A Hybrid SPMD – Coarse Grain Dataflow Parallel Programming Model

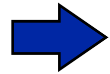
Adrian Soviani

Advisor: Prof. Jaswinder Pal Singh

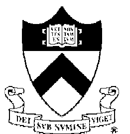
Computer Science Department
Princeton University

May 17th, 2010

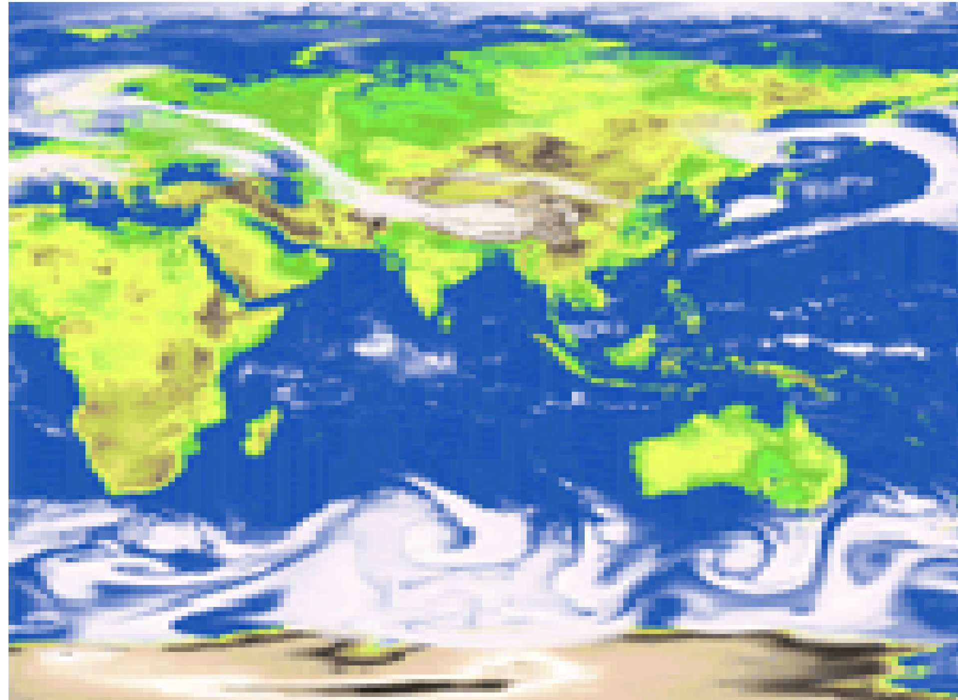
Outline



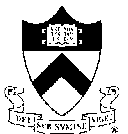
- Motivation
- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
- Cost Model



MOM4: Large Scale Application Performance

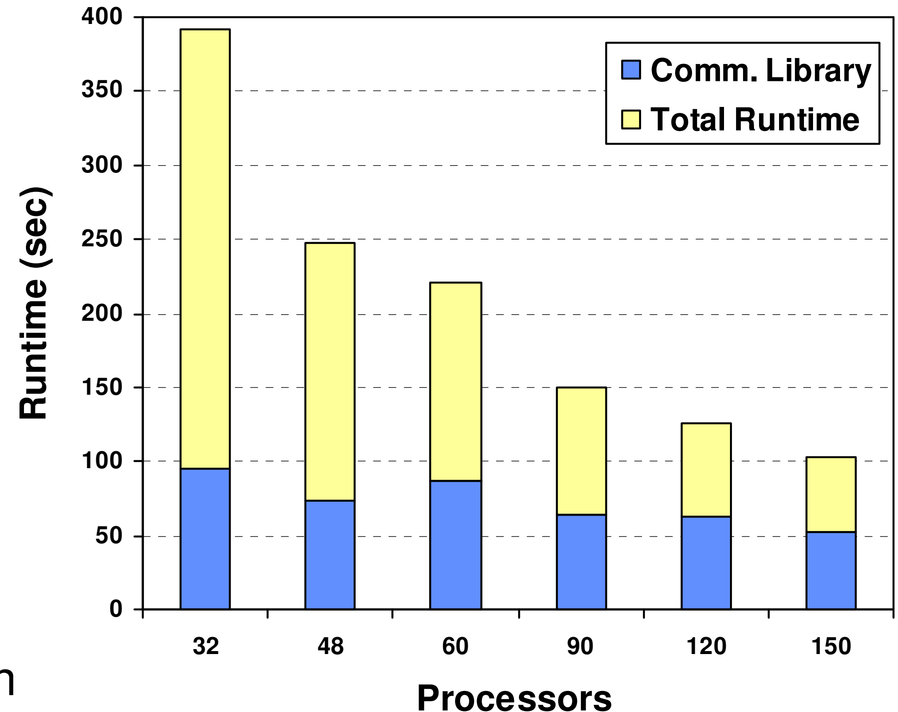


- Geophysical Fluid Dynamics Lab (GFDL), FMS Modular Ocean Model
  - Ocean, Atmosphere, Ice grids
- Initial development effort
  - 150,000+ lines Fortran 90
  - Team: 10 physicists (PDE solvers), 2 computer scientists (library)

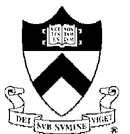


MOM4: Large Scale Application Performance

- Communication library
  - 50% of 150 CPU runtime
- Address scalability issues
  - Identify bottlenecks
  - Cost transparency
- Algorithmic optimizations
  - Global field reduction
  - Data layout 2D/3D decomposition
    - Optimized version, 150 CPU
      - runtime: 85.6 vs. 102.6 (+17%)
      - comm. lib: 34.2 vs. 51.7 (+34%)
      - speedup: 146.5 vs. 122.2

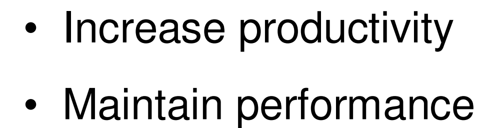


The process of developing and optimizing large scale applications takes a significant effort. A good programming model can increase productivity.



Programming Model Wish List

- Good programmability
- Efficiency, Portability
- Cost transparency
- Low effort optimization



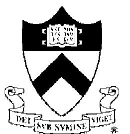
Low Level Libraries

- SHMEM, RDMA, MPI, pthreads
- High programming effort
- High performance
- Optimization portability ?

High Level Languages

- Cilk, OpenMP, Lucid, Chapel
- Problem coverage ?
- Performance ?
- Portability ?

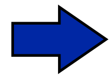
Hard to achieve tradeoff without sacrificing features



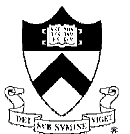
5 min

Outline

- Motivation



- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
- Cost Model

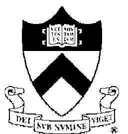


Thesis Contribution

- Introduces hybrid SPMD - Coarse Grain Dataflow Progr. Model (CGD)
  - Data and task parallelism described as dependencies between computations, data distributions
  - Communication, synchronization – added automatically
- Presents programming language for CGD, implementation of
  - compiler, distributed datastructure library for MPI, SHMEM, pthreads
  - benchmarks including NPB FT, Barnes-Hut, PDE solvers
- Shows CGD increases productivity of programming, optimization compared to message passing
  - Similar or better performance, smaller problem coverage
  - Automatic : comm, sync, data management, scheduling optimization; several high level optimizations easily available
  - Performance : CGD FT vs. NPB MPI +27%, CGD heat dissipation vs. MPI +41%, CGD Barnes-Hut vs. pthreads +15% (Altix, 128 CPU)
- CGD has better performance portability compared to CC-SAS

Thesis Contribution, related

- Introduces the αDBSP hierarchical bandwidth machine model
  - Used to estimate application runtime
  - Naturally extends the Decomposable BSP (DBSP)
  - Adds bandwidth growth factor α to each message exchange
    - h-relation becomes (h, α)-relation
- Shows αDBSP is an improvement of DBSP
  - Tighter bounds, globally unbalanced problems : nearest neighbor exchange on pruned butterfly
    - αDBSP : O($\log^3$), DBSP : O(sqrt p)
  - Simpler Analysis : one-to-all broadcast, optimal cost
    - αDBSP : 1 superstep, DBSP : O(log(p)) supersteps
- Shows αDBSP gives lower bound on hierarchical application bandwidth
  - Link capacity planning, avoid inherent bandwidth bottlenecks

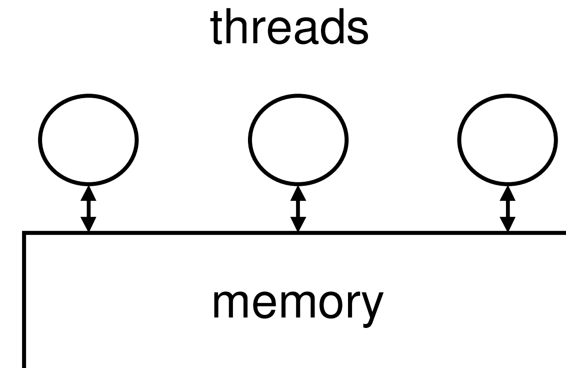


Outline

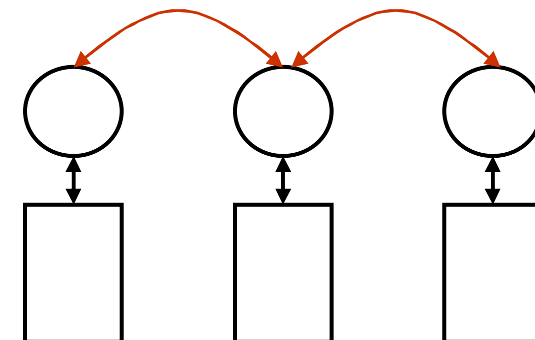
- Motivation
- Thesis Contribution
- ➡ Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
- Cost Model

Programming Models Overview

- Shared address space model (CC-SAS)
  - Concurrent threads
  - Shared address space
  - Synchronization
  - SMP, ccNuma
    - remote memory access transparent
  - pthreads, OpenMP, Cilk, Java



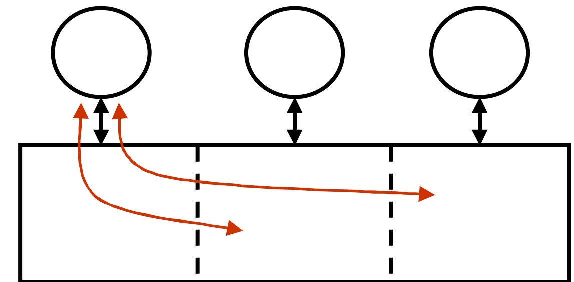
- Message passing (clusters)
  - Concurrent processes
  - Local memories
  - Communication explicit
    - Two sided : MPI send, recv
    - One sided : SHMEM put, get



Issues: CC-SAS - performance portability, message passing - high programming effort

Programming Models Overview

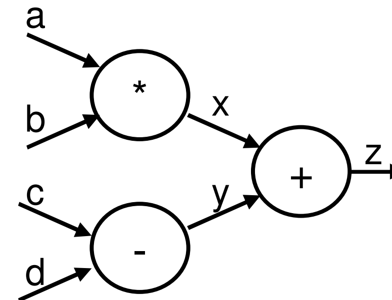
- Partitioned Global Address Space – PGAS (GASNet)
  - Concurrent threads
  - Partitioned address space
  - Array distributed across memory
    - User defines distribution (restrictions)
    - Compiler detects remote accesses
  - UPC – C dialect, SPMD
    - Row, col array distribution, global pointers
  - CoArray – Fortran dialect, SPMD
    - Adds pid index to distributed arrays
  - X10 – Java dialect, SPMD, IBM
  - Chapel – Java/Cilk, recursive tasks, SPMD, Cray
    - Flexible datastructure distributions



Issues: Fine grain message overhead, read latency on clusters

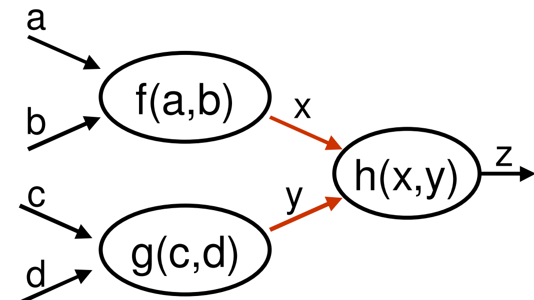
Programming Models Overview

- Dataflow Model
  - Nodes wait for inputs, produce output
    - Token overhead
  - Functional languages : Lucid, Id
  - Single assignment rule
  - Special loop syntax
  - Statement order irrelevant
    - Dependencies determine schedule



