BLUE WATERS

SUSTAINED PETASCALE COMPUTING

Performance-oriented Parallel Programming

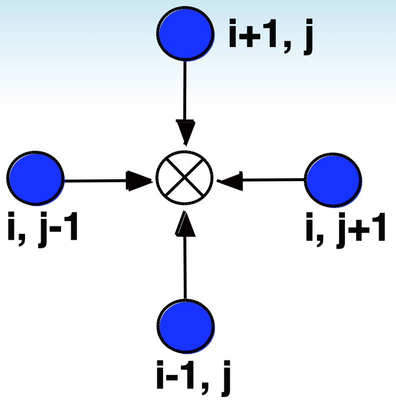
Integrating Hardware, Middleware, and Applications

Torsten Hoefler

15th SIAM Conference on Parallel Processing for Scientific Computing
SIAG Junior Scientist Prize Award and Lecture

GREAT LAKES CONSORTIUM FOR PETASCALE COMPUTATION

How to write efficient code?



- Simplified 5-point (2D) stencil
  - Represents other stencils

```
for(int i=1; i<n; ++i)
  for(int j=1; j<n; ++j)
    a[i,j] = b[i,j]+(b[i-1,j]+b[i+1,j]+b[i,j-1]+b[i,j+1])/4.0;
```

3 SLOC

- Simple code, easy to read
  - Very slow to execute → 150 MF/s (peak ~18 GF/s)

Serial Code Transformations

unroll-and-jam, vectorization, prefetch, nt stores, reg blocking, alignment

```
for(int i=1; i<n; ++i)
  for(int j=1; j<n; ++j)
    a[i,j] = b[i,j]+(b[i-1,j]+b[i+1,j]+b[i,j-1]+b[i,j+1])/4.0;
```

3 SLOC

Tuning →

```
for(int ib=1; ib<n+1; ib+=bs)
  for(int jb=1; jb<n+1; jb+=bs)
    for(register int i=ib; i<min(ib+bs, n+1); ++i)
      #pragma ivdep
      for(register int j=jb; j<min(jb+bs,n+1); j+=2) {
        register double x=anew[ind(i,j)]/2.0;
        register double y=anew[ind(i,j+1)]/2.0;
        register double z=anew[ind(i,j+2)]/2.0;
        x = x + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold[ind(i,j-1)] + aold[ind(i,j+1)])/8.0;
        y = y + (aold[ind(i-1,j+1)] + aold[ind(i+1,j+1)] + aold[ind(i,j)] + aold[ind(i,j+2)])/8.0;
        z = z + (aold[ind(i-1,j+2)] + aold[ind(i+1,j+2)] + aold[ind(i,j+1)] + aold[ind(i,j+3)])/8.0;
        heat += x + y + z;
        anew[ind(i,j)] = x;
        anew[ind(i,j+1)] = y;
        anew[ind(i,j+2)] = z;
      }
```

121 SLOC

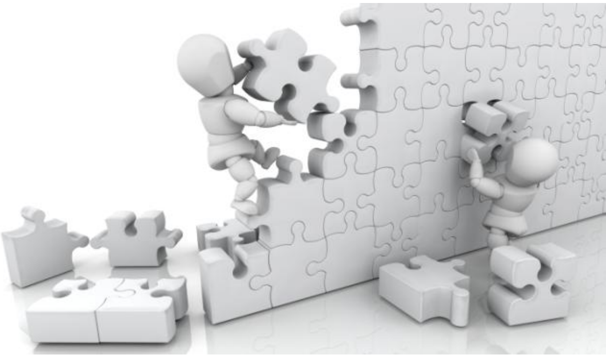
150 MF/s **1.32 GF/s**

- Huge programming effort (5 minutes vs. 5 hours)
  - Very hard to read and change!
  - Not portable (techniques and constants differ!)
- Best automatic compiler optimization (+manual optimization):

GCC 4.6.1	PGCC 11.9	PathCC 4.0.9	CrayC 7.4.4
0.66 GF/s (18%)	0.66 GF/s (16%)	1.22 GF/s (8%)	0.38 GF/s (-35%)

Data collected on Hector/UK

Serial Optimization is well understood



- Follows higher-order principles
  - Autotuning for specific parameters
  - CS help needed to drive the effort
- Compilers improve significantly
  - Optimizations improve performance 10x
  - Humans or domain-specific (auto)tuning beat them[1]
- Parallelism becomes ubiquitous
  - Key question: How do we make parallel programming **as** "easy" as serial programming?

