Optimizing Sequences of Skeleton Calls

Herbert Kuchen

University of Münster

1. Why Skeletons?

- parallel programming today: SPMD on top of MPI
  - ⋆ local view: $\longrightarrow$ deadlocks,. . .
  - ⋆ optimization difficult
- higher level of parallel programming
  - ⋆ BSP: supersteps
  - ⋆ algorithmic skeletons
  - ⋆ . . .

Algorithmic Skeletons

- typical parallel programming patterns
- provided as polymorphic higher-order functions

Our Skeleton Library

- skeletons provided as a C++ library ($\rightarrow$ polymorphism, HOFs)
- two-tier model as in P^3L
- data parallel skeletons as in Skil
- implementation on top of MPI: $\rightarrow$ platform independence

2. Data Parallel Skeletons

- based on parallel operations manipulating distributed data structures

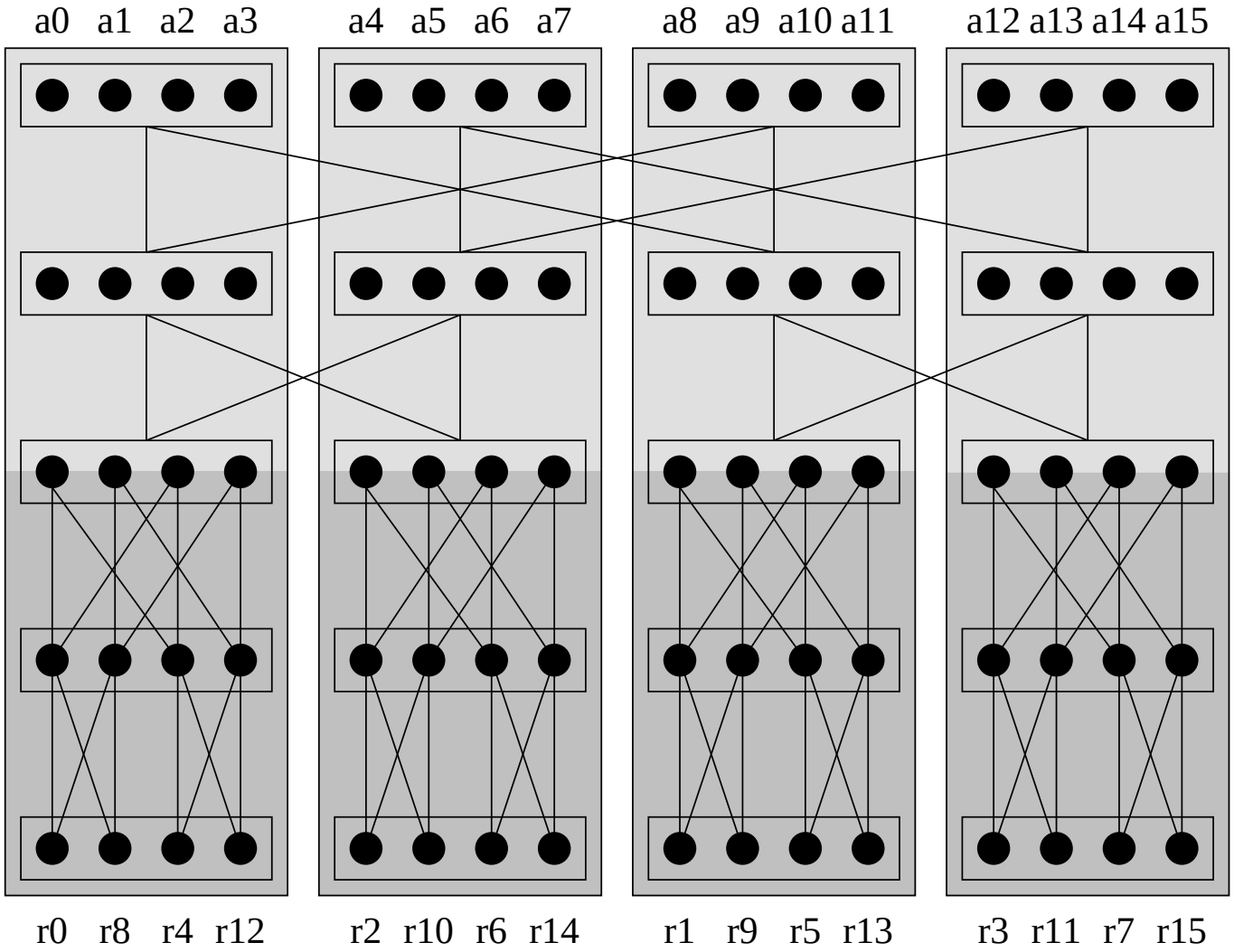
```
template <class E> class DistributedArray{...}
template <class E> class DistributedMatrix{...}
```

