinghe-Lam PLDI'93]

Our approach is dependence-based

- + Dependence information is already available (last writer property would mean some of the analysis need not be redone)
- + Natural
- − May not be the right granularity

Pluto-distmem: Dependences and Communication Sets

- Flow dependences lead to communication (anti and output dependences do not)
- The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile
- The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile
- Construct flow-out sets using flow dependences

Pluto-distmem: Dependences and Communication Sets

- Flow dependences lead to communication (anti and output dependences do not)
- The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile
- The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile
- Construct flow-out sets using flow dependences

Pluto-distmem: Dependences and Communication Sets

- Flow dependences lead to communication (anti and output dependences do not)
- The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile
- The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile
- Construct flow-out sets using flow dependences

Pluto-distmem: Dependences and Communication Sets

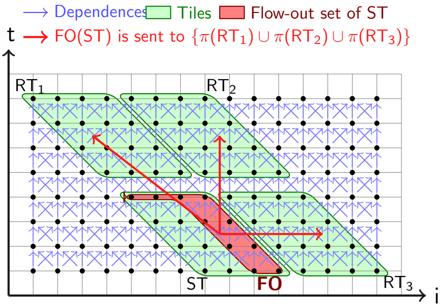
- Flow dependences lead to communication (anti and output dependences do not)
- The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile
- The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile
- Construct flow-out sets using flow dependences

Flow-out set

```
for (t=1; t<=T−1; t++)
  for (j=1; j<=N−1; j++)
    u[t%2][j] = 0.333∗(u[(t−1)%2][j−1] + u[(t−1)%2][j] + u[(t−1)%2][j+1]);
```

Flow-out set

```
for (t=1; t<=T−1; t++)
  for (j=1; j<=N−1; j++)
    u[t%2][j] = 0.333*(u[(t−1)%2][j−1] + u[(t−1)%2][j] + u[(t−1)%2][j+1]);
```



Computing flow-out set for variable x

Input Depth of parallel loop: l; set $\mathbf{S_w}$ of $\langle$write access, statement$\rangle$ pairs for variable x

$F^x_{out} = \emptyset$

for each $\langle M_w, S_i \rangle \in \mathbf{S_w}$ **do**

  for each dependence $e(S_i \rightarrow S_j) \in E$ **do**

    if e is of type RAW and source access of e is M_w **then**

      $E_l = \left\{ t^i_1 = t^j_1 \ \wedge \ t^i_2 = t^j_2 \ \wedge \ \ldots \ \wedge \ t^i_l = t^j_l \right\}$

      $C^t_e = D^T_e \cap E_l$

      $I^t_e = project\_out \left(C^t_e, m_{S_i} + 1, m_{S_j} \right)$

      $O^t_e = project\_out \left(D^T_e, m_{S_i} + 1, m_{S_j} \right) \ \setminus \ I^t_e$

      $F^x_{out} = F^x_{out} \cup \mathcal{I}_p(M^{S_i}_w, O^t_e, l)$

    end if

  end for

end for

Output F^x_{out}

Determining communication partners

1. A compiler-assisted runtime technique

   Define two functions as part of the output code for each data variable, x. If $t_1, \ldots, t_l$ is the set of sequential dimensions surrounding parallel dimension t_p:

2. $\sigma_x(t_1, t_2, \ldots, t_l, t_p)$: set of processors that need the flow-out set for data variable x from the processor calling this function

3. $\pi(t_1, t_2, \ldots, t_l, t_p)$: rank of processor that executes $(t_1,\ t_2,\ \ldots,\ t_l,\ t_p)$

Determining communication partners

1. A compiler-assisted runtime technique

   Define two functions as part of the output code for each data variable, x. If $t_1, \ldots, t_l$ is the set of sequential dimensions surrounding parallel dimension t_p:

2. $\sigma_x(t_1, t_2, \ldots, t_l, t_p)$: set of processors that need the flow-out set for data variable x from the processor calling this function
3. $\pi(t_1, t_2, \ldots, t_l, t_p)$: rank of processor that executes $(t_1, t_2, \ldots, t_l, t_p)$

Determining communication partners

1. A compiler-assisted runtime technique

   Define two functions as part of the output code for each data variable, x. If $t_1, \ldots, t_l$ is the set of sequential dimensions surrounding parallel dimension t_p:

2. $\sigma_x(t_1, t_2, \ldots, t_l, t_p)$: set of processors that need the flow-out set for data variable x from the processor calling this function