[1]: Tang et al. "The Pochoir Stencil Compiler" (SPAA 2011)

Parallel 2D Stencil in “six-function” MPI

- Simple 2D stencil

```
// figure out my coordinates (x,y) -- r=x*a+y in the 2d processor array
  int rx = r % px;
  int ry = r / px;
  // figure out my four neighbors
  int north = (ry-1)*px+rx; if(ry-1 < 0)   north = MPI_PROC_NULL;
  int south = (ry+1)*px+rx; if(ry+1 >= py) south = MPI_PROC_NULL;
  int west= ry*px+rx-1;     if(rx-1 < 0)   west = MPI_PROC_NULL;
  int east = ry*px+rx+1;    if(rx+1 >= px) east = MPI_PROC_NULL;
  // decompose the domain
  int bx = n/px; // block size in x
  int by = n/py; // block size in y
  int offx = rx*bx; // offset in x
  int offy = ry*by; // offset in y

  MPI_Request reqs[8];
  double *sbufnorth = (double*)calloc(1,bx*sizeof(double)); // send buffers
  double *sbufsouth = (double*)calloc(1,bx*sizeof(double));
  double *sbufeast = (double*)calloc(1,by*sizeof(double));
  double *sbufwest = (double*)calloc(1,by*sizeof(double));
  double *rbufnorth = (double*)calloc(1,bx*sizeof(double)); // receive buffers
  double *rbufsouth = (double*)calloc(1,bx*sizeof(double));
  double *rbufeast = (double*)calloc(1,by*sizeof(double));
  double *rbufwest = (double*)calloc(1,by*sizeof(double));
  for(int i=0; i<bx; ++i) sbufnorth[i] = aold[ind(i+1,1)]; // pack loop - last valid region
  for(int i=0; i<bx; ++i) sbufsouth[i] = aold[ind(i+1,by)]; // pack loop
  for(int i=0; i<by; ++i) sbufeast[i] = aold[ind(bx,i+1)]; // pack loop
  for(int i=0; i<by; ++i) sbufwest[i] = aold[ind(1,i+1)]; // pack loop
  MPI_Isend(sbufnorth, bx, MPI_DOUBLE, north, 9, comm, &reqs[0]);
  MPI_Isend(sbufsouth, bx, MPI_DOUBLE, south, 9, comm, &reqs[1]);
  MPI_Isend(sbufeast, by, MPI_DOUBLE, east, 9, comm, &reqs[2]);
  MPI_Isend(sbufwest, by, MPI_DOUBLE, west, 9, comm, &reqs[3]);
  MPI_Irecv(rbufnorth, bx, MPI_DOUBLE, north, 9, comm, &reqs[4]);
  MPI_Irecv(rbufsouth, bx, MPI_DOUBLE, south, 9, comm, &reqs[5]);
  MPI_Irecv(rbufeast, by, MPI_DOUBLE, east, 9, comm, &reqs[6]);
  MPI_Irecv(rbufwest, by, MPI_DOUBLE, west, 9, comm, &reqs[7]);
  MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);
  for(int i=0; i<bx; ++i) aold[ind(i+1,0)] = rbufnorth[i]; // unpack loop - into ghost cells
  for(int i=0; i<bx; ++i) aold[ind(i+1,by+1)] = rbufsouth[i]; // unpack loop
  for(int i=0; i<by; ++i) aold[ind(bx+1,i+1)] = rbufeast[i]; // unpack loop
  for(int i=0; i<by; ++i) aold[ind(0,i+1)] = rbufwest[i]; // unpack loop

  heat = 0.0;
  for(int i=1; i<bx+1; ++i) {
    for(int j=1; j<by+1; ++j) {
      anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
      heat += anew[ind(i,j)];
    }
  }
```

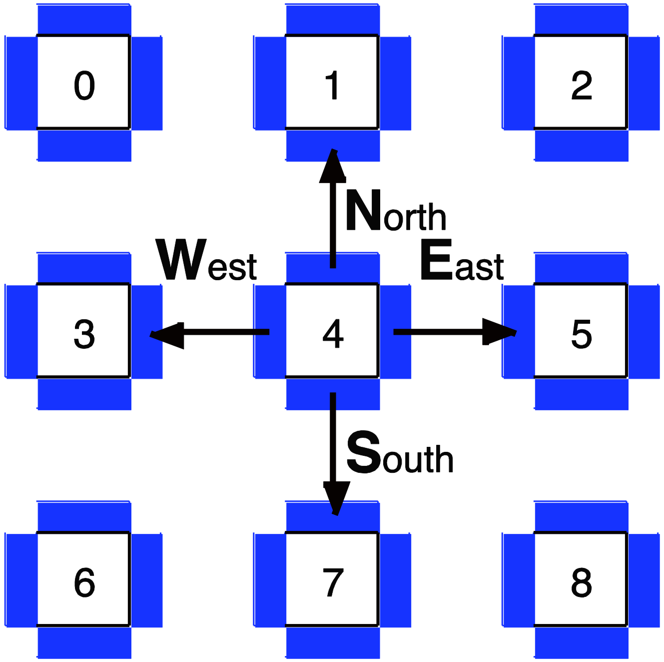
50 SLOC

26% decomposition

26% data movement

20% communication

4.67 GF (P=32)



- Packs and communicates four directions
  - Simple code, easy to read, slow execution