- DDS is processed as a whole (e.g. using map, rotate)
- coordinated communication $\longrightarrow$ no deadlocks
- parallel interleaved with sequential operations
- data parallel skeletons:
  - ⋆ computation: map, fold, zip, scan . . .
  - ⋆ communication: rotate, broadcast, permute, gather, all2all, . . .
  - ⋆ no implicit communication!
  - ⋆ auxiliary operations: get, getFirstRow, getLocalSize, . . .

Example: Matrix Multiplication

```
inline int negate(const int a){return -a;}

template <class C, int n>
C sprod(const DistributedMatrix<C,n,n>& A,
        const DistributedMatrix<C,n,n>& B, int i, int j, C Cij){
  C sum = Cij;
  for (int k=0; k<A.getLocalRows(); k++)
    sum += A.get(i,k+A.getFirstCol()) * B.get(k+B.getFirstRow(),j);
  return sum;}

template <class C, int n>
DistributedMatrix<C,n,n> matmult(DistributedMatrix<C,n,n> A,
                                 DistributedMatrix<C,n,n> B){
  A.rotateRows(negate);
  B.rotateCols(negate);
  DistributedMatrix<C,n,n> R(0,A.getBlocksInCol(),A.getBlocksInRow());
  for (int i=0; i< A.getBlocksInRow(); i++){
    R.mapIndexInPlace(curry(sprod<C,n>)(A)(B));
    A.rotateRows(-1);
    B.rotateCols(-1);}
  return R;}
```