3. $\pi(t_1, t_2, \ldots, t_l, t_p)$: rank of processor that executes $(t_1, t_2, \ldots, t_l, t_p)$

Determining communication partners

1. A compiler-assisted runtime technique

   Define two functions as part of the output code for each data variable, x. If $t_1, \ldots, t_l$ is the set of sequential dimensions surrounding parallel dimension t_p:

2. $\sigma_x(t_1, t_2, \ldots, t_l, t_p)$: set of processors that need the flow-out set for data variable x from the processor calling this function
3. $\pi(t_1, t_2, \ldots, t_l, t_p)$: rank of processor that executes $(t_1,\ t_2,\ \ldots,\ t_l,\ t_p)$

The sigma function

- Dependence: a relation between source and target iterations ($\vec{s} \rightarrow \vec{t}$)
- For each such RAW dependence: $(s_1, s_2, \ldots, s_p, \ldots, s_m) \rightarrow (t_1, t_2, \ldots, t_p, \ldots, t_m)$
- Project out intra-tile iterators to obtain inter-tile dependences: $(s_1, s_2, \ldots, s_p) \rightarrow (t_1, t_2, \ldots, t_p)$
- Scanning $(t_1, t_2, \ldots, t_p)$ parametric in $(s_1, s_2, \ldots, s_p)$ enumerates receiver tiles for a given sending tile
- Apply π function to determine your receivers
- Code generated at compile-time: at runtime, we have the identities of the receivers for a flexible π

The sigma function

- Dependence: a relation between source and target iterations $(\vec{s} \rightarrow \vec{t})$
- For each such RAW dependence: $(s_1, s_2, \ldots, s_p, \ldots, s_m) \rightarrow (t_1, t_2, \ldots, t_p, \ldots, t_m)$
- Project out intra-tile iterators to obtain inter-tile dependences: $(s_1, s_2, \ldots, s_p) \rightarrow (t_1, t_2, \ldots, t_p)$
- Scanning $(t_1, t_2, \ldots, t_p)$ parametric in $(s_1, s_2, \ldots, s_p)$ enumerates receiver tiles for a given sending tile
- Apply π function to determine your receivers
- Code generated at compile-time: at runtime, we have the identities of the receivers for a flexible π

The sigma function

- Dependence: a relation between source and target iterations ($\vec{s} \rightarrow \vec{t}$)
- For each such RAW dependence:
  $(s_1, s_2, \ldots, s_p, \ldots, s_m) \rightarrow (t_1, t_2, \ldots, t_p, \ldots, t_m)$
- Project out intra-tile iterators to obtain inter-tile dependences:
  $(s_1, s_2, \ldots, s_p) \rightarrow (t_1, t_2, \ldots, t_p)$
- Scanning $(t_1, t_2, \ldots, t_p)$ parametric in $(s_1, s_2, \ldots, s_p)$ enumerates receiver tiles for a given sending tile
- Apply π function to determine your receivers
- Code generated at compile-time: at runtime, we have the identities of the receivers for a flexible π

The sigma function

- Dependence: a relation between source and target iterations ($\vec{s} \rightarrow \vec{t}$)
- For each such RAW dependence:
  $(s_1, s_2, \ldots, s_p, \ldots, s_m) \rightarrow (t_1, t_2, \ldots, t_p, \ldots, t_m)$
- Project out intra-tile iterators to obtain inter-tile dependences:
  $(s_1, s_2, \ldots, s_p) \rightarrow (t_1, t_2, \ldots, t_p)$
- Scanning $(t_1, t_2, \ldots, t_p)$ parametric in $(s_1, s_2, \ldots, s_p)$ enumerates receiver tiles for a given sending tile
- Apply π function to determine your receivers
- Code generated at compile-time: at runtime, we have the identities of the receivers for a flexible π

The sigma function

- Dependence: a relation between source and target iterations $(\vec{s} \rightarrow \vec{t})$
- For each such RAW dependence: $(s_1, s_2, \ldots, s_p, \ldots, s_m) \rightarrow (t_1, t_2, \ldots, t_p, \ldots, t_m)$
- Project out intra-tile iterators to obtain inter-tile dependences: $(s_1, s_2, \ldots, s_p) \rightarrow (t_1, t_2, \ldots, t_p)$
- Scanning $(t_1, t_2, \ldots, t_p)$ parametric in $(s_1, s_2, \ldots, s_p)$ enumerates receiver tiles for a given sending tile
- Apply π function to determine your receivers
- Code generated at compile-time: at runtime, we have the identities of the receivers for a flexible π