Parallel Optimizations are less understood

- Many techniques are known, applied manually
  - Often with good libraries (PETSc, ScaLAPACK, …)
  - CS needs to drive automation
- Identify necessary optimization techniques
  - Define the right abstractions
  - I will show several examples
- Define composable interfaces
  - CS researchers adapt programs and systems

Optimization Principles for Parallel Programs

1. Serialization/Deserialization of sent/recvd data
2. Optimizing communication patterns
   - Collective operations cf. “Communication BLAS”
3. Communication/computation overlap
   - Pipelining, cf. “Cache prefetch”
4. Synchronization and System Noise
5. Topology mapping
6. Scalable Algorithms and Load Balance
7. Domain Decomposition/Data Distribution

1. Serialization/Deserialization of data

- Network needs contiguous byte stream
  - Often leads to inefficient "manual pack loops"
  - MPI datatypes can avoid copying[1]

```
// figure out my coordinates (x,y) -- r=x*a+y in the 2d processor array
  int rx = r % px;
  int ry = r / px;
  // figure out my four neighbors
  int north = (ry-1)*px+rx; if(ry-1 < 0)   north = MPI_PROC_NULL;
  int south = (ry+1)*px+rx; if(ry+1 >= py) south = MPI_PROC_NULL;
  int west= ry*px+rx-1;     if(rx-1 < 0)   west = MPI_PROC_NULL;
  int east = ry*px+rx+1;    if(rx+1 >= px) east = MPI_PROC_NULL;
  // decompose the domain
  int bx = n/px; // block size in x
  int by = n/py; // block size in y
  int offx = rx*bx; // offset in x
  int offy = ry*by; // offset in y
   MPI_Datatype north_south_type;
   MPI_Type_contiguous(bx, MPI_DOUBLE, &north_south_type);
   MPI_Type_commit(&north_south_type);
   MPI_Datatype east_west_type;
   MPI_Type_vector(by,1,bx+2,MPI_DOUBLE, &east_west_type);
   MPI_Type_commit(&east_west_type);

   MPI_Request reqs[8];
   MPI_Isend(&aold[ind(1,1)] /* north */, 1, north_south_type, north, 9, comm, &reqs[0]);
   MPI_Isend(&aold[ind(1,by)] /* south */, 1, north_south_type, south, 9, comm, &reqs[1]);
   MPI_Isend(&aold[ind(bx,1)] /* east */, 1, east_west_type, east, 9, comm, &reqs[2]);
   MPI_Isend(&aold[ind(1,1)] /* west */, 1, east_west_type, west, 9, comm, &reqs[3]);
   MPI_Irecv(&aold[ind(1,0)] /* north */, 1, north_south_type, north, 9, comm, &reqs[4]);
   MPI_Irecv(&aold[ind(1,by+1)] /* south */, 1, north_south_type, south, 9, comm, &reqs[5]);
   MPI_Irecv(&aold[ind(bx+1,1)] /* west */, 1, east_west_type, east, 9, comm, &reqs[6]);
   MPI_Irecv(&aold[ind(0,1)] /* east */, 1, east_west_type, west, 9, comm, &reqs[7]);
   MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);

   heat = 0.0;
   for(int i=1; i<bx+1; ++i) {
    for(int j=1; j<by+1; ++j) {
      anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
      heat += anew[ind(i,j)];
    }
   }
```

24% less SLOC

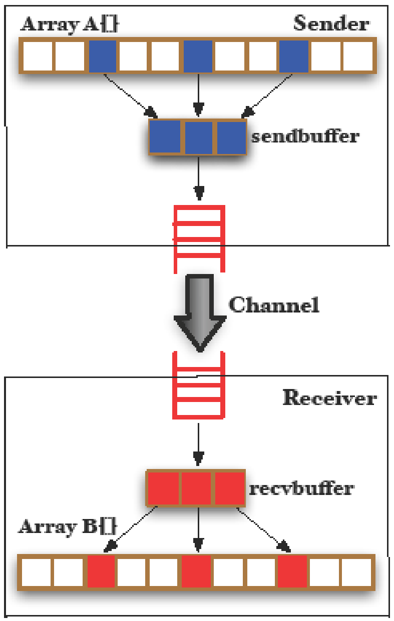
26% decomposition

15% declaration

26% communication

4.67 GF → 5.05 GF

(32 cores, IB)