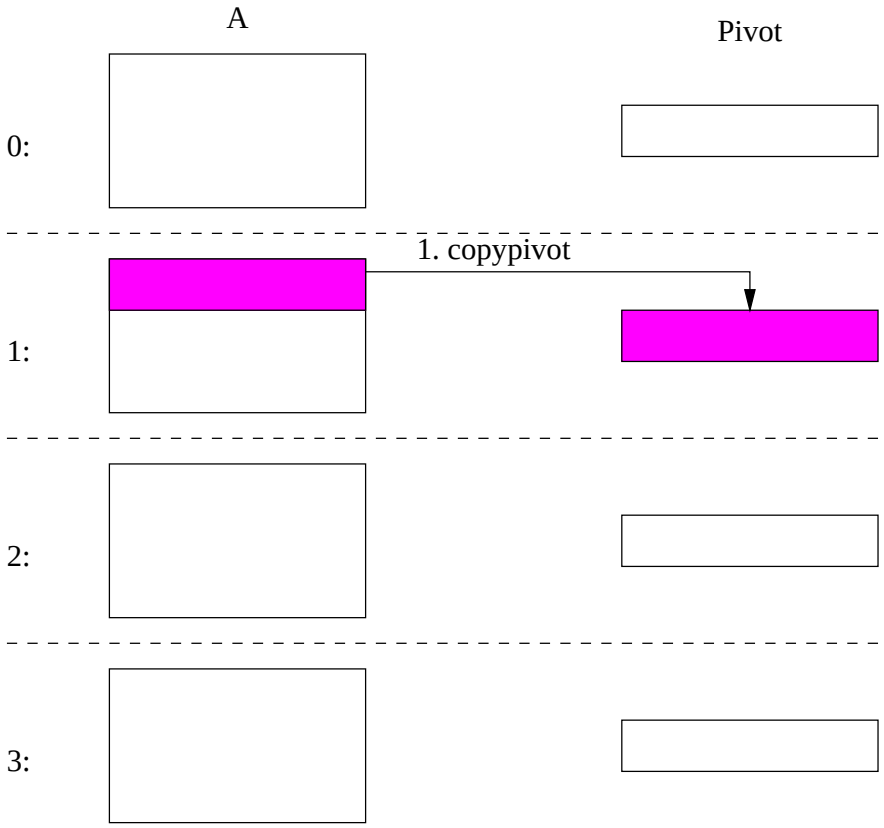
Example: FFT

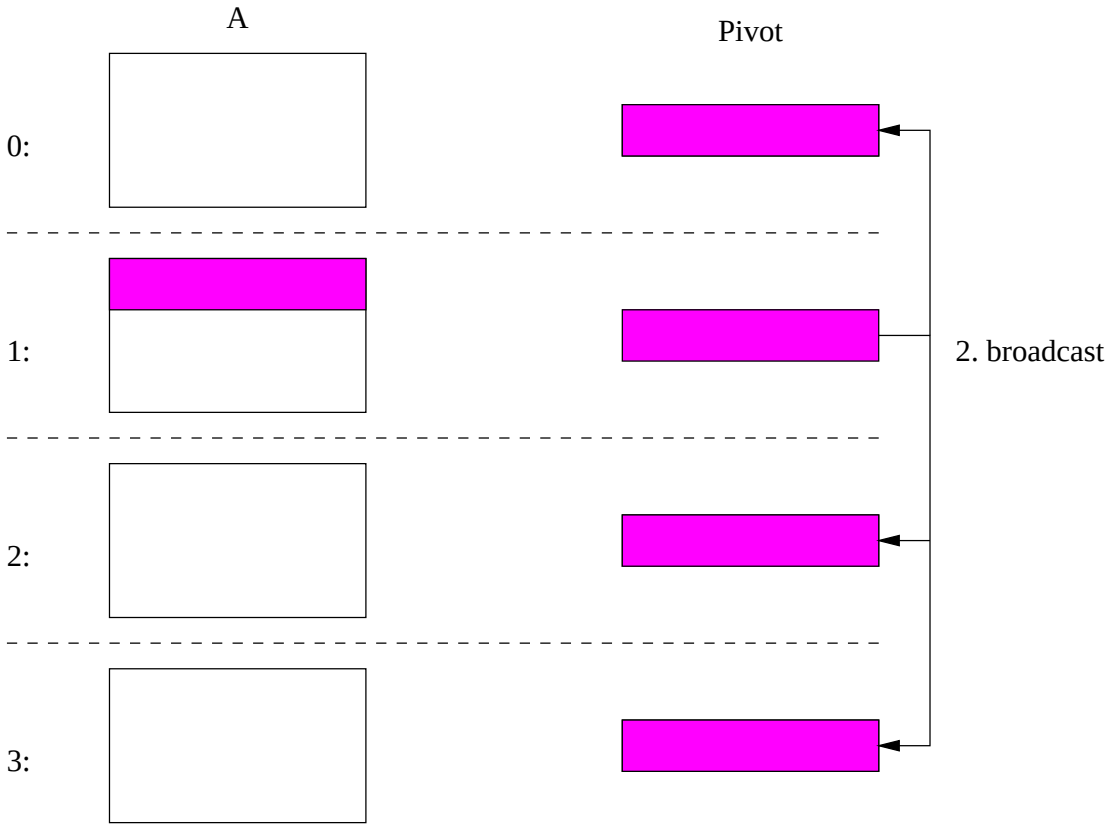

a0 a1 a2 a3
a4 a5 a6 a7
a8 a9 a10 a11
a12 a13 a14 a15
r0 r8 r4 r12
r2 r10 r6 r14
r1 r9 r5 r13
r3 r11 r7 r15