Packing and unpacking data

- Use a linearized counted buffer

```
for (d0=max(max(1,32*t1-32*t3),32*t3-N+32);
     d0<=min(T-2,32*t1-32*t3+30);d0++) for
     d1=max(1,32*t3-d0+30);d1<=min(N-2,32*t3-d0+31);d1++) {
        send_buf_u[send_count_u++] = u[d0][d1];

      if (t1 <= min(floord(32*t3+T-33,32),2*t3-1)) {
        for (d1=-32*t1+64*t3-31;d1<=min(N-1,-32*t1+64*t3);d1++)
          send_buf_u[send_count_u++] = u[32*t1-32*t3+31][d1];
      }
}
```

- Unpacking – just reverse the assignment

Packing and unpacking data

- Use a linearized counted buffer

```
for (d0=max(max(1,32*t1-32*t3),32*t3-N+32);
     d0<=min(T-2,32*t1-32*t3+30);d0++) for
     d1=max(1,32*t3-d0+30);d1<=min(N-2,32*t3-d0+31);d1++) {
         send_buf_u[send_count_u++] = u[d0][d1];

      if (t1 <= min(floord(32*t3+T-33,32),2*t3-1)) {
        for (d1=-32*t1+64*t3-31;d1<=min(N-1,-32*t1+64*t3);d1++)
          send_buf_u[send_count_u++] = u[32*t1-32*t3+31][d1];
      }
}
```

- Unpacking – just reverse the assignment

Packing and unpacking data

- Use a linearized counted buffer

```
for (d0=max(max(1,32*t1-32*t3),32*t3-N+32);
     d0<=min(T-2,32*t1-32*t3+30);d0++) for
     d1=max(1,32*t3-d0+30);d1<=min(N-2,32*t3-d0+31);d1++) {
        send_buf_u[send_count_u++] = u[d0][d1];

     if (t1 <= min(floord(32*t3+T-33,32),2*t3-1)) {
       for (d1=-32*t1+64*t3-31;d1<=min(N-1,-32*t1+64*t3);d1++)
         send_buf_u[send_count_u++] = u[32*t1-32*t3+31][d1];
     }
}
```

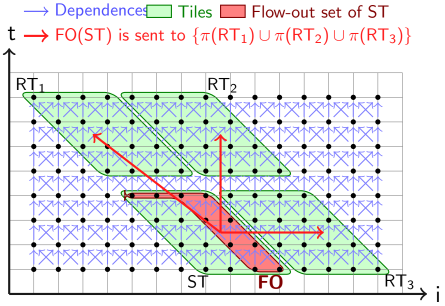
- Unpacking – just reverse the assignment

Determining Communication Partners

$$\sigma_x(s_1, s_2, \ldots, s_l, s_p) = \{\pi(t_1, t_2, \ldots, t_l, t_p) \mid \exists e \in E \text{ on } x, D_e^T(s_1, .., s_p, .., t_1, .., t_p, .., \vec{p}, 1)\}$$

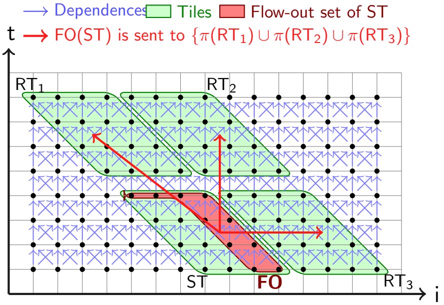
D_e^T is the dependence polyhedron corresponding to e

Strengths and Limitations



- \+ Good for broadcast or multicast style communication
- \+ A processor will never receive the same data twice
- – Okay for disjoint point-to-point communication
- – A processor could be sent data that it does not need

Strengths and Limitations



- + Good for broadcast or multicast style communication
- + A processor will never receive the same data twice
- − Okay for disjoint point-to-point communication
- − A processor could be sent data that it does not need

Sub-problems

1. Constructing communication sets
2. Packing and unpacking data
3. Determining receivers
4. Generating actual communication primitives

Improvement over previous approaches

- Based on last-writer dependences, more precise
- Avoids redundant communication due to virtual-physical processor mapping in several cases
- Works with all polyhedral transformations on affine loop nests
- Further refinements possible: flow-out intersection flow-in, flow-out set partitioning, and data movement for heterogeneous systems (CPU/GPU) [Dathathri et al. PACT 2013]