[1]: Hoefler, Gottlieb: "Parallel Zero-Copy Algorithms […] using MPI Datatypes" (EuroMPI 2010)
Kjolstad, Hoefler, Snir: "[…] Convert Packing Code to Compact Datatypes […]" (PPoPP 2012)

2. Optimizing communication patterns

- Architecture/Network-specific optimization
  - Post send/recv manually in right order
  - Or use neighborhood collectives [MPI-3.0][1]

```
MPI_Datatype north_south_type;
MPI_Type_contiguous(bx, MPI_DOUBLE, &north_south_type);
MPI_Type_commit(&north_south_type);
MPI_Datatype east_west_type;
MPI_Type_vector(by,1,bx+2,MPI_DOUBLE, &east_west_type);
MPI_Type_commit(&east_west_type);

MPI_Comm comm_new;
MPI_Cart_create(comm, 2, {bx, by}, {0,0}, 1, &comm_new);
MPI_Neighbor_alltoallw(sbuf, {1,1,1,1}, {0,0,0,0}, {north_south_type, east_west_type, north_south_type, east_west_type},
          rbuf, {1,1,1,1}, {0,0,0,0}, {north_south_type, east_west_type, north_south_type, east_west_type}, comm_new);

heat = 0.0;
for(int i=1; i<bx+1; ++i) {
  for(int j=1; j<by+1; ++j) {
    anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
    heat += anew[ind(i,j)];
  }
```

64% less SLOC

33% declaration

11% communication

5.05 GF → 5.11 GF

  - Declare the communication topology like a type
    - Optimizations possible in the library

[1]: Hoefler, Traeff: “Sparse Collective Operations for MPI” (HIPS 2009)

3. Communication/computation overlap

- Utilize the machine efficiently (no wait cycles)
  - Often done manually for simple problems
  - Nonblocking collectives provide huge benefit[1]

```
MPI_Datatype north_south_type;
MPI_Type_contiguous(bx, MPI_DOUBLE, &north_south_type);
MPI_Type_commit(&north_south_type);
MPI_Datatype east_west_type;
MPI_Type_vector(by,1,bx+2,MPI_DOUBLE, &east_west_type);
MPI_Type_commit(&east_west_type);

MPI_Comm comm_new;
MPI_Cart_create(comm, 2, {bx, by}, {0,0}, 1, &comm_new);
MPI_Ineighbor_alltoallw(sbuf, {1,1,1,1}, {0,0,0,0}, {north_south_type, east_west_type, north_south_type, east_west_type},
          rbuf, {1,1,1,1}, {0,0,0,0}, {north_south_type, east_west_type, north_south_type, east_west_type}, comm_new, &req);

heat = 0.0;
for(int i=2; i<bx; ++i)
  for(int j=2; j<by; ++j) {
    anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
    heat += anew[ind(i,j)];
  }

MPI_Wait(&req, MPI_STATUS_IGNORE);

for(int i=2; i<bx; ++i)
  for(int j=1; j < by+1; j+=by-1) {
    anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] + aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
    heat += anew[ind(i,j)];
  }
```