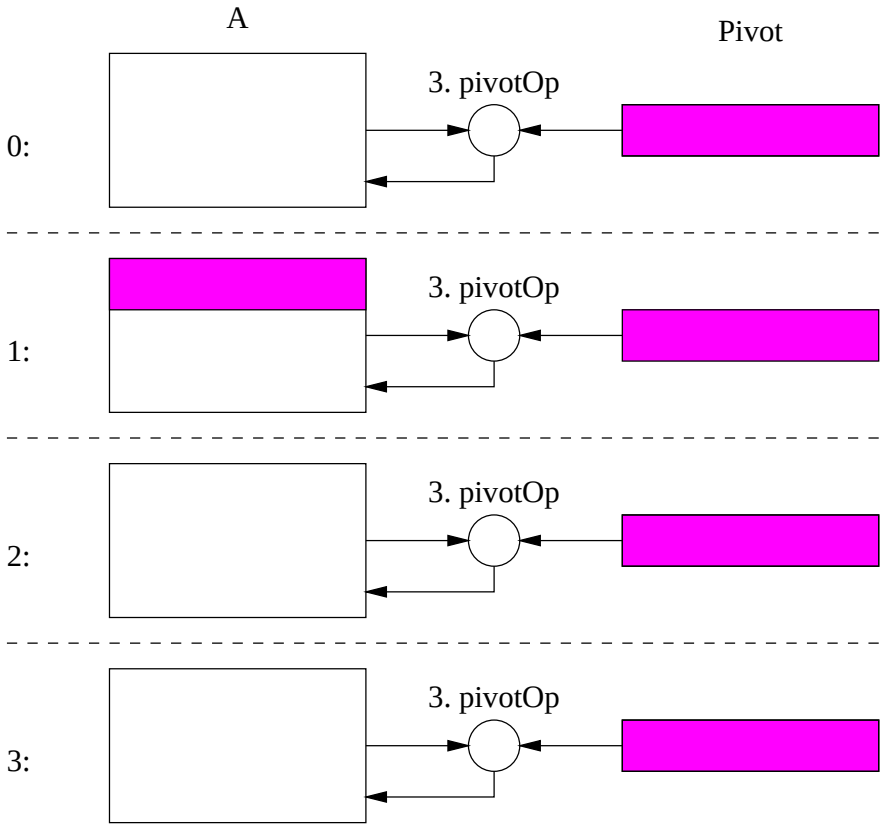
FFT with Data Parallel Skeletons

```
struct complex{...} ... with operators +,*,init ...

inline int bitcomplement(int k, int i){return i ^ (1<<k);}

inline complex omega(int j, int i){
  int b = i >> (log2n-j-1);   int b2 = 0;
  for (int l=0; l<=j; l++){b2 = (b & 1) ? 2*b2+1 : 2*b2;  b >>= 1;}
  double v = 2.0 * 3.141592653589/ n * (b2 << (log2n-j-1));
  return complex(cos(v), sin(v));}

inline complex combine(const DistributedArray<complex>& T, int j, int i, complex Ri){
  return (i & (1<<log2n-1-j)) ? T.get(i) + omega(j,i) * Ri : Ri + omega(j,i) * T.get(i);}

inline complex fetch(const DistributedArray<complex>& R, int j, int i, complex Ti){
  return R.get(bitcomplement(log2n-1-j,i));}

DistributedArray<complex> fft(const DistributedArray<double>& A){int j;
    DistributedArray<complex> R(n,curry(init)(A));
    DistributedArray<complex> T(n);
    for (j=0; j<log2p; j++) { T = R.copy();
      T.permutePartition(curry(bitcomplement)(log2p-1-j));
      R.mapIndexInPlace(curry(combine)(T)(j));}
    for (j=log2p; j<log2n; j++){
      T.mapIndexInPlace(curry(fetch)(R)(j));
      R.mapIndexInPlace(curry(combine)(T)(j));}
    return R;}
```