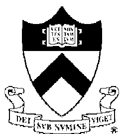

```
X := a * b;
Z := x + y;
Y := c – d;
```

- Threaded Dataflow
  - Nodes compiled into threads : GLU
- Large Grain Dataflow
  - Nodes / macro-actors are seq. functions
  - Coordination languages : FBP, Linda

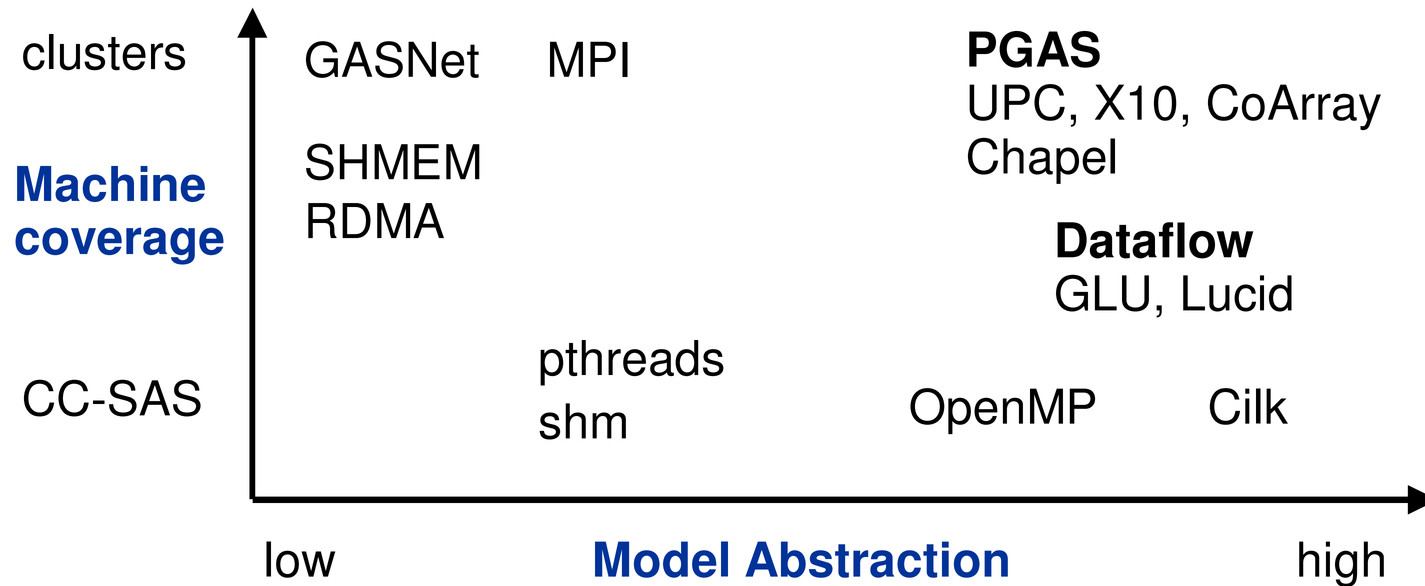



```
X := f (a,b);
Z := h (x,y);
Y := g (c,d);
```

Issues: granularity, scheduler overheads

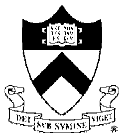


Programming Model Landscape



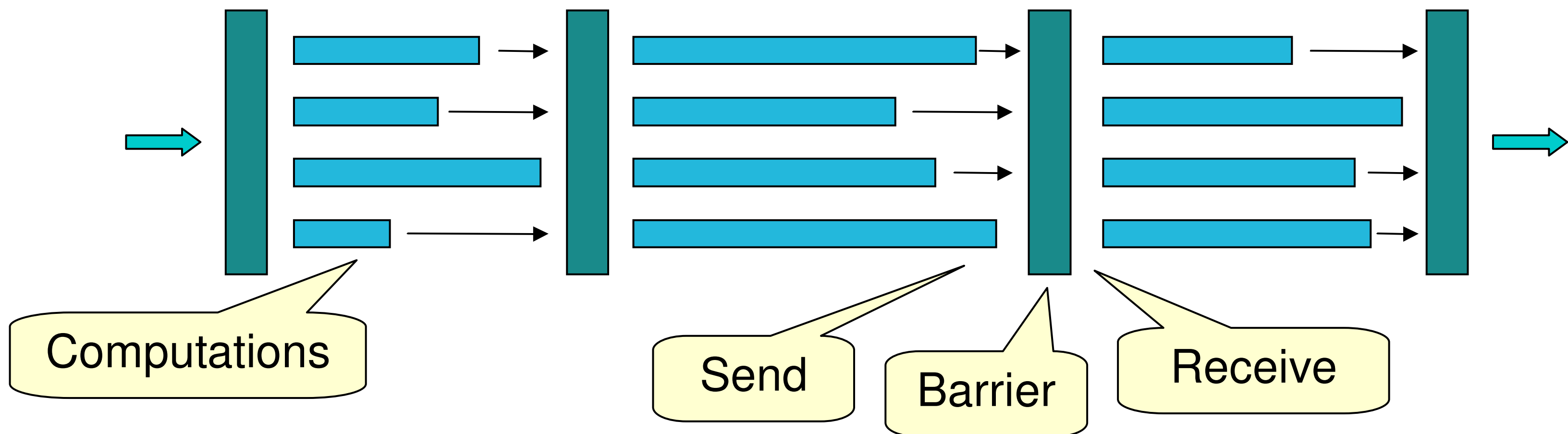
	Issues	*Partial solutions*
Dataflow	Task granularity, scheduling	Coarser grain
OpenMP, Cilk	Portability, data locality	Task creation
PGAS	Fine grain messages, latency	Explicit local to global copy

No definite winner - different solutions work best for each problem class and arch.



5 min

Cost Estimation, BSP Machine Models



- Decomposable BSP : synchronized steps
  - scientific computing: computation steps, followed by data redistribution
  - h-relation: each proc. sends and receives at most h packets
- Cost estimation : T = w + h g(i) + l(i)
  - g ~ bandwidth, l ~ latency : generic machine parameters
  - Globally unbalanced patterns an issue
    - One element halo exchange - O(sqrt p) vs. O(1) on 2D mesh
- Other models: PRAM, LogP, EBSP, HBSP

Outline

- Motivation
- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
- Cost Model

Coarse Grain Dataflow Model (CGD)

- Hybrid Model
  - Short answer: dataflow of SPMD computations
- ~ Dataflow, ! message passing
  - Dependency graph between data and computations
  - Single assignment rule
  - Order irrelevant, dependencies determine schedule
- ~ SPMD, ! dataflow
  - Explicit decomposition, assignment of data domains
  - Explicit assignment of computation
- CGD scheduler : ~ dataflow, ! message passing
  - Computations ordered topologically
  - Inserts communication, synchronization
  - Optimizations

CGD Matrix Multiplication Example

X = A + B

D = X * C

Dataflow Graph

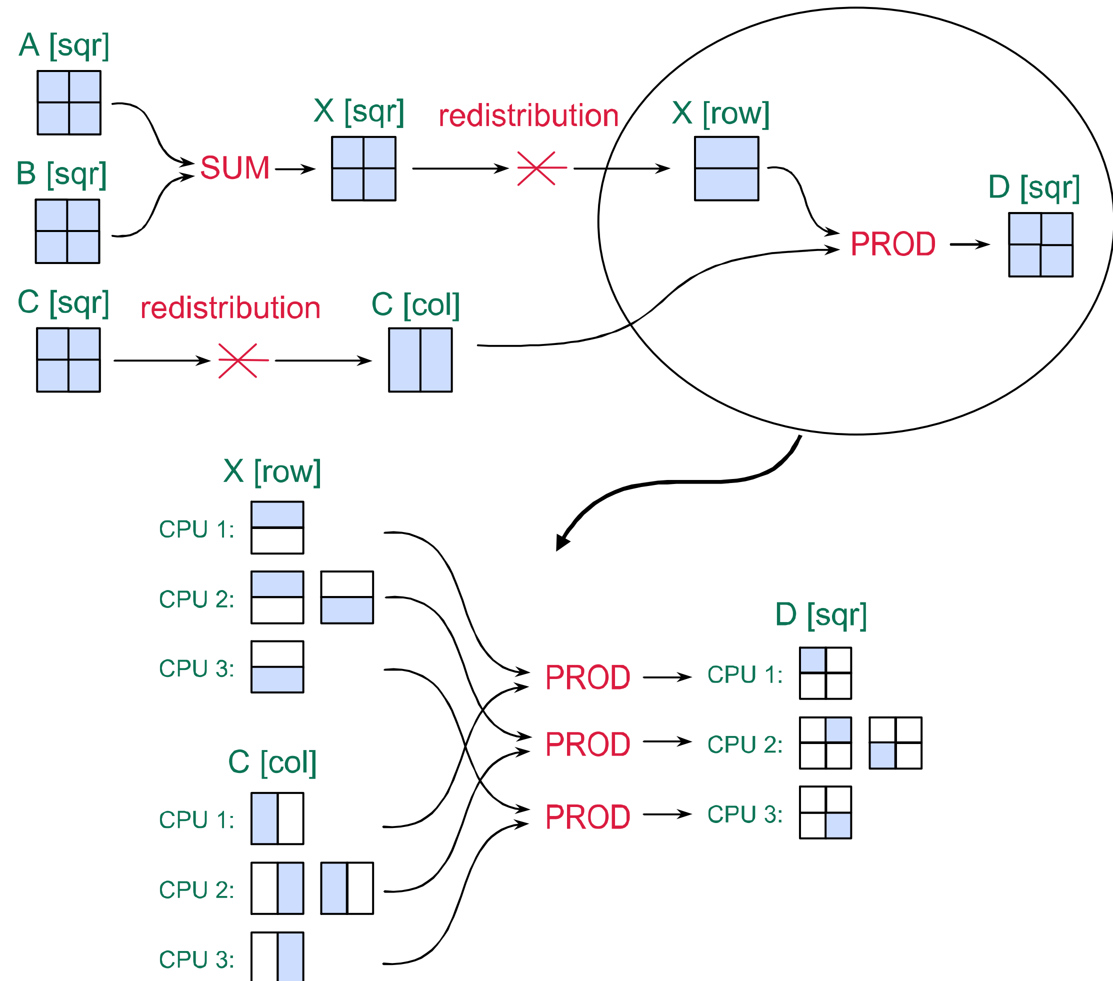
Nodes :

- (data, distribution) pairs
- parallel computations

Links : dependencies

A[sqr] : 2D matrix partition

C[sqr], C[col]



Parallel Computation

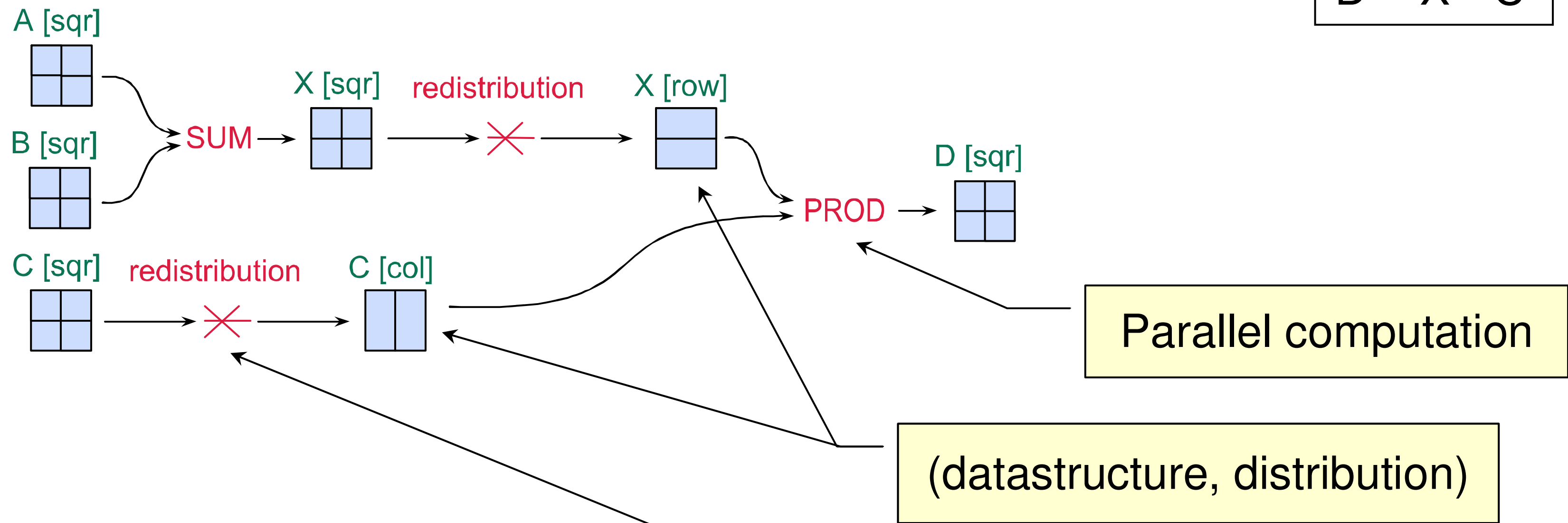
- data parallel computation
  - sequential kernel
- overlapping domains

Hybrid Model: Data → Data distribution, Computation → SPMD

Matrix Multiplication: Dataflow Graph

X = A + B

D = X * C

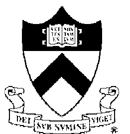


CGD Dataflow code

```
SUM  ( A[sqr], B[sqr] → X[sqr] );

PROD ( X[row], C[col] → D[sqr] );
```

Developer writes

- CGD Dataflow, Rules
- C++ Kernels, Distributions : SUM, PROD, sqr, etc.