Improvement over previous approaches

- Based on last-writer dependences, more precise
- Avoids redundant communication due to virtual-physical processor mapping in several cases
- Works with all polyhedral transformations on affine loop nests
- Further refinements possible: flow-out intersection flow-in, flow-out set partitioning, and data movement for heterogeneous systems (CPU/GPU) [Dathathri et al. PACT 2013]

Driven by Computation / Data flow

- Code generation is for a given computation transformation / distribution
- Data moves as dictated by (last-writer) dependences for the computation partitioning specified
- There is no owning processor for data
- Data distribution only affects communication at start, and is needed for weak scaling and allocation purposes
- We use a push model (synchronous with clear separation between computation and communication phases)

Driven by Computation / Data flow

- Code generation is for a given computation transformation / distribution
- Data moves as dictated by (last-writer) dependences for the computation partitioning specified
- There is no owning processor for data
- Data distribution only affects communication at start, and is needed for weak scaling and allocation purposes
- We use a push model (synchronous with clear separation between computation and communication phases)

Experimental evaluation

- Code generation support implemented in the Pluto tool (http://pluto-compiler.sourceforge.net)
- Experiments on a 32-node InfiniBand cluster running MVAPICH2 (running 1 process per node)
- Codes experimented capture different communication styles (near-neighbor, broadcast style, multicast style)
- All codes automatically transformed
- Generated codes were compiled with *icc -fast (-O3 -ipo -static)* version 11.1

Performance summary

Benchmark	*seq* (icc)	*pluto-seq*	Execution time for *our* (number of procs)						Speedup: *our-32*	
			1	2	4	8	16	32	*seq*	*our-1*
strmm	30.4m	247s	240s	124.6s	63.5s	33.6s	17.3s	9.4s	194	26.3
trmm	35.5m	91.8s	96.4s	51.3s	27.4s	15.3s	7.14s	3.74s	570	24.5
dsyr2k	127s	39s	38.8s	22.4s	13.5s	6.80s	3.80s	1.57s	80.8	24.7
covcol	462s	30.9s	30.7s	16.7s	8.8s	4.60s	2.48s	1.30s	355	23.8
seidel	17.3m	643.5s	692s	338.7s	174.3s	94s	65.6s	33.0s	31.0	20.8
jac-2d	21.9m	206.7s	218s	111.2s	62.3s	40.7s	29.3s	21.5s	61.3	9.6
fdtd-2d	139s	129.7s	95.2s	70.7s	40.3s	25.3s	16.8s	11.7s	11.9	11.0
2d-heat	19m	266s	280s	157s	81s	52s	33s	24.0s	47.5	11.7
3d-heat	590.6s	222s	236s	118s	68.7s	41.5s	26.3s	18.8s	31.4	12.6
lu	82.9s	28s	29.5s	18.8s	9.28s	5.67s	4.3s	3.9s	21.3	7.56
floyd-warshall	2012s	2012s	2062s	1041s	527s	273s	153s	112s	18.0	18.0

- Mean (geometric) speedup of 60.7× over icc-seq and of 15.9× over pluto-seq
- A more detailed comparison with manually written code and HPF in the paper
- Often hard to write such code by hand even for simple affine loop nests (non-rectangularlity, tiling, discontiguity)

Tool available (BETA)

Available publicly at: http://pluto-compiler.sourceforge.net

$ ../../polycc floyd.c –distmem –commreport –mpiomp –tile –isldep –lastwriter –cloogsh -o seidel.distopt.c

$ mpicc -O3 -openmp floyd.distopt.c sigma.c pi.c -o distopt -lpolyrt -lm

DISCLAIMER: beta release, not responsible for crashing your cluster!

Conclusions and future work

- First source-to-source tool for MPI code generation for affine loop nests
- Improves over previous distributed memory code generation approaches
- When coupled with prior work in polyhedral transformation, a fully automatic distributed-memory parallelizer
- Future work: integrating it with dynamic scheduling runtimes and enabling *data-flow style parallelization*: asynchronous communication and overlap of computation and communication, load balance come free of cost

Conclusions and future work

- First source-to-source tool for MPI code generation for affine loop nests
- Improves over previous distributed memory code generation approaches
- When coupled with prior work in polyhedral transformation, a fully automatic distributed-memory parallelizer
- Future work: integrating it with dynamic scheduling runtimes and enabling *data-flow style parallelization*: asynchronous communication and overlap of computation and communication, load balance come free of cost

Thank you

Questions?

Acknowledgments

- AMD for an unrestricted research grant (2011–)
- Department of Science and Technology (India) for a grant under the FIST program