Example: Gaussian Elimination




A
Pivot
0:
1:
2:
3:
2. broadcast


A
Pivot
0:
3. pivotOp
1:
3. pivotOp
2:
3. pivotOp
3:
3. pivotOp

Gaussian Elimination with Data Parallel Skeletons

```
double copyPivot(const DistributedMatrix<double>& A,
                 int k, int i, int j, double Pij){
  return A.isLocal(k,k) ? A.get(k,j)/A.get(k,k) : 0;}

void pivotOp(const DistributedMatrix<double>& Pivot,
             int rows, int firstrow, int k, double** LA){
  for (int l=0; l<rows; l++){
    double Alk = LA[l][k];
    for (int j=k; j<=Problemsize; j++)
      if (firstrow+l == k) LA[l][j] = Pivot.getLocalGlobal(0,j);
      else  LA[l][j] -=  Alk * Pivot.getLocalGlobal(0,j);}}

void gauss(DistributedMatrix<double>& A){
  DistributedMatrix<double> Pivot(sk_numprocs,n+1,0.0,p,1);
  for (int k=0; k<Problemsize; k++){
    Pivot.mapIndexInPlace(curry(copyPivot)(A)(k));
    Pivot.broadcastRow(k);
    A.mapPartitionInPlace(curry(pivotOp)(Pivot,n/p,A.getFirstRow(),k));}}
```

3. Implementation Issues

- replicated sequential computations
- map, zip: $O(1)$
- fold, scan, broadcast: $O(\log p)$
- permutation, rotation: $O(1)$

4. Optimizations

- global view simplifies global optimizations
- seqquences of skeleton calls replaced by better sequences

```
A.mapIndexInPlace(f);
A.mapIndexInPlace(g);
```

$\longrightarrow$

```
A.mapIndexInPlace(curry(compose)(g)(f));
```

4. Optimizations

- global view simplifies global optimizations
- seqquences of skeleton calls replaced by better sequences

```
A.mapIndexInPlace(f);
A.mapIndexInPlace(g);
```
$\longrightarrow$
```
A.mapIndexInPlace(curry(compose)(g)(f));
```

- similar to loop fusion
- precondition: independent loops

```
for(int i = 0; i<localsize; i++)
   a[i] = f(i+first,a[i]);
for(int i = 0; i<localsize; i++)
   a[i] = g(i+first,a[i]);
```
$\longrightarrow$
```
for(int i = 0; i<localsize; i++)
  a[i] = g(i+first,f(i+first,a[i]));
```

Another Example

```
A.mapIndexInPlace(f);
result = A.fold(g);
```

$\longrightarrow$

```
result = A.homomorphism(f,g);
```

- requires new skeleton `homomorphism`

Experimental Results

example	n	original	combined	speedup
`permutePartition` ∘ `permutePartition`	2^7	37.00 μs	16.43 μs	2.25
`mapIndexInPlace` ∘ `mapIndexInPlace`	2^7	0.69 μs	0.40 μs	1.73
`mapIndexInPlace` ∘ `fold`	2^7	144.82 μs	144.60 μs	1.00
`mapIndexInPlace` ∘ `scan`	2^7	144.52 μs	144.48 μs	1.00
`multiMapIndexInPlace`	2^7	0.67 μs	0.65 μs	1.03
`mapIndexInPlace` ∘ `permutePartition`	2^7	16.60 μs	31.43 μs	0.53
`permutePartition` ∘ `permutePartition`	2^{19}	10.76 ms	5.32 ms	2.02
`mapIndexInPlace` ∘ `mapIndexInPlace`	2^{19}	2.79 ms	1.59 ms	1.75
`mapIndexInPlace` ∘ `fold`	2^{19}	2.97 ms	2.67 ms	1.11
`mapIndexInPlace` ∘ `scan`	2^{19}	6.01 ms	5.83 ms	1.03
`multiMapIndexInPlace`	2^{19}	2.77 ms	2.62 ms	1.06
`mapIndexInPlace` ∘ `permutePartition`	2^{19}	6.68 ms	5.72 ms	1.17

Table 1: Runtimes on 4 processors for sequences of skeletons working on a distributed array of n elements and corresponding speedups.