Matrix Multiplication: Distribution Rules

- Describe how CGD can transform datastructures
  - Domain inclusion $A < B < C$; (A, B, C are distributions)
  - Domain union $A = B + C$;
  - Global redistribution $M : A \rightarrow B$;
    - M matrix of domains
    - M_{ij} = domain from P_i needed by P_j

```
part sqr < col;

part sqr < row;

part sqr2row : sqr → row;

part sqr2col : sqr → col;
```

Predefined function computes redistribution matrix

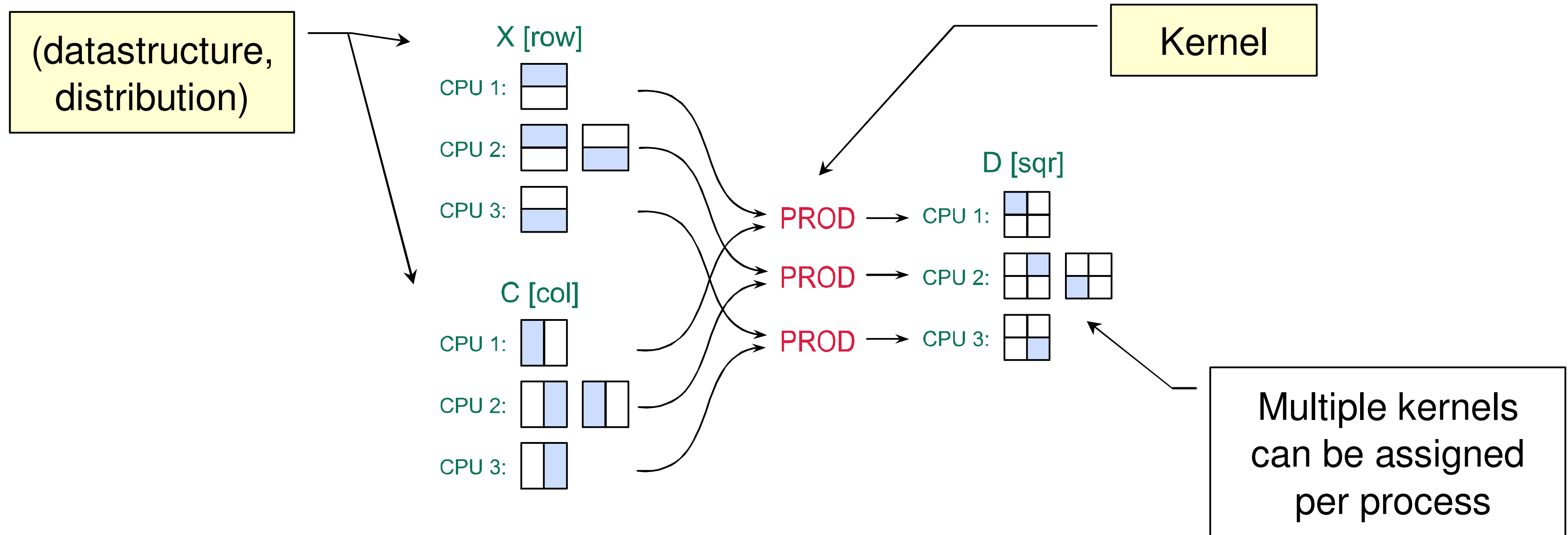
```
computeSwap (sqr, row → sqr2row [PPEn]);

computeSwap (sqr, col → sqr2col [PPEn]);
```

Distribution rules are defined once and automatically used for all datastructures throughout the relevant program section

15 min

Matrix Multiplication: Parallel Computation

```
PROD ( X[row], C[col] → D[sqr] );
```

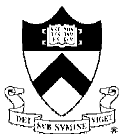


- Parallel computation assigns
  - (kernel, in/out data domains) → process

Matrix Multiplication: Kernels

- Kernel is a sequential computation
  - Acts on local datastructure domains, uses global indexes
  - Implemented as C++ function
  - Datastructure, domain types
    - user defined, predefined by library (more later)

```
void prod ( Vector2Dreal &A, Vector2Dreal &B,            // in
            Range2D &sqr, Range2D &row, Range2D &col,    // in
            Vector2Dreal &C )                            // out
{
  for (int i=sqr.s1; i<=sqr.e1; i++)
    for (int j=sqr.s2; j<=sqr.e2; j++) {
      double s = 0.0;
      for (int k=col.s1; k<=col.e1; k++)
         s += A(i,k) * B(k,j);
      C(i,j) = s;
    }
}
```



Matrix Multiplication: Dataflow, Kernels

```
// Kernel Declaration

function add (A, B, ra -> C)
  Vector2Dreal A[ra], B[ra], C[ra],
  Range2D ra;

function prod (A, B, sqr, row, col -> C)
  Vector2Dreal A[row], B[col], C[sqr],
  Range2D row, col, sqr;

function init (cfg, ra -> A)
  Vector2Dreal A[ra], Range2D ra,
  Config cfg;

// Dataflow

function foo (A, B, C -> D)
  Vector2Dreal A[sqr], B[sqr], C[sqr], D[sqr]
{
  add   ( A[sqr], B[sqr] -> X[sqr] );
  prod  ( C[row], X[col], sqr -> D[sqr] );
}

function bar () [gen]
{
  init (cfg, sqr -> A[sqr]);
  init (cfg, sqr -> B[sqr]);
  init (cfg, sqr -> C[sqr]);

  foo   (A[sqr], B[sqr], C[sqr] -> D[sqr]);

  print (fout, D[sqr]);
}
```

```
// Kernel Implementation

void add (Vector2Dreal &A, Vector2Dreal &B, Range2D &ra,
          Vector2Dreal &C)
{
  for (int i=ra.s1; i<=ra.e1; i++)
    for (int j=ra.s2; j<=ra.e2; j++)
      C(i,j) = A(i,j) + B(i,j);
}

void prod (Vector2Dreal &A, Vector2Dreal &B,
           Range2D &sqr, Range2D &row, Range2D &col,
           Vector2Dreal &C)
{
  for (int i=sqr.s1; i<=sqr.e1; i++)
    for (int j=sqr.s2; j<=sqr.e2; j++) {
      double s = 0.0;
      for (int k=col.s1; k<=col.e1; k++)
              s += A(i,k) * B(k,j);
      C(i,j) = s;
    }
}

void init (Config &cfg, Range2D &ra,
           Vector2Dreal &A)
{
  for (int i=ra.s1; i<=ra.e1; i++)
    for (int j=ra.s2; j<=ra.e2; j++)
      A (i,j) = (random()*i+101*j) % 311;
}
```

- Application developer / scientist writes dataflow, kernels

15 min

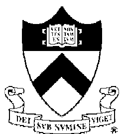
Matrix Multiplication: Types, Rules, Distributions

```
// Types

type range Range2D;
type partition <Range2D> PartRange2D [PartNP];
type swap <Range2D> SwapRange2D [PartNP];

type data Int, Config, File;
type data Vector2Dreal [PartRange2D];

// Constants

const PartRange2D row[ALLn], col[ALLn], sqr[ALLn];
const SwapRange2D sqr2row[PPEn], sqr2col[PPEn];
const Config cfg[ALL1], File fout[ALL1];

// Decomposition Rules

part sqr < row;
part sqr < col;
part sqr2row : sqr -> row;
part sqr2col : sqr -> col;
```

```
// Distribution Definition

void env_init (Config cfg, Env& env)
{
  Alias (int, pe, env.pre.pe); Alias (RangeNP, ppen, env.pre.PPEn[pe]);
  Alias (RangeNP, alln, env.pre.ALLn[pe]);
  Alloc (env.sqr, alln); Alloc (env.col, alln); Alloc (env.row, alln);
  Alloc (env.sqr2col, ppen); Alloc (env.sqr2row, ppen);
  env.cfg = cfg; env.fout = stdout;

  makePart2D (env.sqr, cfg.size, cfg.nopy, cfg.size, cfg.nopx);
  for (int p=0; p<env.sqr.getNP(); p++) {
    env.row[p] = env.sqr[p]; env.row[p].s2=0; env.row[p].e2=cfg.size-1;
    env.col[p] = env.sqr[p]; env.col[p].s1=0; env.col[p].e1=cfg.size-1;
  }
  setNoPart2D (env.row); setNoPart2D (env.col);
  computeSwap (env.sqr, env.col, env.sqr2col);
  computeSwap (env.sqr, env.row, env.sqr2row);
}
```

▮ is parallel overhead code

- Library developer writes rules, distributions once
  - Used by entire application
- Common distributions, rules for nD arrays - can be incl. in runtime



15 min

Outline

- Motivation
- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
- Cost Model

Optimization : Programming Model Support

OpenMP | PGAS | pthreads | CGD | MPI | SHMEM | PGAS | CGD

User

Programming Model

Algo. spec. opt.

HW spec. opt.

Algorithm

Decomposition

Assignment

Dependencies

Schedule/Sync

Encoding

Messaging

HW Primitives

Compiler

Comp, rt

C,rt

CC-SAS:
SMP, ccNuma

Lib

Lib

Compiler, GASNet

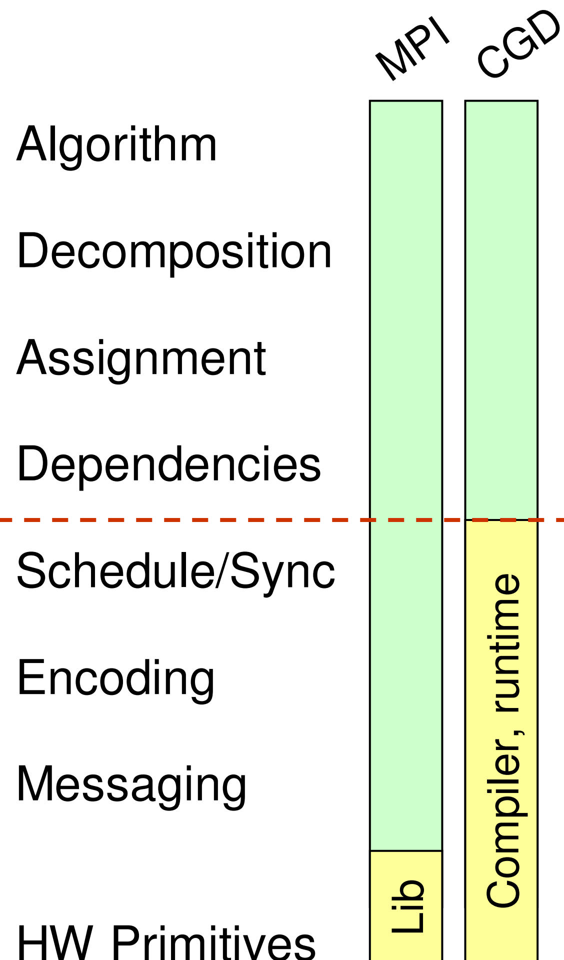
Compiler, runtime

UDP

RDMA: Infiniband
Myrinet, Quadrics

Which optimizations should be provided by the PM ?