30% more SLOC

31% declaration

12% communication

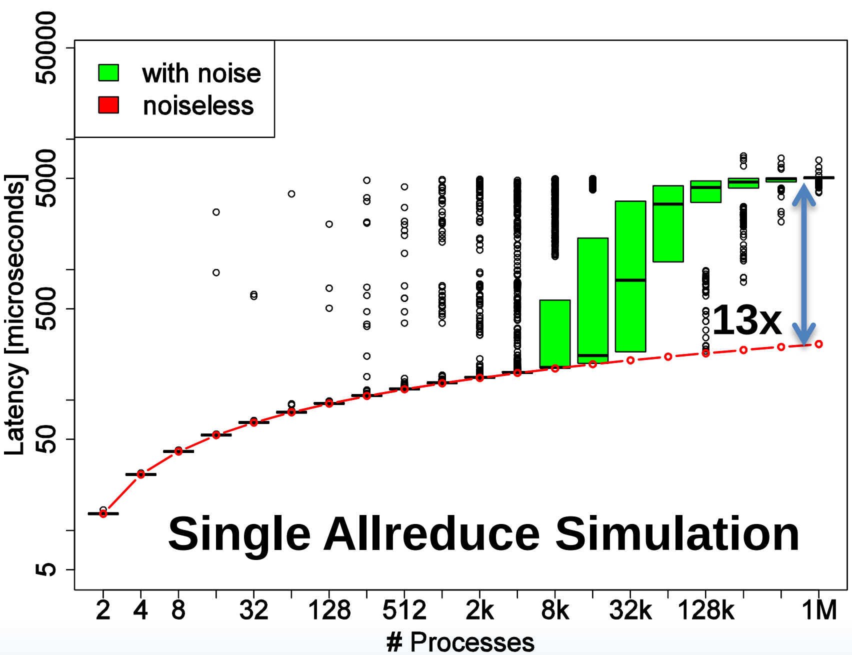
5.11 GF → 5.19 GF

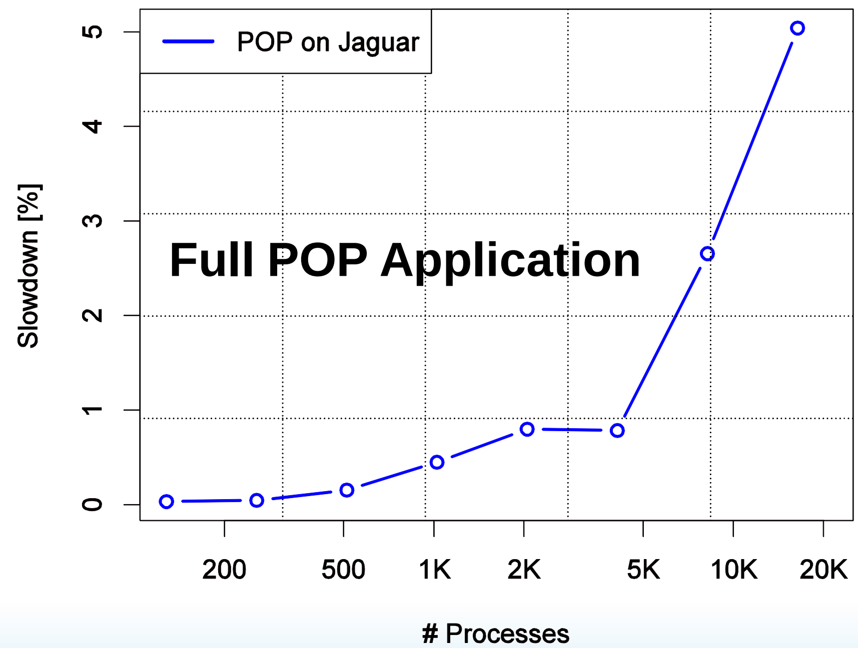
  - Break computation into pieces and arrange with communication (software pipelining)[1]

[1]: Hoefler et al.: "Leveraging Non-blocking Collective Communication […]" (SPAA 2008)

4. Synchronization and System Noise

- Small nondeterministic delays in processes can lead to **huge** delays in parallel applications
  - Depends on the shape and distribution of delays[1]