- fusion of commmunication and computation skeletons enables speedup ≤ 2

Experimental Results (continued)

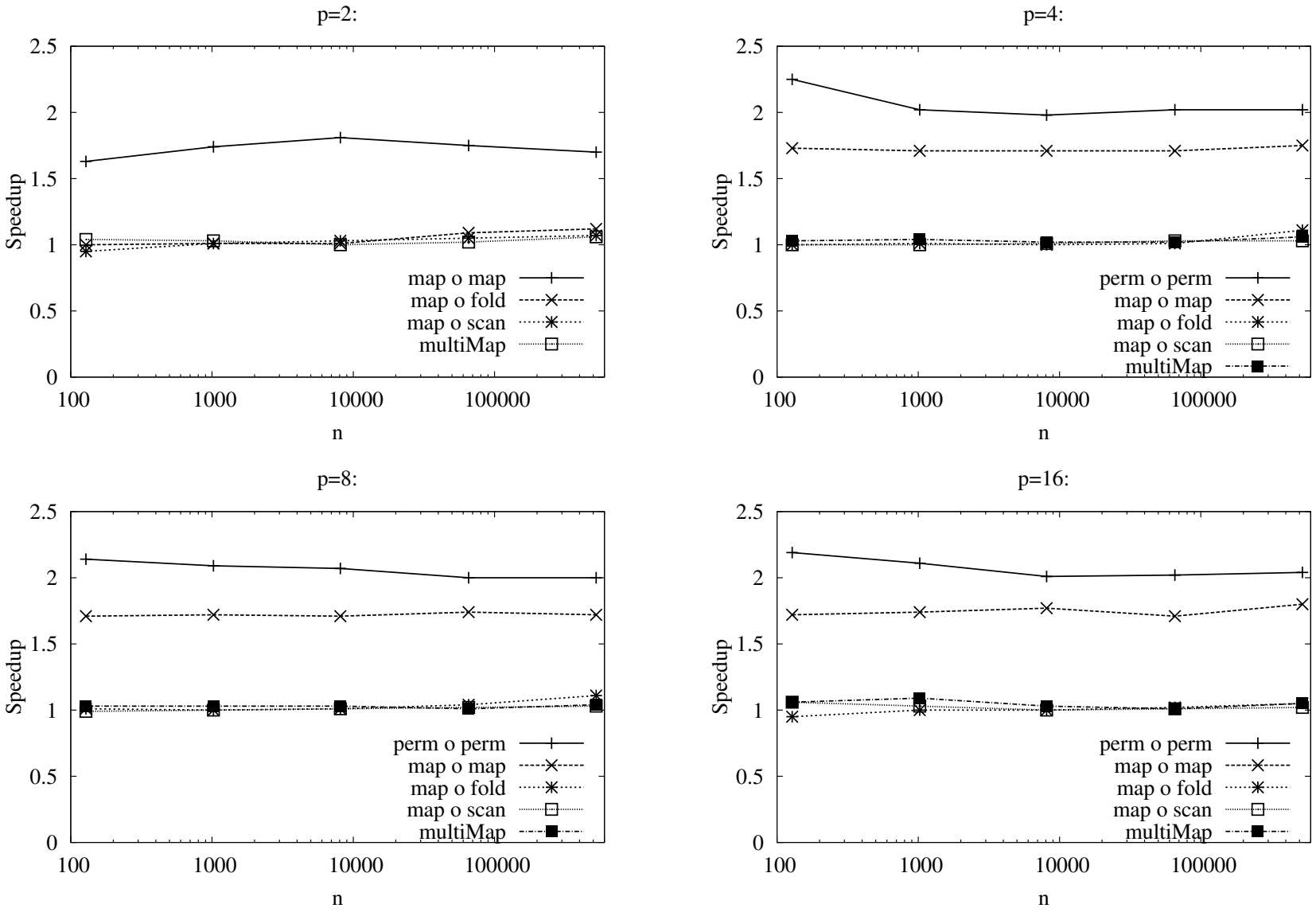


Figure 1: Speedups for the combination of skeletons on 2, 4, 8, and 16 processors.

Combining Communication Skeletons on Arrays

original	combined
`A.permute(f);` `A.permute(g);`	`A.permute(curry(compose)(g)(f));` (analogously for `permutePartition`)
`A.permute(f);` `A.broadcast(k);`	`A.broadcast(`$\mathtt{f}^{-1}$`(k));` (analogously for `permutePartition`)
`A.broadcast(k);` `A.permute(f);`	`A.broadcast(k);` (analogously for `permutePartition`)
`A.broadcast(k);` `A.broadcast(i);`	`A.broadcast(k);`

- seldom occurs in practice

Combining Communication Skeletons on Matrices

`M.permutePartition(f,g);` `M.rotateRows(h);`	`M.permutePartition(curry(compose)(h')(f),g);`
`M.rotateRows(h);` `M.permutePartition(f,g);`	`M.permutePartition(curry(compose)(f)(h'),g);`
`M.permutePartition(f,g);` `M.rotateCols(h);`	`M.permutePartition(f,curry(compose)(h')(g));`
`M.rotateCols(h);` `M.permutePartition(f,g);`	`M.permutePartition(f,curry(compose)(g)(h'));`
`M.broadcast(i,j);` `M.rotateRows(h);`	`M.broadcast(i,j);` (analogously for `rotateCols`)
`M.rotateRows(h);` `M.broadcast(i,j);`	`M.broadcast(h'`$^{-1}$`(i),j);` (analogously for `rotateCols`)