CGD Application design and optimization



- Algo: design space exploration
  - Algorithm tradeoffs
    - Comm. vs. computation, result accuracy
  - Data layout: decomposition, assignment
    - Decrease comm. bandwidth, latency
- PM: compiler optimizations
  - Scheduler: comm. overlap, reorder, aggregation
  - Data domain reuse (more later)
- PM: runtime optimizations
  - Aggregates multiple stages
    - Data encoding, buffer management, message scheduling
  - Avoids unnecessary layers, overheads
    - MPI vs. SHMEM / CC-SAS

Optimization effort: MPI - all of the above, CGD - only algo. specific, made easier

NPB FT: "Slabs" Communication Overlapping

Dataflow

A [row] → FFT_ROW → B [row] → redistribution → B [mcol] → FFT_COL → C [mcol]

Execution Timeline

A [row] — FFT_ROW — B [row] — redistribution — B [mcol][0] — FFT_COL — C [mcol][0]; redistribution — B [mcol][1] — FFT_COL — C [mcol][1]

2nd "slab"

Time

- Typical bottleneck: 3D transpose, communication bandwidth
- Optimization: split domain into smaller "slabs", compute FFT_COL on each
  - First communication step shorter than full transpose
  - Subsequent comm. steps are overlapped with computation (auto)

NPB FT: Adding Communication Overlapping

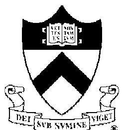
FT in CGD

```
part sw01 : dom0 -> dom1;
part sw12 : dom1 -> dom2;
part sw21 : dom2 -> dom1;
part sw10 : dom1 -> dom0;
part sw02 : dom0 -> dom2;
part sw20 : dom2 -> dom0;

function ifft (A -> D)
  Vector3Dcplx A[dom2], D[dom0]
{
  cffts3 (sp, finv, A[dom2] -> B[dom2]);

  if (lay1d, B[dom2] -> C[dom0]) {
    // 1D
    cffts2 (sp, finv, B[dom0] -> C[dom0]);
  } else {
    // 2D
    cffts2 (sp, finv, B[dom1] -> C[dom1]);
  }
  cffts1 (sp, finv, C[dom0] -> D[dom0]);
}
```

FT "slabs" in CGD

```
part sw01 : dom0 -> mdom1;
part sw12 : dom1 -> mdom2;
part sw21 : dom2 -> mdom1;
part sw10 : dom1 -> mdom0;
part sw02 : dom0 -> mdom2;
part sw20 : dom2 -> mdom0;

part dom0 = mdom0; part dom1 = mdom1; part dom2 = mdom2;

function ifft (A -> D)
  Vector3Dcplx A[dom2], D[dom0]
{
  cffts3 <ALL1>  (sp, finv, A[dom2] -> B[dom2]);

  if (lay1d, B[dom2] -> D[dom0]) {
    // 1D
    cffts2 <mdom0> (sp, finv, B[mdom0] -> C[mdom0]);
    cffts1 <ALL1>  (sp, finv, C[dom0]  -> D[dom0]);
  } else {
    // 2D
    cffts2 <mdom1> (sp, finv, B[mdom1] -> C[mdom1]);
    cffts1 <mdom0> (sp, finv, C[mdom0] -> D[mdom0]);
  }
}
```

- Changes vs. NPB FT
  - 17 lines CGD : types, distribution rules, (data, distribution) pairs
  - 32 lines C++ : initialize distributions, redistributions
  - << MPI : reallocate arrays, buffers, redo messaging, reorder code
    - Above steps done auto in CGD



Programming Effort and Scalability Comparison

128 CPU, FT C

Version	Layout	Language	Line Count				Speedup
Serial	0D	Fortran				704	1
OpenMP	1D	C				752	17.21
MPI	1D, 2D	Fortran				1261	103.10
			C++	CGD func.	CGD decl.	Total	
CGD	1D, 2D	C++, CGD	703	45	38	786	111.46
CGD	1D, 2D slabs	C++, CGD	740	47	43	830	125.72

- C functions take ~700 lines, mostly unchanged between OpenMP, CGD
  - C code translated by Omni compiler project from Fortran
- NPB MPI adds 500 lines implementing optimized communication
- CGD dataflow describes parallelism - 45 vs. 786 total lines
- CGD “slabs” optimization requires simple changes (prev)

22 min

2D PDE: Communication Aggregation

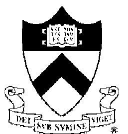
2 halo exchanges

2 steps

1 halo exchange

2 steps

- MOM4 `freesurface_update` main scalability bottleneck
- 2D solver optimization
  - Trade little extra computation for latency / synchronization overhead
  - Optimization: replicate computation, merge two communication steps



18 min

2D PDE: Adding Communication Aggregation

```
part cell < halom < halo;
part boundary < halo;
part halo = boundary + halom;

part sw_chm : cell -> halom;
part sw_ca  : cell -> fullone;

function foo () [gen]
{
  matrix_read (cfg, cell, halo -> A[cell], BD[halo]);

  // main iteration
  loop ( idx, cond ; A [cell] -> C [cell] )
  {
    copy ( BD[boundary] -> A[boundary] );
    copy ( BD[boundary] -> B[boundary] );
    ftcs_step ( cfg, A[halo] -> B[cell] );
    ftcs_step ( cfg, B[halo] -> C[cell] );
    econd (cfg, idx -> cond);
  }

  print (fout, C[fullone]);
  check_sym (C[fullone]);   // check correctness
}
```

```
part cell < halom < halo < halo2;
part halom2 < halo2;
part boundary < halo;
part boundary < boundary2 < halo2;
part halo  = boundary + halom;
part halo2 = boundary2 + halom2;

part sw_chm : cell -> halom2;
part sw_ca  : cell -> fullone;

function foo () [gen]
{
  matrix_read (cfg, cell, halo2 -> A, BD);

  // main iteration
  loop ( idx, cond ; A [cell] -> C [cell] )
  {
    copy ( BD[boundary2] -> A[boundary2] );
    copy ( BD[boundary] -> B[boundary] );
    ftcs_step ( cfg, A[halo2] -> B[halom] );
    ftcs_step ( cfg, B[halo] -> C[cell] );
    econd (cfg, idx -> cond);
  }

  print (fout, C[fullone]);
  check_sym (C[fullone]);   // check correctness
}
```

- Changes
  - 10 lines CGD : distribution rules, (data, distribution) pairs
  - 34 lines C++ : initialize distributions, redistributions
  - rules redefined once, easily applied throughout code (large scale apps)

Outline

- Motivation
- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
- Cost Model

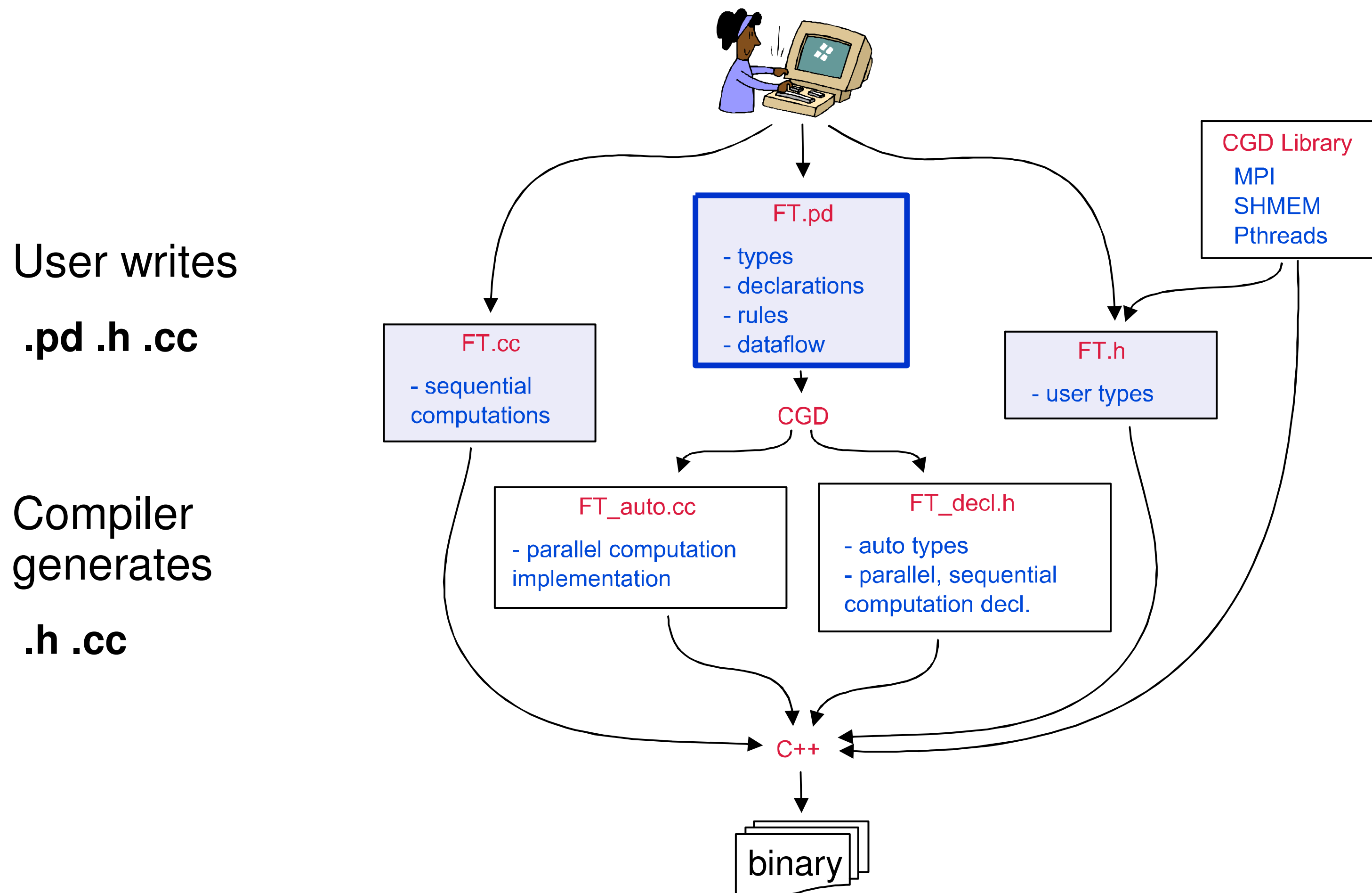
Language Implementation

User writes

.pd .h .cc

Compiler generates

.h .cc



- Dataflow coordination language, not fully featured
  - Short top level dataflow parallel section (paral)
  - Most code written as sequential C++ (scientists)

CGD Type Declaration

Type Declaration	Syntax
Domain (range), arbitrary	**type range** range-ty;
Domain distribution (partition)	
distribution : 0,1 domains assigned per proc multi-distribution : any # domains assigned per proc	**type partition** <range-ty> part-ty [predef-part-ty]; **type mpartition** <range-ty> part-ty [predef-part-ty];
Redistribution matrix (swap)	
transformation between distributions transformation between multi-distributions	**type swap** <range-ty> swap-ty [predef-part-ty]; **type mswap** <range-ty> swap-ty [predef-part-ty];
Datastructure, arbitrary	
non-decomposable decomposable accepting the specified distributions	**type data** data-ty; **type data** data-ty [part-ty, …];

- Domain, datastructure types
  - Declared in CGD, defined by user in C++, operators needed
- Distribution, redistribution types
  - Declared in CGD, automatically generated C++ code

22 min

Type Declaration Example: FT "slabs"

```
// distribution types

type range Range3D;

type partition <Range3D> PartRange3D [PartNP];
type mpartition <Range3D> MPartRange3D [PartNP];

type mswap <Range3D> MSwapRange3D [PartNP];

// datastructure types

type data Setup, Checksums, Int, Bool, Cplx;

type data Vector3Dcplx [PartRange3D, MPartRange3D];
type data <Cplx> ArrayCplx [PartNP];
```

3D domain type (user)