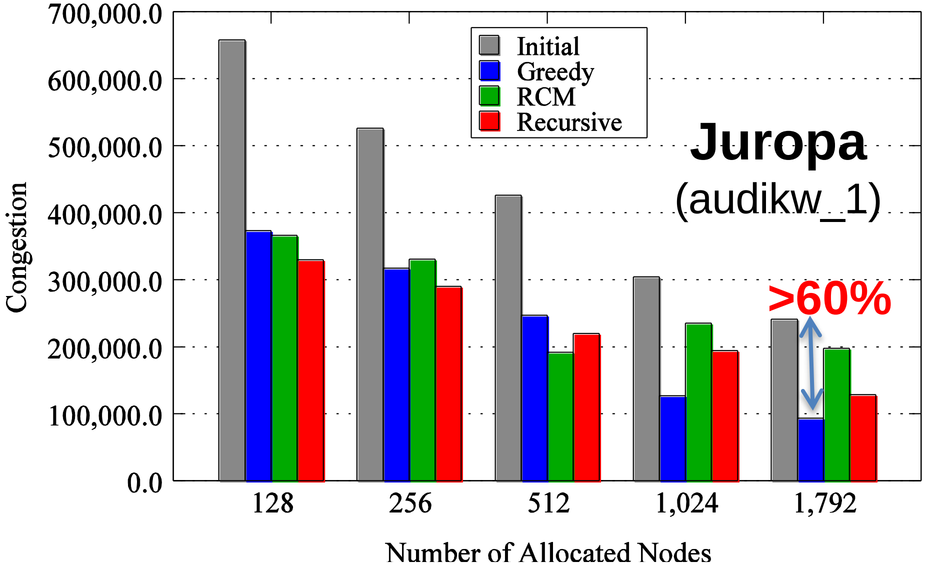


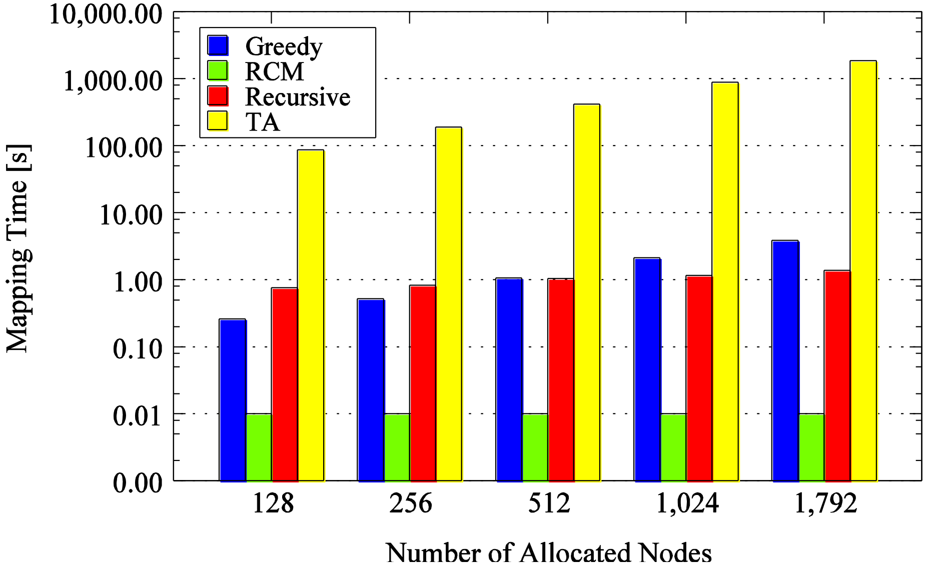


[1]: Hoefler et al.: "Characterizing the Influence of System Noise […]" (SC'10 Best Paper)

5. Topology mapping

- Low-degree physical networks and low-degree application communication networks
  - Good mappings increase performance (NP hard[1])





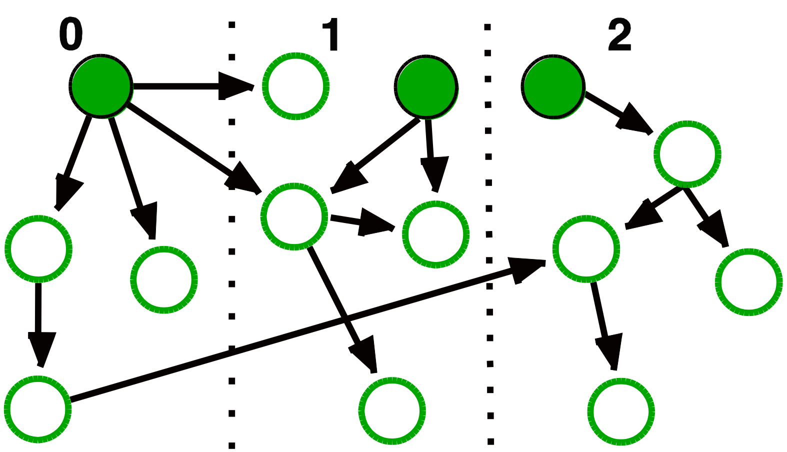
>50% reduction in average dilation!

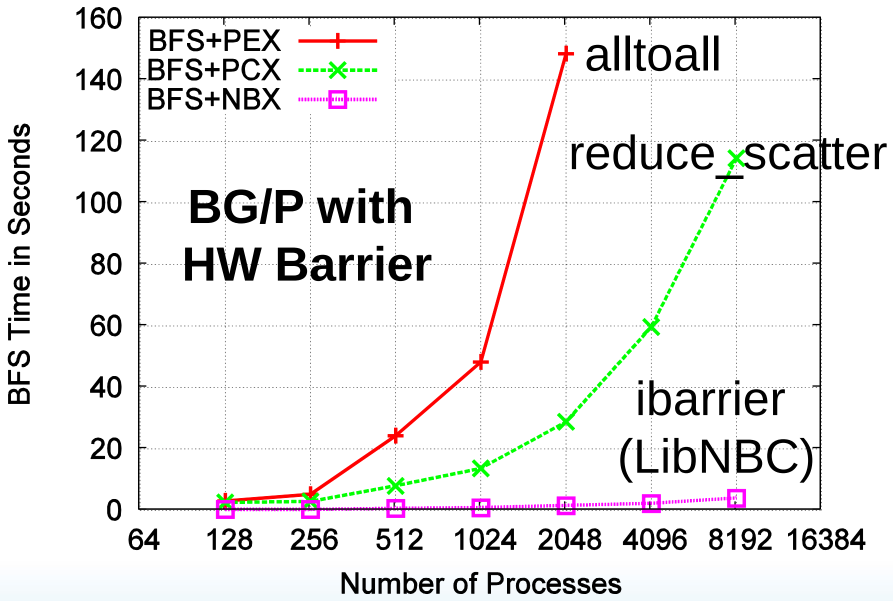
asymptotically faster!