`M.rotateRows(f);` `M.rotateRows(g);`	`M.rotateRows(curry(compose)(g)(f));` (analogously for `rotateCols`)
`M.rotateRows(f);` `M.rotateCols(g);`	`M.permutePartition(f',g');`
`M.rotateCols(f);` `M.rotateRows(g);`	`M.permutePartition(g',f');`

Experimental Result for Combination of Rotations

$n \setminus p$	4	8	16
16	1.64	1.91	1.89
128	1.71	1.92	1.70
1024	1.48	1.81	1.72

- speedup for $n \times n$ matrix on p processors

Impact of Argument Functions

k	n	original	combined	speedup
1	2^7	0.69 μs	0.40 μs	1.73
1000	2^7	21.23 μs	19.79 μs	1.07
1	2^{19}	2.79 ms	1.59 ms	1.75
1000	2^{19}	83.00 ms	80.12 ms	1.04

- combination of 2 $\times$ `mapIndexInPlace`
- argument function requires k iterations
- $\longrightarrow$ impact of skeleton fusion decreases for complex argument functions

Combining Maps on Different Arrays

- in practical applications: often sequences of maps on different arrays
- hence: new "skeleton" `multiMapIndexInPlace`

```
A.mapIndexInPlace(f);
B.mapIndexInPlace(g);
```

$\longrightarrow$

```
multiMapIndexInPlace(A,f,B,g);
```

5. Conclusions

- algorithmic skeletons provided in form of a C++ library
- data parallel skeletons manipulate a distributed data structure as a whole
- skeleton fusion
  - ⋆ reduces control overhead ($\hat{=}$ loop fusion)
  - ⋆ allows to overlap communication and computation