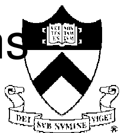
3D domain distribution, multi-distribution

Redistribution matrix for
3D domain multi-distributions

Non-decomposable data types (user)

3D complex vector type allowing
3D domain decomposition (user)

- Library provides predefined
  - Datastructure and domain implementation for nD arrays, lists, …
  - Distributions of n elements PartNP : ALLn, ONEn, PPEn
  - Distributions of one element PartOne : ALL1, ONE1
- All graph nodes are (datastructure, distribution)
  - Non-decomposable datastructures use PartOne distributions
  - Distributions, redistributions are datastructures use PartNP distributions



22 min

CGD Dataflow Definition

	Syntax
Parallel, sequential function declaration argument type, distribution, and optional inout flag (*) dataflow graph for parallel functions	**function** func-name (IN-arg-list → OUT-arg-list) data-ty arg | arg [part-arg] | arg* [part-arg], … ; { dataflow }
Dataflow graph	{ distribution-rule | loop | if | computation ; … ; }
Loop construct idx-arg is iteration count, cond-arg is end condition	**loop** (idx-arg, cond-arg ; IN-arg-list → OUT-arg-list) { dataflow }
If construct cond-arg determines which sub-graph is evaluated	**if** (cond-arg, IN-arg-list → OUT-arg-list) { true-dataflow } **else** { false-dataflow }
Computation invocation, parallel or sequential function	func-name <task-part> (IN-arg-list → OUT-arg-list);

- LOOP – outputs logically copied to inputs after each iteration (~Lucid)
- IF – subgraphs produce same outputs, only one evaluated
- <task-part> – assigns computations to proc. (default ALL1)
- INOUT arguments
  - in and out dataflow nodes, single modified C++ datastructure
  - in node logically copied before execution (avoided)

Dataflow Example: FT "slabs"

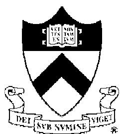
```
function ifft (A -> D)
  Vector3Dcplx A[dom2], D[dom0]
{
  cffts3 <ALL1>  (sp, finv, A[dom2] -> B[dom2]);

  if (lay1d, B[dom2] -> D[dom0]) {
    // 1D
    cffts2 <mdom0> (sp, finv, B[mdom0] -> C[mdom0]);
    cffts1 <ALL1>  (sp, finv, C[dom0]  -> D[dom0]);
  } else {
    // 2D
    cffts2 <mdom1> (sp, finv, B[mdom1] -> C[mdom1]);
    cffts1 <mdom0> (sp, finv, C[mdom0] -> D[mdom0]);
  }
}

function mainloop ( -> success) [gen]
  Bool success[ALL1]
{
  init_checksums (sp -> cksum);
  compute_init_cond (sp -> u0[dom0]);

  fft (u0[dom0] -> u1[dom2]);

  loop (iter, cond; cksum[ONE1] -> cksum2[ONE1])
  {
    evolve (sp, iter, u1[dom2] -> u2[dom2]);

    ifft (u2[dom2] -> u3[dom0]);

    checksum (iter, u3[dom0], cksum -> cksum2);
    econd (niter, iter -> cond);
  }

  verify <ONE1> (sp, cksum2[ONE1] -> success[ONE1]);
}
```

IF: cond, IN → OUT
1D or 2D ?

Computation assignment
Multiple slabs / proc

Loop: idx, cond, IN → OUT
Update checksums

INOUT args.
Update in place



Compiler Generated Code: FT “slabs”

```
function mainloop ( -> success) [gen]
  Bool success[ALL1]
{
  init_checksums (sp -> cksum);
  compute_init_cond (sp -> u0[dom0]);

  fft (u0[dom0] -> u1[dom2]);

  loop (iter, cond; cksum[ONE1] -> cksum2[ONE1])
  {
    evolve (sp, iter, u1[dom2] -> u2[dom2]);

    ifft (u2[dom2] -> u3[dom0]);

    checksum (iter, u3[dom0], cksum -> cksum2);
    econd (niter, iter -> cond);
  }

  verify <ONE1> (sp,cksum2[ONE1]-> success[ONE1]);
}
```

```
for (iter=0, cond=1; cond; iter++) {
  evolve (sp, iter, u0_dom2, dom2[pe], u2_dom2);
  cffts3 (sp, finv, u2_dom2 /* mod */, dom2[pe]);

  if (lay1d) { // 1D layout
    swapBeginAM (sw20, u2_dom2, u2_dom0, 0, 3, pe);
    for (int _mi=0; _mi<mdom0.getNo(pe); _mi++) {
      swapEndAM (sw20, u2_dom2, u2_dom0, _mi, 0, 3, pe);
      cffts2 (sp, finv, u2_dom0 /* mod */, mdom0.idx(pe,_mi));
    }
    cffts1 (sp, finv, u2_dom0 /* mod */, dom0[pe]);

  } else { // 2D layout
    swapBeginAM (sw21, u2_dom2, u2_dom1, 0, 4, pe);
    for (int _mi=0; _mi<mdom1.getNo(pe); _mi++) {
      swapEndAM (sw21, u2_dom2, u2_dom1, _mi, 0, 4, pe);
      cffts2 (sp, finv, u2_dom1 /* mod */, mdom1.idx(pe,_mi));
    }
    swapBeginAM (sw10, u2_dom1, u2_dom0, 0, 5, pe);
    for (int _mi=0; _mi<mdom0.getNo(pe); _mi++) {
      swapEndAM (sw10, u2_dom1, u2_dom0, _mi, 0, 5, pe);
      cffts1 (sp, finv, u2_dom0 /* mod */, mdom0.idx(pe,_mi));
     }
  }

  checksum_dom (u2_dom0, dom0[pe], c1e_PPEn[pe]);
  swapBegin (SwPPE2ONEn, c1e_PPEn, c1e_ONEn, 0, 6, pe);
  econd (niter, iter, cond);
  swapEnd (SwPPE2ONEn, c1e_PPEn, c1e_ONEn, 0, 6, pe);
  if (ONE1.getNo(pe)) {
    sumproc (c1e_ONEn, ONEn[pe], c2e);
  }
  if (ONE1.getNo(pe)) {
    checksum_set (sp, c2e, iter, cksum /* mod */);
  }
}
```

Optimizations

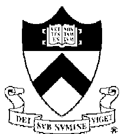
- Expand subgraphs
- Overlap communication
- Reuse entire datastructures
  - 12 → 4 vs. 3 (handwritten)
  - auto advantage lg. scale apps.

CGD Global Domain Access

Parallel computation invocation	Syntax
arg-list is a list of datastructure [decomposition] (local access) datastructure [decomposition+] (local and global access)	*func-name* <task-part> (IN-arg-list → OUT-arg-list); arg [part] arg [part+]
Examples	
Both arguments access local elements only	foo <ALL1> (A[pa] -> B[pb]);
Input argument can access global elements	foo <ALL1> (A[pa+] -> B[pb]);
Global elements can be written to output	foo <ALL1> (A[pa] -> B[pb+]);

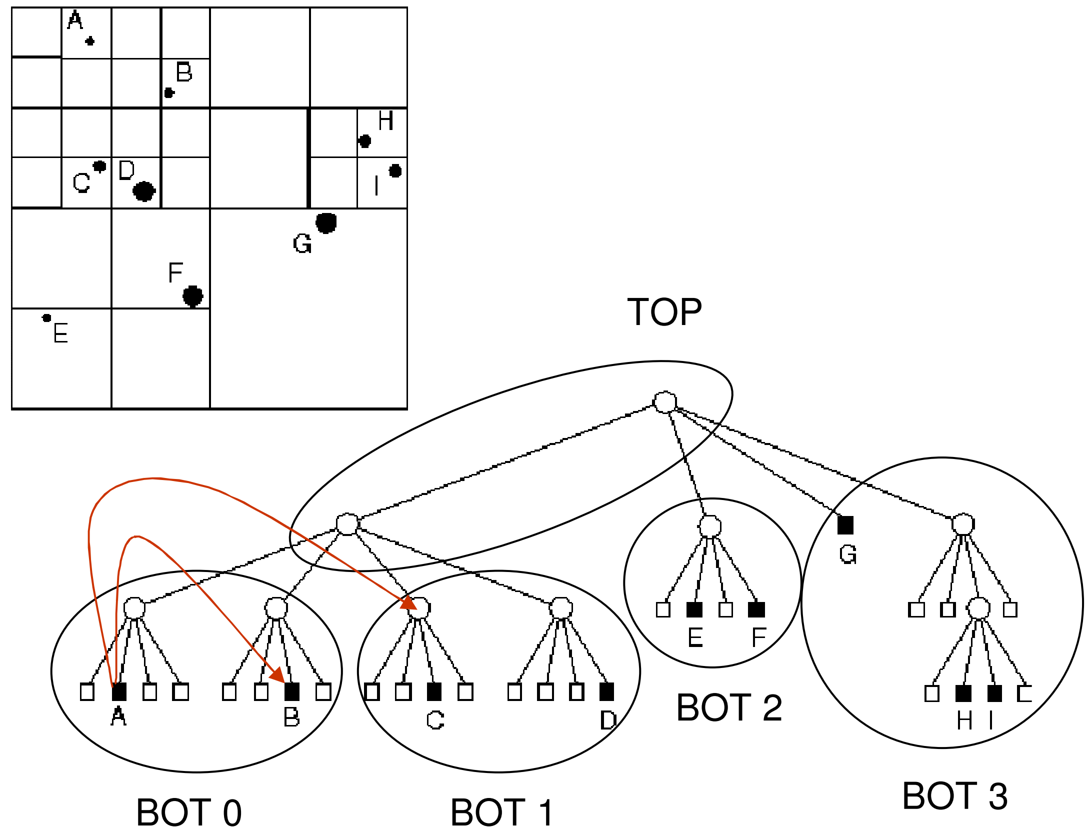
- Restrictions
  - Global domain access not allowed for INOUT arguments
  - Global read, write use a new get operator (slower than local get)
- Read-only: data elements can become replicated
- Write-only: writes are ensured a sequential ordering (impl. specific)

CGD provides both a faster local and a slower global access mode; distribution rules can avoid using global access altogether



CGD Global Domain Example: Barnes Hut

- Distributed octree
  - Replicated TOP
- Each iteration
  - Score weights
  - Repartition tree
  - Insert bodies
  - Center of mass
  - Compute interactions
    - body to node
    - body to body
    - Local/global read
  - Advance particles



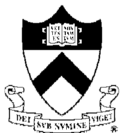
Global Domain Access Example: Barnes-Hut

```
// Main loop
loop ( idx, cond ;
       body0 [tree_loc0], body_sc0 [tree_loc0], stats0 [PPEn],
       tree_loc0 [PPEn], tree_top0 [PPEn], tree_bot0 [PPEn], list0 [PPEn]
     ->
       bodyout [tree_loc], body_sc [tree_loc], stats [PPEn],
       tree_loc [PPEn], tree_top [PPEn], tree_bot [PPEn], list [PPEn] )
{
   // Build partitions
   Score    ( body_sc0 [tree_loc0], tree_top0, tree_bot0 -> node_sc0 [tree_loc0] );
   balance ( par, node_sc0 [tree_loc0+] -> tree_top01 );   // loc0 > top0

   // Build top tree
   Minmax    ( body0 [list0] -> M );
   makeTree  ( body0 [list0], tree_top01, M -> body [tree_loc+], tree_loc [PPEn+] );
   aliasPart ( tree_loc, tree_top01 -> tree_bot, tree_top, list );
   endCond   ( par, idx -> cond );

   // Compute center of mass
   CtrMass ( body [tree_loc], tree_top, tree_bot -> node [tree_loc] );

   // Hierarchical force calculation
   force    ( par, M, body [tree_loc+], node [tree_loc+], stats0 [PPEn] ->
              body_sc [list], body_ac [list], stats [PPEn] );

   // Move particles
   advance ( par, idx, body [list], body_ac [list] -> bodyout [list] );
}

// Compute node center of mass starting from bodies
function CtrMass (body_pm, tree_top, tree_bot, tree_loc -> node_pm)
  BodyPM body_pm [tree_loc], NodePM node_pm [tree_loc],
  TreePart tree_top[PPEn], tree_bot[PPEn], tree_loc[PPEn]
{
  part tree_loc = tree_top + tree_bot;

  ctrmassb (body_pm [tree_bot]  -> node_pm [tree_bot]);
  ctrmasst (node_pm [tree_bot+] -> node_p2 [tree_top]);
  copy     (node_p2 [tree_top]  -> node_pm [tree_top]);
}
```

Insert bodies
global domain write

Force computation
global domain read

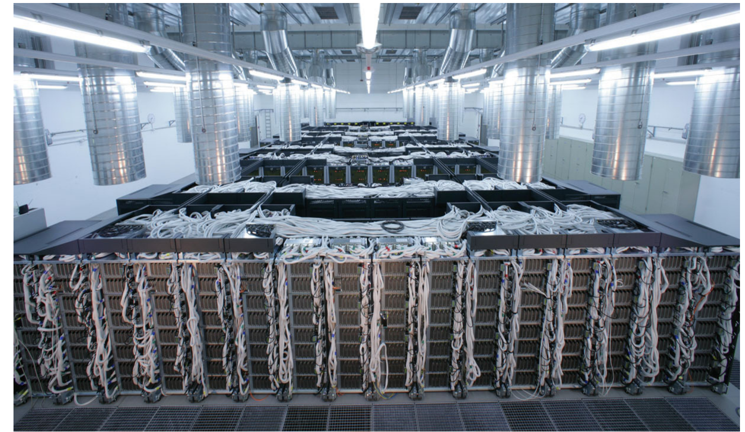
Tree union
decomposition rule



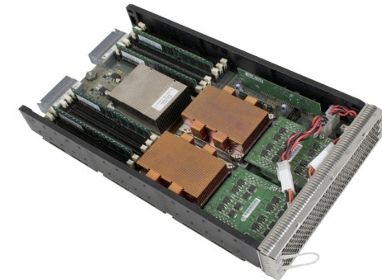
Outline

- Motivation
- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- ➡ Performance
- Cost Model

Machine Setup

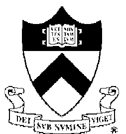


- SGI Altix 4700
  - 256 boxes
    - 2x2 Itanium 1.6GHz, 24MB L3
    - SHub ASIC, NumaLink 4
  - ccNUMA, directory cache coherence
  - MPI (1 usec), SHMEM rely on fast_bcopy
  - CGD w/ SHMEM, MPI



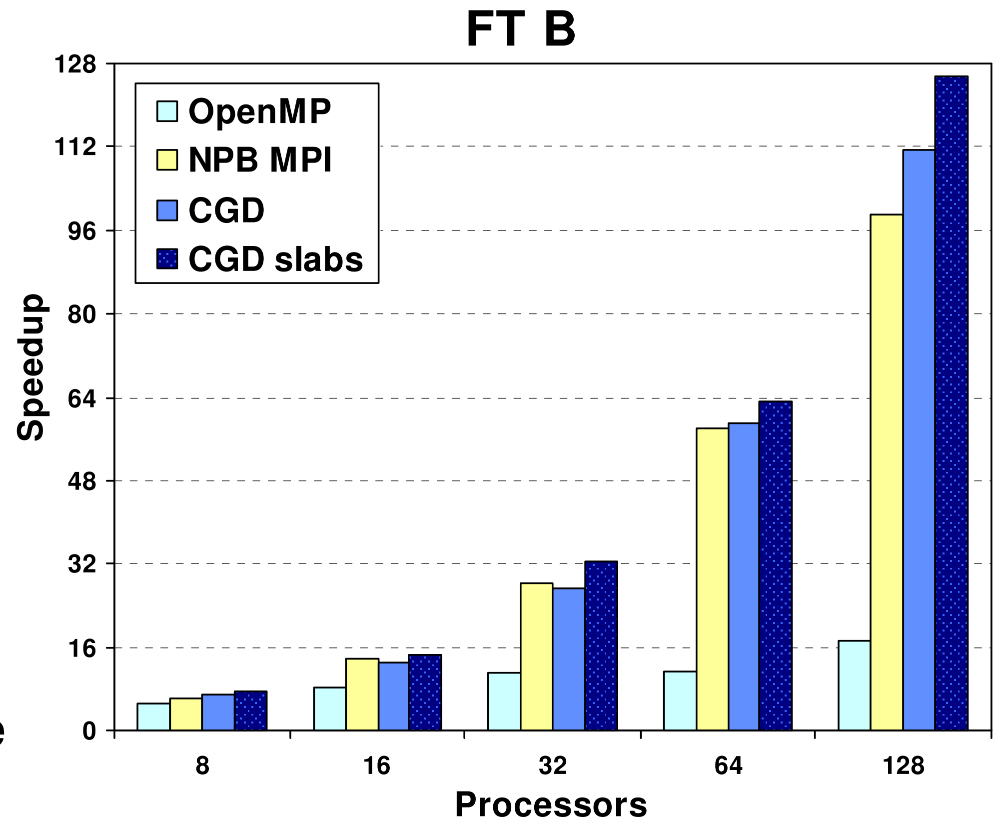
- Opteron SMP
  - 2x4 Shanghai 2.3 GHz, 6 MB L3
  - MPICH 2-1.0.8
    - shared memory support, zero copy
  - CGD w/ pthreads

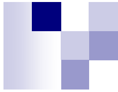
- GCC 4.3.3 -O3 -unroll loops (-fopenmp)

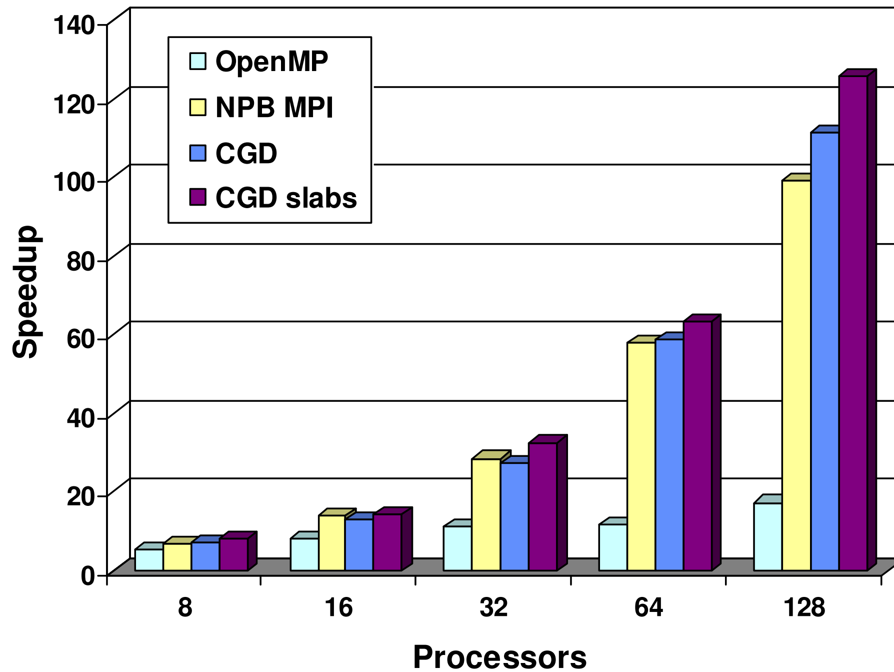


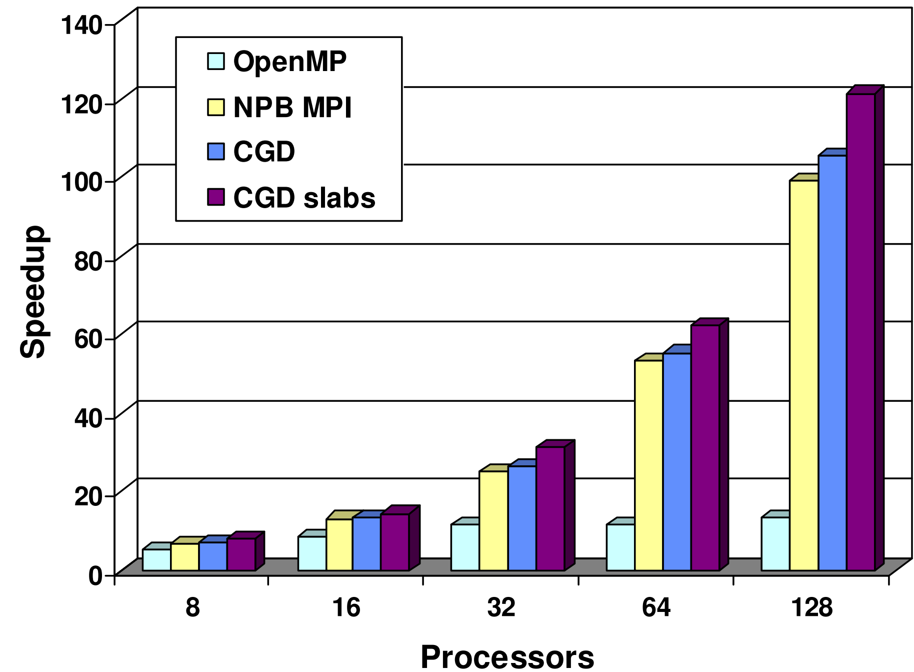
NPB FT performance, SGI Altix 4700

- NPB FT Implementations
  - MPI (Fortran) by NPB 2.3
  - OpenMP (C) by Omni Compiler Project
  - CGD w/ SHMEM, MPI
    - NPB algo
    - “slabs” algo
- Overall speedup ~ linear
  - Based on sequential runtime
  - 16, 32 proc: 14.4x, 32.5x
- CGD slightly faster than MPI (later)
  - CGD “slabs” faster than both







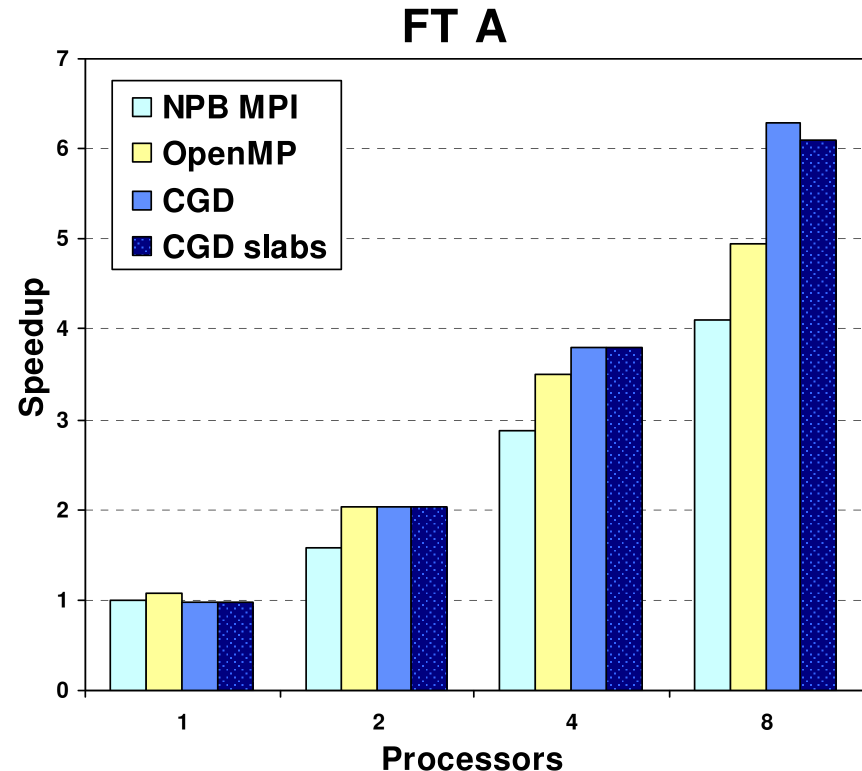


- CGD vs. MPI : data handling 3x faster
  - 500 code lines, MPI fine-tuned to Pentium architecture
  - Case for a higher level abstraction
  - Runtime should be fine-tuned to machine, not application

CGD higher level abstraction increases performance portability; optimizations are moved away from programmer

NPB FT performance, 2x4 Opteron

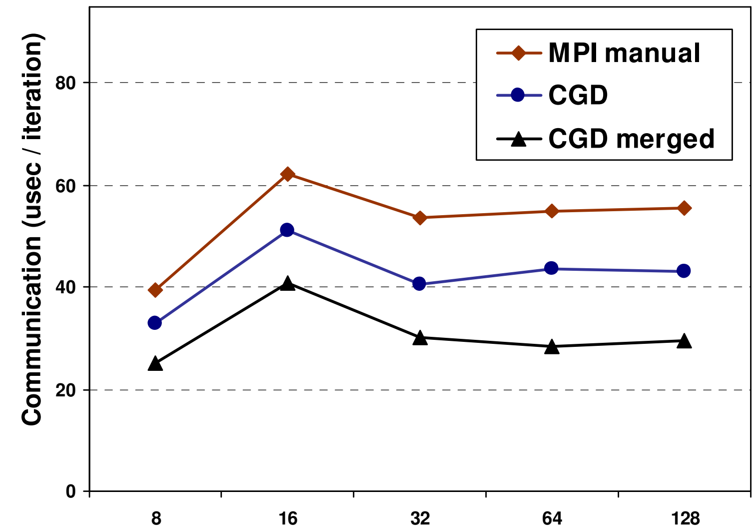
- CGD w/ pthreads
  - Direct DS to DS copy
- CGD vs. MPI
  - 6.3x vs. 4.1x (+50%)
  - MPI DS → buf, buf → DS
- Memory BW bottleneck
- CGD faster than OpenMP
  - Explicit data placement

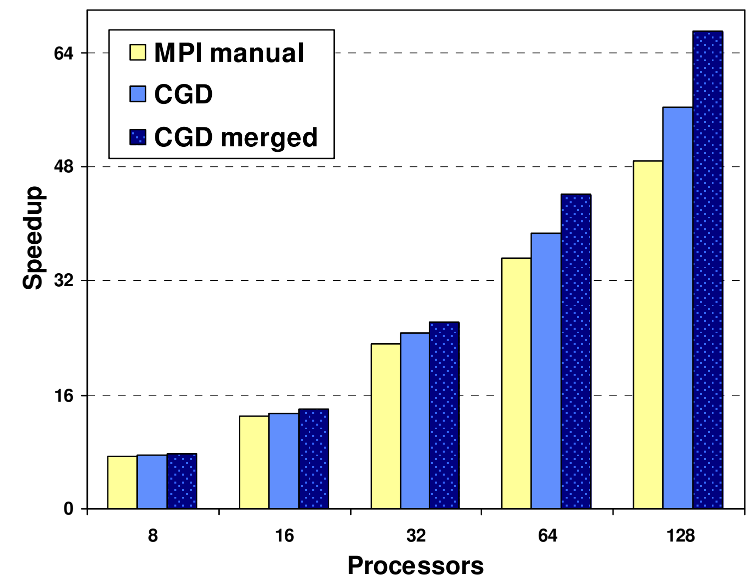


SMP: CGD high level abstractions are implemented using direct mem. access, avoiding MPI overheads

Heat Diffusion performance, Altix 4700

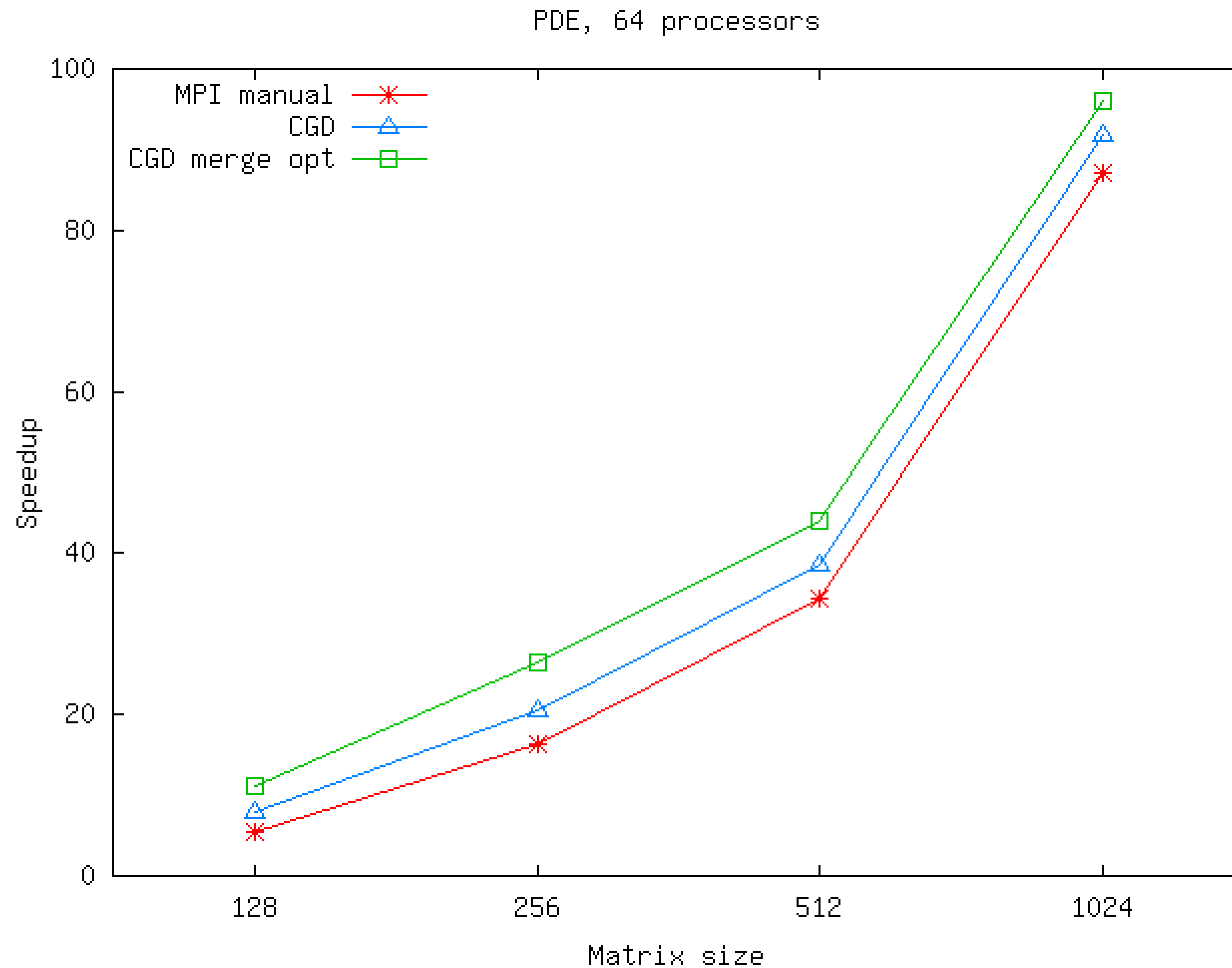
- 2D PDE solver
  - MPI, CGD
  - CGD “merged” comm. steps
- Small comp / comm ratio (2D)
  - Latency bottleneck
- CGD w/ SHMEM faster than MPI
  - CGD “merged” faster than both



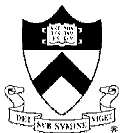


Altix: CGD abstractions are implemented w/ SHMEM, avoiding MPI overheads

Heat Diffusion Performance, Altix 4700

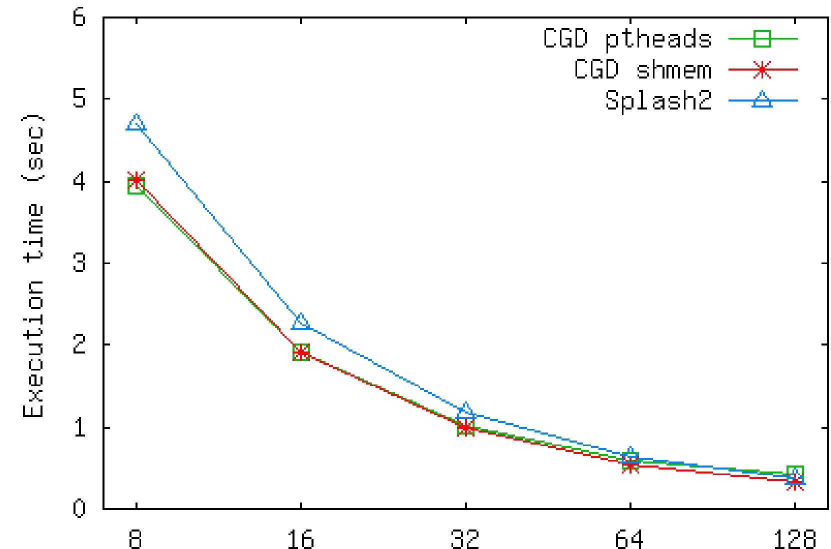


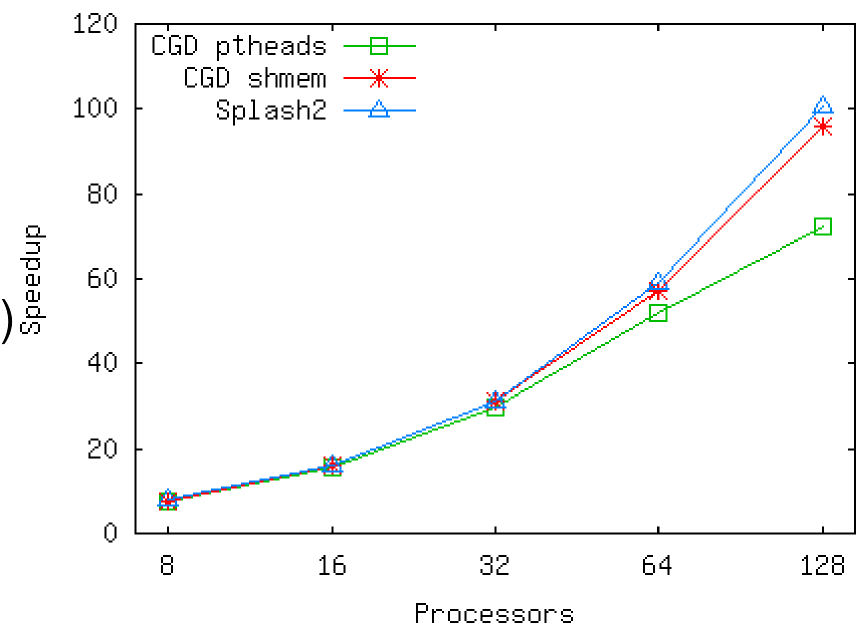
- 512x512 to 1024x1024 : superlinear jump (latency bottleneck)



Barnes-Hut performance, Altix 4700

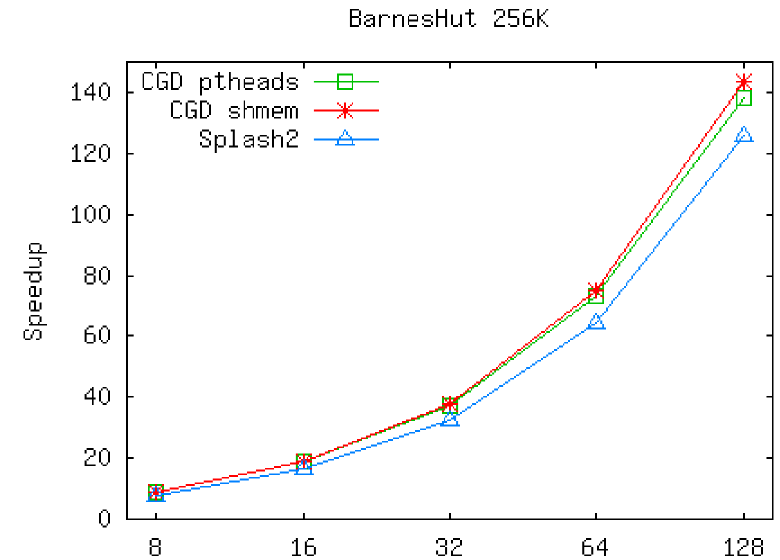
- Barnes Hut implementations
  - Splash2 w/ pthreads
  - CGD w/ pthreads, w/ SHMEM
- CGD global access
- Datasets: 32K, 256K, 1M particles
- CGD vs. Splash2 32K
  - 1 proc: 30.89 vs. 37.07 sec
    - 8 vs. 4 by ptr, node size
  - 128 proc: 0.32 vs. 0.37 sec
  - 95.87 (relative), 115.10 (absolute) vs. 100.53
- CGD SHMEM faster than CGD pthreads for 32K – sync primitives
  - Negligible for larger problems

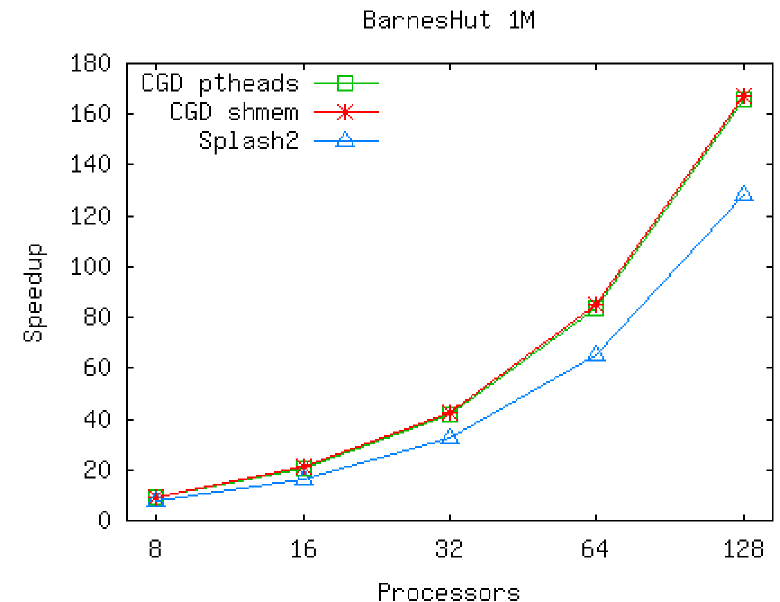




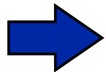
Barnes-Hut performance, Altix 4700

- CGD vs. Splash2 256K, 1M
  - Faster execution, speedup
  - 1M 128 proc: 10.91 vs. 13.54 sec
- Remote node access
  - CGD: remote block transfer
    - stored locally until invalid
  - Splash2: remote memory read
  - CGD – mostly local access
    - Global requests: 114,916, global aliased locally: 92,636,455
    - better remote cache / TLB behavior
  - Independent work draws similar conclusion

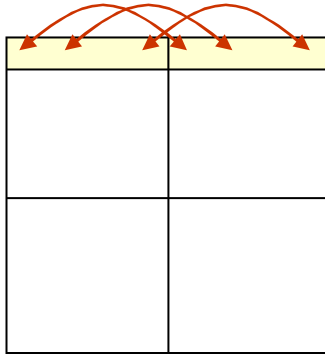




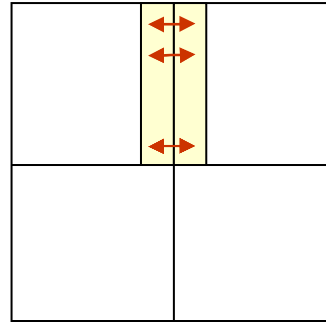
Outline

- Motivation
- Thesis Contribution
- Background on Parallel Programming Models
- Coarse Grain Dataflow Model
- Optimization Process, Design Space Exploration
- Language Implementation, Programming Effort
- Performance
-  Cost Model

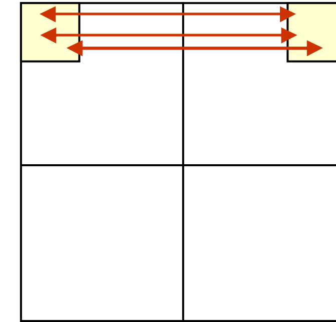
DBSP on Unbalanced Routing Problems



a. North-South 1D FFT
T = log (p)



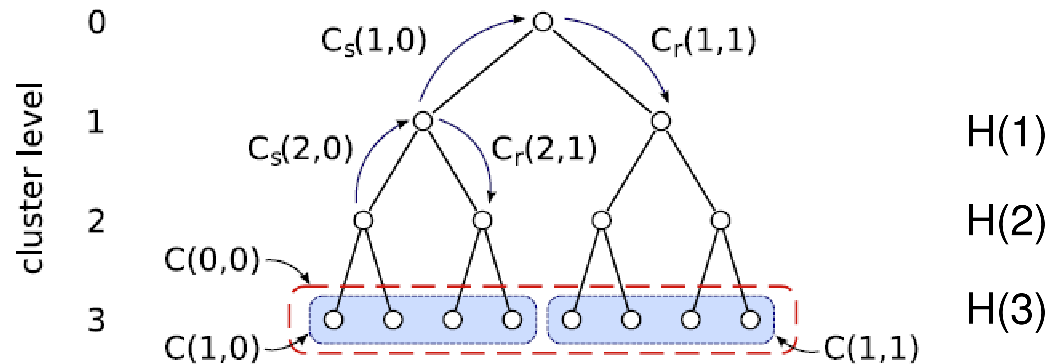
b. PDE nearest neighbor exchange
T = log (p)



c. Corner exchange (same area)
T = sqrt[4] (p)

- Kernels a, b used by scientific computing
- DBSP h-relation : each proc. sends, receives at most h packets
  - T = w + h g(i) + l(i)
- DBSP h-relation, meta-routing K_{12}-routing, (h,m)-relation (m is total)
  - No distinction between a, b, c => T = sqrt[4] (p)
  - Results overestimated for a, b

αDBSP Hierarchical Bandwidth Model



- Define H-relation : messages traveling up to each level

$$H(i) = \frac{1}{2^{i'}}\, max_{0 \leq u < 2^{i+1}} \; \{|C_s(i+1,u)|,\; |C_r(i+1,u)|\}$$

- H-relation is upperbounded by (h,α)-relation

$$H(i) \;\; \leq \;\; \frac{1}{2^{i'\alpha}}\, h$$

- αDBSP machine executes (h,α)-relations each superstep

$$\begin{aligned} T_{DBSP}(i,h,w) &= w + h\,g(i) + l(i) \\ T_{\alpha DBSP}(i,h,\alpha,w) &= w + h\,g(i,\alpha) + l(i,\alpha) \end{aligned}$$

αDBSP Machine Parameters

i-superstep execution bounds on a $p' = p/2^i$ processor submachine. Lower bounds apply to tight relations.

Machine	α_M	(h,α)-relation	DBSP	αDBSP		$g(i,\alpha)$	$l(i,\alpha)$
Butterfly	0	$\alpha \geq 0$	$O(h\,log(p'))$	$O(h\,log(p'))$	$\Omega(h + log(p'))$	$O(log(p'))$	$O(log(p'))$
Pruned nD Butterfly	1/2, $1/n$	$\alpha < \alpha_M$	$O(h\,p'^{\alpha_M})$	$O(h\,p'^{\alpha_M - \alpha})$	$\Omega(h\,p'^{\alpha_M - \alpha})$	$O(p'^{\alpha_M - \alpha})$	$O(log(p'))$
		$\alpha = \alpha_M$		$O(h\,log(p') + log^3(p'))$	$\Omega(h + log(p'))$	$O(log(p'))$	$O(log^3(p'))$
		$\alpha > \alpha_M$		$O(h + log^3(p'))$	$\Omega(h + log(p'))$	$O(1)$	$O(log^3(p'))$
nD Mesh	$1/n$	$\alpha < \alpha_M$	$O(h\,p'^{\alpha_M})$	$O(h\,p'^{\alpha_M - \alpha} + p'^{\alpha_M})$	$\Omega(h\,p'^{\alpha_M - \alpha})$	$O(p'^{\alpha_M - \alpha})$	$O(p'^{\alpha_M})$
		$\alpha = \alpha_M$		$O(h\,log(p') + p'^{\alpha_M})$	$\Omega(h)$	$O(log(p'))$	$O(p'^{\alpha_M})$
		$\alpha > \alpha_M$		$O(h + p'^{\alpha_M})$	$\Omega(h)$	$O(1)$	$O(p'^{\alpha_M})$

- Table computed plugging in machine parameters to prev. slide formulas
  - Improvement of DBSP, $\alpha > \alpha_M$
  - Identical when $\alpha = 0$

North-South 1D FFT Cost Analysis

$$T_{\alpha BSP}(n) = n\,log(n\sqrt{p}) + \sum_{s=0}^{log(p)/2-1} (ng(2s+1, 1/2) + l(2s+1, 1/2))$$

$$T_{DBSP}(n) = n\,log(n\sqrt{p}) + \sum_{s=0}^{log(p)/2-1} (ng(2s+1) + l(2s+1))$$

$$T_{DBSP+}(n) = n\,log(n\sqrt{p}) + \sum_{s=0}^{log(p)/2-1} 2\,K_{12}route\left(n, \left\lceil \frac{2^s n}{2^{2s}} \right\rceil, log(p)-1-2s\right)$$

$$= O\left(n\,log(n\sqrt{p}) + \sum_{s=0}^{log(p)/2-1} \sum_{j=0,\, i=log(p)-1-2s+j}^{2s} \left(min\{n, \left\lceil \frac{n}{2^s} \right\rceil 2^j\} g(i) + l(i) + Pf(i)\right)\right)$$

On the pruned butterly we have:

$$T_{\alpha BSP}(n) = O\left(n\,log(np) + n \sum_{i=0}^{log(p)/2} (2i + (2i)^3)\right) = O\left(n\,log(n) + n\,log^2(p) + log^4(p)\right)$$

$$T_{DBSP}(n) = O\left(n\,log(np) + n \sum_{i=0}^{log(p)/2} (2^i + 2i)\right) = O\left(n\,log(n) + n\sqrt{p}\right)$$

$$T_{DBSP+}(n) = O\left(n\,log(np) + \sum_{s=0}^{log(p)/2} \left(\left\lceil \frac{n}{2^s} \right\rceil^{1/2} n^{1/2} 2^s + s^3\right)\right)$$

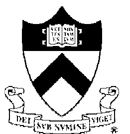
$$= O\left(n\,log(np) + \sum_{s=0}^{log(p)/2} \left(n 2^{s/2} + n^{1/2} 2^s\right) + log^4(p)\right) = O\left(n\,log(n) + n\sqrt[4]{p} + n^{1/2}\sqrt{p}\right)$$

DBSP Family Comparison

Execution time on pruned butterfly: a.) Generalized Broadcast; b.) North-South FFT; c.) Nearest Neighbor Exchange

	DBSP	DBSP+	αDBSP	
a.)	$O(np\sqrt{p})$	$O(np)$	$\Omega(np)$	$O(np)$
b.)	$O(n\,log(n) + n\sqrt{p})$	$O(n\,log(n) + n\sqrt[4]{p} + n^{1/2}\sqrt{p})$	$\Omega(n\,log(np) + log^2(p))$	$O(n\,log(n) + nlog^2(p) + log^4(p))$
c.)	$O(n\sqrt{p})$	$O(n\sqrt[4]{p} + n^{1/2}\sqrt{p})$	$\Omega(n + log(p))$	$O(n\,log(p) + log^3(p))$
$n = 1$				
a.)	$O(p\sqrt{p})$	$O(p)$	$\Omega(p)$	$O(p)$
b.)	$O(\sqrt{p})$	$O(\sqrt{p})$	$\Omega(log^2(p))$	$O(log^4(p))$
c.)	$O(\sqrt{p})$	$O(\sqrt{p})$	$\Omega(log(p))$	$O(log^3(p))$
$\sqrt{p} \leq n \leq p$				
a.)	$O(np\sqrt{p})$	$O(np)$	$\Omega(np)$	$O(np)$
b.)	$O(n\sqrt{p})$	$O(n\sqrt[4]{p})$	$\Omega(n\,log(p))$	$O(n\,log^2(p))$
c.)	$O(n\sqrt{p})$	$O(n\sqrt[4]{p})$	$\Omega(n)$	$O(n\,log(p))$

- NS FFT, PDE total comm. time = $O\left(\sum_{i=0}^{log(p)-1}(ng(i,1/2) + l(i,1/2))\right)$
- Bandwidth growth factor α =1/2, matching a 2D mesh, pruned butterfly



Thesis Plan

- Motivation
- Background
- Coarse Grain Dataflow Model
  - Design Space Exploration, Optimizations
- Language Specification
  - Programming, Optimization Effort
- Runtime Library, Compiler Implementation
  - Performance
- Hierarchical Bandwidth Cost Model
  - Model Accuracy, Analysis Effort

Source code, examples, benchmark results available at

http://www.cs.princeton.edu/~asoviani/cgd/