[1]: Hoefler, Snir: "Generic Topology Mapping Strategies […]" (ICS 2011)

6. Scalable Algorithms and Load Balance

- Termination detection (TD) is an important problem
- DSDE[1] is a special case of 1-level TD:
  - Sparse exchange, knowledge only at sender

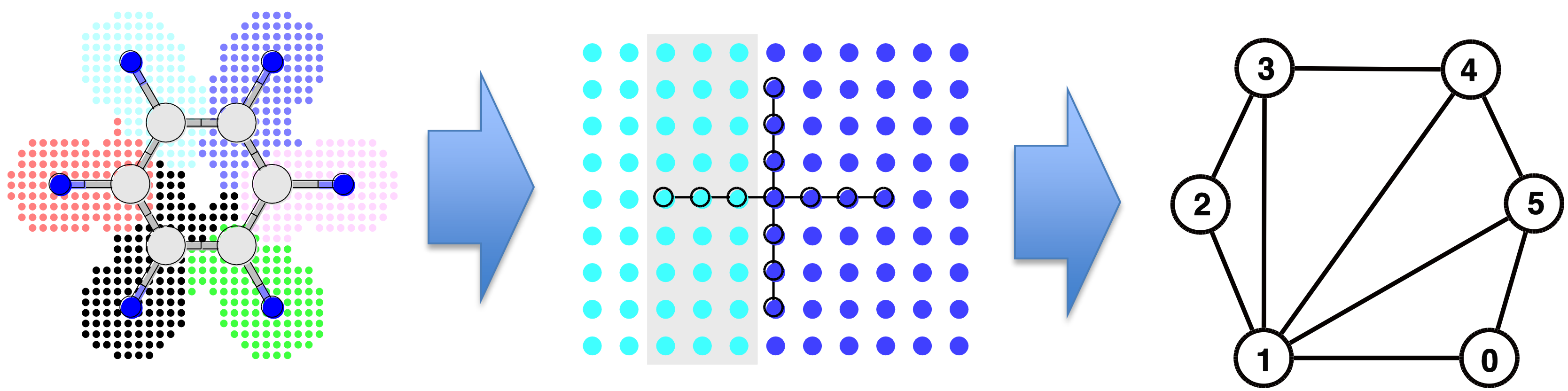




[1]: Hoefler et al.: "Scalable Communication […] for Dynamic Sparse Data Exchange" (PPoPP 2010)

7. Domain Decomposition/Data Distribution

- Most important but also hard (NP hard)
  - Good heuristics exist (METIS, SCOTCH, Chaco, …)
  - Use in conjunction with topology mapping (MPI-2.2)



MPI 2.2

Hoefler et al.: "The Scalable Process Topology Interface of MPI 2.2" (CCPE 2010)

The Optimization Space is Large

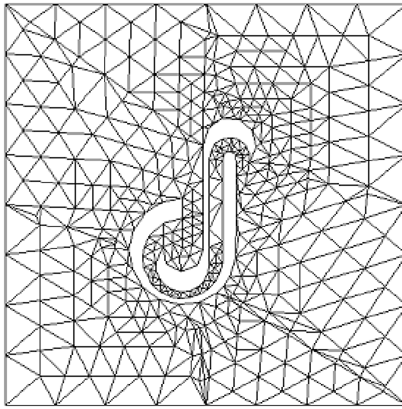
- All criteria are (performance) composable in layers!
- The **right** abstractions and declarations are key
  - Enable CS systems people (like me) to optimize malleable applications
  - Specify as much as possible statically
- Automatic composition: DSLs
  - PBGL / AP (C++ templates)
  - PMTL (C++ templates)
  - … many more!

Work at the System or Implementation Level

- Implement abstractions!
  - SerDes with Datatypes
    - Datatype "JIT" Compiler *[in progress]*
  - Topology mapping
    - Mapping heuristics (RCM) *[ICS'11]*
  - Optimized (asynchronous) Nonblocking Collectives
    - LibNBC *[SC'07],* kernel-level progression *[EuroPar'11]*
  - Optimized neighborhood collectives
    - Group Operation Assembly Language *[ICPP'09]*
  - Optimized Routing in HPC networks *[IPDPS'11]*

Work at the Interface to static Applications

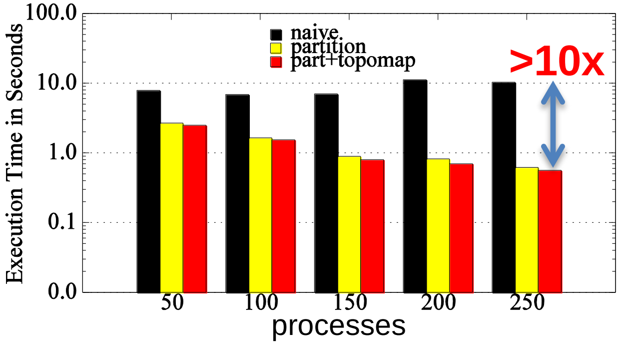
- Matrix Template Library - Linear Algebra
  - Automatic partitioning, load balancing, topology mapping, serial optimizations, neighborhood collectives



Parallel LU

```
for (std::size_t k= 0; k < num rows(LU)−1; k++) {
  if(abs(LU[k][k]) <= eps) throw matrix singular();
  irange r(k+1, imax); // Interval [k+1, n−1]
  LU[r][k] /= LU[k][k];
  LU[r][r] −= LU[r][k] * LU[k][r];
}
```

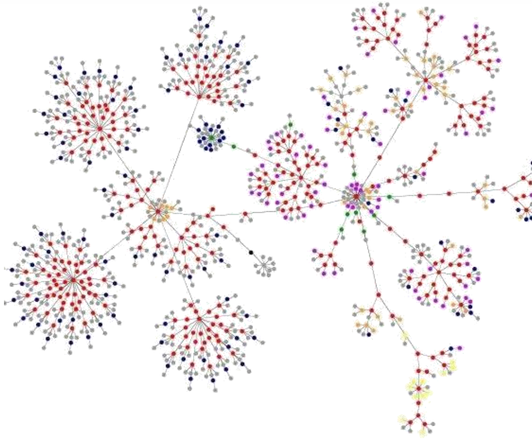
Single MatVec (Idoor) Partitioning/Topology Mapping



Gottschling, Hoefler: “Productive Parallel Linear Algebra Programming […] “ (CCGrid 2012)

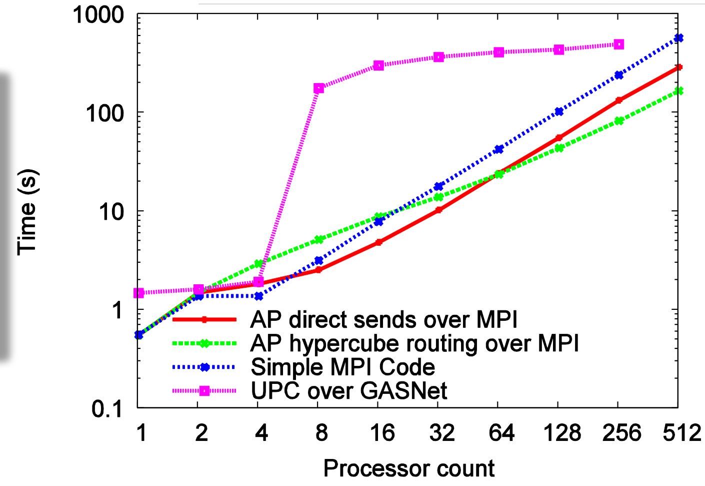
Work at the Interface to dynamic Applications

- Active Pebbles - Graph Programming
  - Automatic overlap, coalescing, active routing, termination detection, vectorized handlers



Parallel RandomAccess

```
struct update_handler {
  bool operator()(uint64_t ran) const
  { table[ran % (N/P)] ^= ran; } // update to table
};
```



Willcock, Hoefler et al.: “Active Pebbles: Parallel Programming for Data-Driven Applications “ (ICS’11)

Summary & Conclusions

- Is parallel programming as “easy” as serial programming? – No(t yet)
  - We made some progress (still far to go)
- Communication and network become issue → network-centric programming
- New languages can and will help
  - But they shall never ignore the past (MPI)
- Domain-specific languages can isolate problems
  - How does a good general parallel language look like?

Collaborators, Acknowledgments & Support

- Marc Snir, Bill Gropp, Bill Kramer
- Andrew Lumsdaine, J. Willcock, N. Edmonds
- Peter Gottschling
- Students: T. Schneider, J. Domke, C. Siebert
- Sponsored by: