

Optimizing Communication Scheduling using Dataflow Semantics

Adrian Soviani and Jaswinder Pal Singh

Department of Computer Science, Princeton University

{asoviani, jps}@cs.princeton.edu

Abstract

We show how coarse grain dataflow semantics (CGD) applied to SPMD algorithms makes application development and design space exploration simpler compared to message passing, at the same time providing on par performance. CGD applications are specified as dependencies between computation modules and data distributions; communication and synchronization are added automatically and optimized for specific architectures, relieving programmers of this task.

Many high level algorithm changes are easy to implement in CGD by redefining data distributions. These include exposing communication overlap by decreasing task grain, and aggregating communication by replicating data and computation.

We briefly present a coordination language with dataflow semantics that implements the CGD model. Our implementation currently supports MPI, SHMEM, and pthreads. Results on Altix 4700 show our optimized CGD FT is 27% faster than the original NPB 2.3 MPI implementation, and the optimized CGD stencil has a 41% advantage over handwritten MPI.

1. Introduction

Writing efficient parallel applications that are portable across architectures can be a daunting task. The programming model has a great impact on the implementation effort. The most desired features of a model include good programmability, efficiency, portability, and ease of design space exploration. Achieving a good trade-off between these features is not trivial: a model close to hardware yields good performance but increases programming effort and may jeopardize portability. Relying on a high level library or language to do all the work may not result in the best performance.

The key to address these issues is finding the right high level abstraction that is general enough to make programming and algorithm exploration simple, yet it provides sufficient details to make efficient runtime or compiler implementation practical. Such an abstraction may lead to better performance than lower level libraries having more room to exploit architecture specific optimizations within the compiler or runtime. E.g., algorithms implemented in UPC have shown performance exceeding MPI when using faster fine-grain communication and improved scheduling [3, 9, 13].

This paper presents how coarse grain dataflow semantics can describe data and task parallelism at high level. Us-

ing explicit dependencies between parallel computations and data decompositions, a scheduler can optimize communication and maximize overlap. Communication patterns can be automatically fine-tuned to specific architectures by relying on the most efficient available primitives¹.

Our programming model combines and develops several concepts defined by existing models. Similar to dataflows, CGD uses the single assignment rule, and data dependencies determine scheduling. Unlike dataflows, our model relies on user specified decomposition and assignment for datastructures and computations. We feel that these requirements are essential to providing performance transparency and good efficiency. Similar to message passing SPMD and the newly developed Partitioned Global Address Space (PGAS) languages, CGD uses explicit data distribution and computation assignment. Unlike these models, CGD takes advantage of its dataflow semantics and explicit distribution rules to schedule computations and automatically insert, aggregate, and overlap communication; distribution rules describe how distributions can be transformed when needed. If message passing implementations require explicit messaging and a greater programming effort, PGAS languages rely on non-trivial optimizations to aggregate remote data requests and achieve similar performance².

CGD datastructure distributions are arbitrarily defined by the user rather than being built in the language. This choice makes possible data replication, and allows any distribution to be defined as long as the user creates a domain representation; distributions simply specify a list of domains assigned to each process. Distributions define parallel computations as assignments of sequential computations to processes, each computation acting on local domains of global datastructures. Sequential computations are functions defined in an iterative language such as C++.

Exploring data layouts and communication patterns requires significantly less effort in CGD than using message passing. In CGD these changes are done only by redefining data distributions, while in message passing they require rewriting the messaging, buffering, and synchronization code, and reordering computations when needed. To illustrate

¹On SGI Altix shared memory access or SHMEM; on clusters shared memory access within nodes, and RDMA between nodes.

²Iterative languages can use dataflow techniques to better coordinate communication. When tasks specify which data domains they access, a compiler may aggregate and retrieve remote elements earlier.

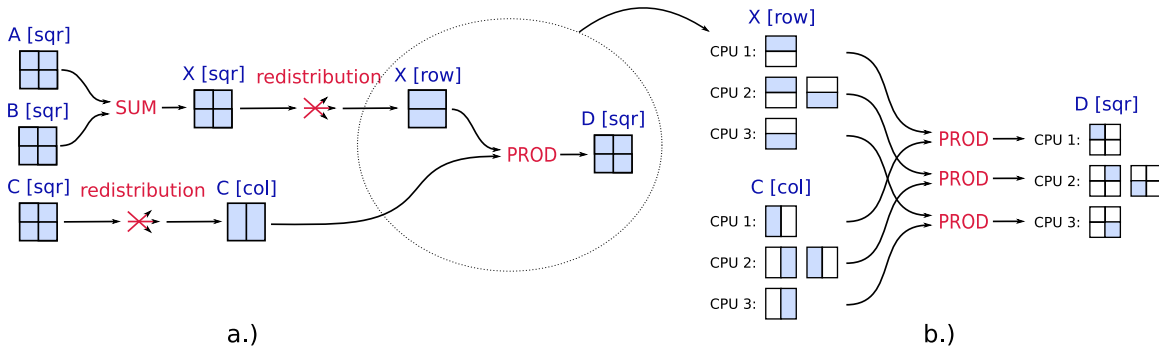


Figure 1: Dataflow describing parallel computation for $D = (A + B) * C$ where A, B, C, D are matrices. a.) CGD specifies dependencies between parallel computations and matrix distributions b.) each parallel computation is a list of sequential computations acting on matrix subdomains. More than one computation may be assigned to each process

this concept we present the “slabs” FT algorithm that allows communication overlap, and the stencil algorithm that aggregates communication steps [1, 3].

The paper is organized as follows. Section 2 presents the CGD model and argues about the simplicity of implementing and optimizing applications. Section 3 shortly describes the features and syntax of a coordination language with dataflow semantics that implements the model. Section 4 summarizes related work. In Section 5 we evaluate the performance of the basic and optimized algorithms for the CGD FT and stencil benchmarks.

2. Programming Model

This section first describes how CGD defines algorithms in terms of computations and data distributions, and how these algorithms are executed in parallel. Then it presents two high level optimizations that are easily done by changing data decompositions.

Dataflow Let’s consider a dataflow graph where nodes are either $(datastructure, distribution)$ pairs or SPMD parallel computations involving no communication. Links in the graph indicate which data distributions are needed to execute a parallel computation, and which computations produce a data distribution. Fig. 1a shows this graph for simple matrix operations where matrices are decomposed as rows, columns, or blocks.

Such a dataflow specification contains both data and task parallelism: all parallel computations are SPMD, and parallel computations which are independent in the graph can be executed concurrently on different process subsets.

Sequential Computations A parallel computation is a list of sequential computations assigned to processors. These are functions that contain no communication primitives and can be implemented in any iterative language. Their input and output arguments are sub-domains of global datastructures (Fig. 1b). Datastructure elements are accessed with global indexes, however a datastructure implementation only needs to provide access to local sub-domains. Computations pro-

duce data distributions with non-overlapping domains unless computation replication is desired.

Sequential computations are allowed to modify datastructure arguments. These arguments are logically duplicated before execution; the compiler avoids duplication when possible by using a single underlying datastructure (Section 3).

Distribution Rules Parallel computations produce data with a distribution while other computations may require it as a different distribution. The new distribution can be created by moving data between processes.

When a compiler knows everything about datastructure organization and distribution domains, data movement can be done without further user involvement. In the general case, arbitrarily defined decompositions don’t allow this. CGD relies on user defined distributions, and it requires programmers to create these distributions, to specify rules for transforming them, and to compute redistribution matrices. Distribution rules are defined in terms of domain inclusion and union, and redistribution matrices (Table 2e-g). Redistributions can be computed using a predefined function that takes as arguments the source and target distributions (Section 3).

Orchestration A scheduler can automatically generate a sequence of computations, communication, and synchronization primitives running on each process such that all data-computation dependencies are satisfied. Compared to classic dataflow models, the CGD scheduler only needs to determine computation order and insert communication since partitioning and assignment of data and computation are explicit. These constraints simplify the problem, and may lead to better results.

One scheduling solution is traversing the graph in topological order and inserting start or end communication phases when data becomes available or is used. When multiple computations are assigned to a process these are executed in sequence; data needed by an iteration is transferred during the previous iteration³ (Section 3).

³Other schedules may lead to faster execution. Optimizing scheduling using profiler data is an interesting area to investigate.

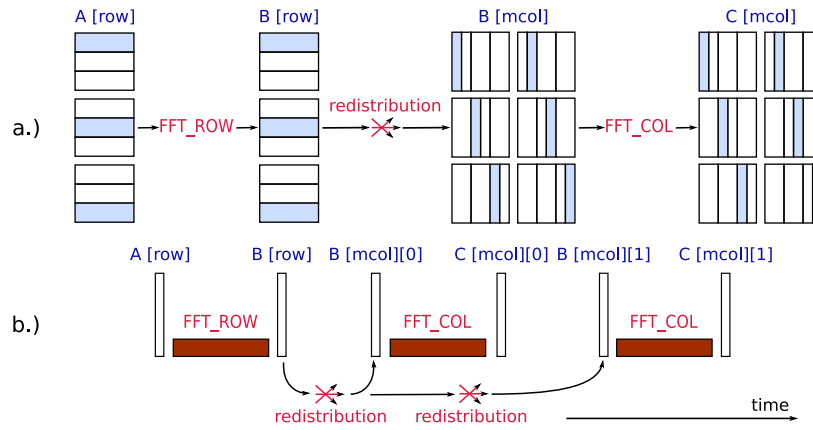


Figure 2: a.) Communication overlap is achieved for a 2D FFT parallel computation by defining the column-wise FFT on multiple smaller domains assigned to each process. b.) Data needed by each column-wise FFT is brought in and waited for as needed

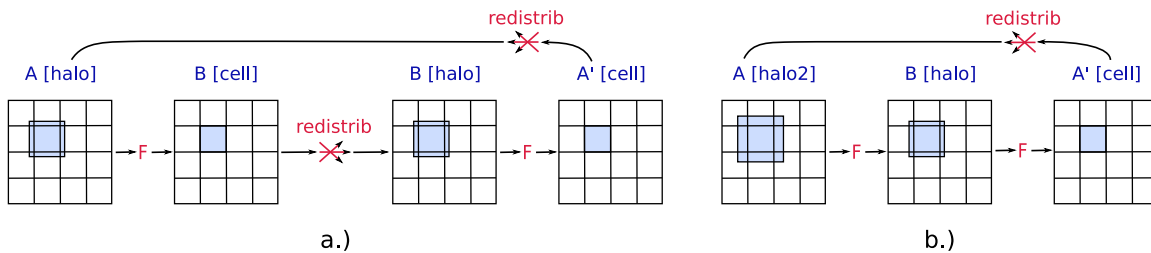


Figure 3: a.) Basic stencil computation with two iterations and two communication steps b.) Communication aggregation is possible by replicating data and computation: first iteration halo elements are assigned to and computed by multiple processes

2.1. Communication Overlapping

Asynchronous communication allows processors do work while data is sent, reducing communication footprint. This technique requires identifying and overlapping independent computations and communication.

FT benchmark NPB FT is a good example to showcase this concept. The benchmark uses spectral methods to solve a partial differential equation. The time consuming code executes a direct 3D FFT followed by a sequence of inverse 3D FFTs. The parallel algorithm used by the NPB MPI implementation decomposes the 3D domain in Z planes and computes X and Y FFTs, then transposes the vector to Y planes and computes Z FFTs. A similar 2D decomposition method using two transpositions is triggered for large CPU numbers when the number of Z planes is too small.

Each transposition is an all-to-all communication pattern that blocks until all remote data arrives, then it computes local FFTs. The “overlap slab” optimization works by splitting each domain into more smaller domains and computing FFTs on each slab. Data needed to compute a slab is sent during the previous iteration. This algorithm can produce code significantly faster than the original algorithm [3]. For this to be effective the system should support asynchronous remote memory operations.

The CGD scheduler will automatically overlap communi-

cation with computation when possible. Fig. 2a shows how a column distribution covered by more smaller domains splits the FFT.COL computation into multiple finer grain tasks assigned to each process. This technique exposes communication overlap since data will be waited for in smaller batches at later times (Fig. 2b). The CGD FT implementation refines the 2D decomposition of the global 3D domain by defining domains of size $a \times b \cdot \text{ftblock} \times \text{dim}$. Slab size is computed to match a target inter-process message size.

2.2. Communication Aggregation

Aggregating communication steps can dramatically improve communication time for latency bound applications that send little data and spend most time with synchronization.

Stencil Computation This micro-kernel implements the nearest neighbor communication pattern commonly employed by PDE solvers; 2D solvers with small work loads are often latency bound. An effective optimization is replicating computation by grouping iterations by two and computing vector values for a larger “halo” domain during the first iteration (Fig. 3). Data needed by the second iteration is already in place and communication is avoided. Fig. 3b shows how this optimization is done in CGD by replacing the `halo` and `cell` argument distributions for computation `F` with `halo2` and `halo`.

Table 1: SLOC line count and speedup for different NPB 2.3 FT implementations. FT B, 128 proc Altix

Version	Layout	Language	Lines	Speedup
Serial	0D	Fortran	704	
OpenMP	1D	C	752	17.21
MPI	1D, 2D	Fortran	1261	103.10
CGD	1D, 2D	C++; CGD	703; 83	111.46
CGD	2D slabs	C++; CGD	740; 90	125.72

2.3. Programming effort

In message passing communication optimization is typically done by reordering program blocks and changing messaging calls. The FT “slabs” and stencil optimizations require data layout changes; all relevant messaging code has to be modified along with buffer management and synchronization.

Table 1 gives a rough estimation of FT programming effort using MPI, OpenMP, and CGD. The CGD C++ sequential computations are derived from the OpenMP C version developed by Omni starting from the serial NPB 2.3 in Fortran. The bulk of these functions is mostly unchanged between all three implementations with the exception of variable renaming and array indexation; they take about 700 lines of code.

The CGD version adds up to 800 lines: C++ section adds code to initialize distributions and implement a few short functions; dataflow section has about 80 lines, 50% being function and type declarations. The NPB 2.3 MPI implementation almost doubles overall code size. OpenMP implementation is slightly shorter than CGD but performance is not on par with MPI or CGD [10].

Adding the “slabs” optimization to the original CGD FT requires trivially changing 10 lines of dataflow code and adding 40 lines of sequential code to define slab distributions.

3. Language Implementation

This section briefly presents a language implementing the Coarse Grain Dataflow programming model. It is a coordination language with dataflow semantics that aims at expressing parallelism at high level; it is not a fully featured language.

Sequential computations are C++ functions containing no communication primitives; extensions to other languages are possible subject to datastructure compatibility. The compiler generates C++ code calling these functions, and template collective operations defined for decomposable datastructure types. We use fragments from the FT “slabs” implementation to illustrate the language syntax and its features (Table 2).

Types The CGD language allows arbitrary types for distributed datastructures, domains, distributions, and redistribution matrices. Domain and datastructure types are declared in CGD and defined by the user in C++ (Table 2a,d). Distributions and redistribution matrices are built-in template datastructures based on domain types (Table 2b,c). The later are distributed according to predefined distributions, frequently being replicated constants. Multi-dimensional arrays and

other basic data types are predefined. When defining a new domain or datastructure type, the user provides a few helper functions that allow data movement between datastructures⁴.

All data nodes from the CGD dataflow graph are (*datastructure, distribution*) pairs. Datastructures with domains that cannot be decomposed are typically described by predefined distributions `ALL1` and `ONE1`, meaning data is replicated among all processes, or kept by a single process. `ALLn`, `ONEn` are similar distributions of `n` elements, and `PPEn` assigns a distinct element to each process. When a distribution is not provided for a datastructure, `PPEn` is used for datastructures with distribution types, and `ALL1` for all other types.

Dataflow The CGD language represents dataflow graphs as parallel functions. The list of parallel computation nodes forms the function body, and the input and output data nodes correspond to function arguments (Fig. 1, Table 2h,l). Parallel computations can be other parallel functions (dataflow subgraphs), or sequential functions (assignments of sequential computations). Parallel functions take datastructure distributions as arguments (Table 2h) while sequential functions take an extra `<task-part>` distribution argument defining how many computations are assigned to each process (Table 2k); `ALL1` is used by default, meaning a single computation is assigned to each process. Parallel and sequential function declarations require all arguments to have types. For all other datastructures the compiler determines types based on these declarations.

Loop constructs are semantically similar to Lucid. The body of the loop is a subgraph that produces outputs starting from inputs (Table 2i). All the output arguments are assigned to the input arguments at the beginning of the next iteration. These copies are avoided if the compiler can map both input and output labels to the same underlying datastructure.

If constructs provide two alternative subgraphs that produce the same outputs (Table 2j). They take as inputs the union of inputs required by either subgraph.

Distribution Rules Datastructures may be transformed between distributions according to rules defined in terms of domain sets (Table 2e-g). The inclusion and merge rules are trivially implemented. Redistribution rules are defined by user by providing a matrix $S_{i,j}$ that transforms distribution *src* to *tar*. For each source domain src_i and target domain tar_j , $S_{i,j}$ specifies a common subdomain, if any, that is copied from src_i to tar_j to build the target distribution. A predefined `computeSwap` function automatically generates this matrix when both source and target distributions are fully known and the intersect domain operation is implemented.

The mechanism for copying remote data is implementation specific. The current CGD runtime supports direct memory copy for pthreads, and data encoding, decoding, and message passing under MPI and SHMEM.

Orchestration The scheduler traverses the graph in topological order to compute the output datastructures. When mul-

⁴These are copy, to/from/sizeBytes, and intersect for domain types.

Table 2: CGD syntax and FT “slabs” example for type declaration, distribution rule specification, parallel computation definition

	Type Declaration	Syntax
a.)	Domain defined by user	type range <i>range-ty</i> ;
b.)	Distribution assigning 0 or 1 <i>range-ty</i> domains per proc Multi-distribution assigning any # <i>range-ty</i> domains per proc	type partition < <i>range-ty</i> > <i>part-ty</i> [predef-part-ty]; type mpartition < <i>range-ty</i> > <i>part-ty</i> [predef-part-ty];
c.)	Redistribution matrix describing a transformation of distributions (partition), and multi-distributions (mpartition)	type swap < <i>range-ty</i> > <i>swap-ty</i> [predef-part-ty]; type mswap < <i>range-ty</i> > <i>swap-ty</i> [predef-part-ty];
d.)	Datastructure defined by user : non-decomposable, and decomposable accepting the specified distributions	type data <i>data-ty</i> ; type data <i>data-ty</i> [part-ty, ...];
FT Example		
	Declaration of 3D domain user type	type range <i>Range3D</i> ;
	Definition of 3D domain multi-distribution Definition of redistribution matrix transforming 3D domain multi-distributions	type mpartition < <i>Range3D</i> > <i>MPartRange3D</i> [PartNP]; type mswap < <i>Range3D</i> > <i>MSwapRange3D</i> [PartNP];
	Declaration of non-decomposable user datastructure types Declaration of 3D complex vector type allowing 3D domain distributions	type data <i>Setup, Checksums, Int, Bool, Cplx</i> ; type data <i>Vector3Dcplx</i> [PartRange3D, MPartRange3D];

	Distribution Rules	Operation	Syntax
e.)	Order : part-a is included in part-b	None	part part-a < part-b < ... ;
f.)	Merge : part-a is union of part-b and part-c	Copy/none	part part-a = part-b + part-c + ... ;
g.)	Swap : part-b can be obtained from part-a	Redistribution	part swap : part-a -> part-b ;
FT Example			
	3D vector: from block-Z to slab-Y decomposition 3D vector: from block-Y to slab-X decomposition	sw21 sw10	part sw21 : dom2 → mdom1 ; part sw10 : dom1 → mdom0 ;
	block-X and slab-X cover same domains block-Y and slab-Y cover same domains	None None	part dom0 = mdom0 ; part dom1 = mdom1 ;

	Parallel Computations	Syntax
h.)	Parallel, sequential function declaration argument type, distribution, and optional inout flag (*) dataflow graph for parallel functions	function <i>func-name</i> (IN-arg-list → OUT-arg-list) data-ty arg arg [part-arg] arg* [part-arg], ... ; { dataflow }
i.)	Loop construct idx-arg is iteration count, cond-arg is end condition	loop (idx-arg, cond-arg ; IN-arg-list → OUT-arg-list) { dataflow }
j.)	If construct cond-arg determines which sub-graph is evaluated	if (cond-arg, IN-arg-list → OUT-arg-list) { true-dataflow } else { false-dataflow }
k.)	Computation invocation, parallel or sequential function	<i>func-name</i> <task-part> (IN-arg-list → OUT-arg-list);
l.)	Dataflow graph	{ distribution-rule loop if computation ; ... ; }
FT Example		
	Declaration of <i>cffts3</i> ; it computes Z-wise FFTs on domain dom of datastructure A sp, dir, dom are read-only arguments result A* uses same storage as input A	function <i>cffts3</i> (sp, dir, A, dom → A*) Setup sp, Int dir, Range3D dom, Vector3Dcplx A [dom], A* [dom];
	standard computation invocation assigning 1 computation per proc, and alternative short invocation	<i>cffts3</i> <ALL1> (sp, dir, A, dom2 → B) ; <i>cffts3</i> (sp, dir, A [dom2] → B [dom2]) ;

NPB FT “slabs” Main Function

```

const Setup sp[ALL1], Int niter[ALL1], finv[ALL1], lay1d[ALL1];
const PartRange3D dom0[ALLn], dom1[ALLn], dom2[ALLn];
const MPartRange3D mdom0[ALLn], mdom1[ALLn], mdom2[ALLn];

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]);
  }

function ifft (A -> D)
  Vector3Dcplx A[dom2], D[dom0]
  {
    cffts3 <ALL1> (sp, finv, A[dom2] -> B[dom2]);
    if (lay1d, B[dom2] -> D[dom0]) {
      // 1D layout
      cffts2 <mdom0> (sp, finv, B[mdom0] -> C[mdom0]);
      cffts1 <ALL1> (sp, finv, C[dom0] -> D[dom0]);
    } else {
      // 2D layout
      cffts2 <mdom1> (sp, finv, B[mdom1] -> C[mdom1]);
      cffts1 <mdom0> (sp, finv, C[mdom0] -> D[dom0]);
    }
  }

```

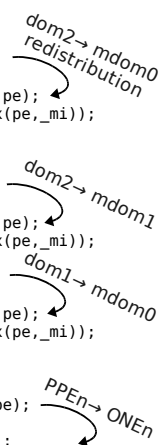
Compiler Generated Code

```

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], cle_PPEn[pe]);
  swapBegin (SwPPE20En, cle_PPEn, cle_ONEn, 0, 6, pe);
  econd (niter, iter, cond);
  swapEnd (SwPPE20En, cle_PPEn, cle_ONEn, 0, 6, pe);
  if (ONE1.getNo(pe)) {
    sumproc (cle_ONEn, ONEn[pe], c2e);
  }
  if (ONE1.getNo(pe)) {
    checksum_set (sp, c2e, iter, cksum /* mod */);
  }
}

```

expands to



m.)

multiple options are possible it preserves the original computation order. The scheduler attempts to fill-in missing links between datastructure distributions using applicable decomposition rules. Redistributions are avoided when distributions can be obtained otherwise (Table 2e,f). The scheduler generates a sequence of computations, data allocations, and data transfer operations that are executed by each processor. Multiple data transfers can be executed concurrently.

For each applied redistribution rule the compiler inserts setup, begin, and end data transfer operations. For multi-distributions (Table 2c) these operations translate to a loop iterating over all domains assigned to a process. E.g., in function `ifft` from Table 2m computation `cffts2 <mdom0>` expands to a loop computing `cffts2` for all “slabs”, and redistributing data from `dom2` to `mdom0` via `SwapBegin/EndAM`. Until the last iteration `SwapEndAM` first ends, then it starts a data transfer that is overlapped with the slab computation.

The compiler implements several optimizations. When allowed, computations are moved out of loop constructs or removed altogether. Data allocations and data transfer setups are moved as early as possible. Communication is scheduled to increase overlap. Parallel function calls are inlined to increase the scope of optimizations.

Datastructure Mapping While the CGD single assignment rule enforces any data node be produced only once, the compiler may allocate fewer underlying datastructures. Multiple data nodes may be assigned to the same C++ datastructure if there is no use conflict, or their values are identical.

The following data nodes are mapped to the same allocated datastructure: i) in and out datastructures corresponding to inout arguments; ii) loop in and out arguments; iii) distinct compatible distributions of the same datastructure; iv) compatible distributions of distinct datastructures without a use conflict. The number of allocated datastructures is reduced using dynamic programming.

In Table 2m, the FT dataflow uses 11 data nodes corresponding to 3D vectors. The handwritten Fortran MPI code relies on reshapable arrays and uses 3 such vectors, while the CGD generated code uses just 4: `u0_dom2`, `u2_dom2`, `u2_dom1`, and `u2_dom0`. CGD is likely to allocate fewer datastructures than handwritten code for large independently developed projects where data conflicts are less tractable.

4. Related work

The message passing programming model requires application developers to design datastructure decompositions and explicitly write marshaling, communication, and synchronization code. Most popular libraries include MPI, SHMEM, and more recently ARMCI and GASNet [4, 15]. A higher level alternative to message passing are distributed array libraries developed for scientific computing applications, such as MPP and Global Arrays [2, 14].

Dataflow models describe computations as a graph, links representing data dependencies. While initially they required

hardware support, gradually they evolved to threaded, coarse grain, and large grain dataflow models by aggregating computation into nodes compiled for iterative machines. Dataflow languages include Lucid, Id, Sisal, Valid [12]. GLU is a dataflow coordination language that makes large scale application development more tractable [11]. While dataflows allow a high level of parallelism and flexibility of scheduling, computation granularity and data replication may impact performance. In CGD data and computation decomposition and assignment are explicit, based on the SPMD view. Initial results show CGD performance is similar to SPMD algorithms implemented for MPI.

In OpenMP data layout and synchronization are implicit. While applications are easy to write many times optimized code is written matching threads with array strides to preserve locality [16]. In Cilk tasks are explicitly spawned and synchronized in a recursive fashion, each task accessing global datastructures [18]. Load balancing is dynamic implemented via work stealing. These languages run only on cache-coherent machines.

A more recent development are the Partitioned Global Address Space (PGAS) languages including Co-array Fortran, UPC, Titanium, Chapel and X10 [5–9]. The global address space abstraction allows processes to access any global data, but local data accesses are faster. A PGAS compiler tries to determine local and global references, aggregate remote messages and overlap communication when possible. How array domains are distributed among processes depends on each language, Co-Array and UPC having some limitations. Chapel provides a wide support for such distributions even adding custom distributions and datastructures. The abstractions provided by these languages make application development easy, however many times optimized code will copy data between local and global datastructure domains to preserve access locality [17]. CGD requires programmers to specify data dependencies making communication easy to schedule and ensuring access locality for possibly replicated domains, including large domains exceeding the cache size⁵.

5. Experimental Results

This section first describes machine setup and measurement methodology, then it evaluates the performance of NPB FT and stencil benchmarks.

Setup First experimental machine is an SGI Altix 4700 with 1.6 GHz Itanium processors, 6.4 Gb/sec NUMalink4 interconnect, and 1024 nodes. Both SHMEM and MPI implementations rely on `fast_bcopy` to transfer data via memory copy. Short messages have very low latency but the amount of communication overlap allowed by asynchronous communication is limited. This machine is used for production, running multiple large-scale applications at the same time. Due to measurement variability all tests were run multiple times and only

⁵A more relaxed CGD model can allow fine-grain remote accesses be implemented via datastructure or compiler support.

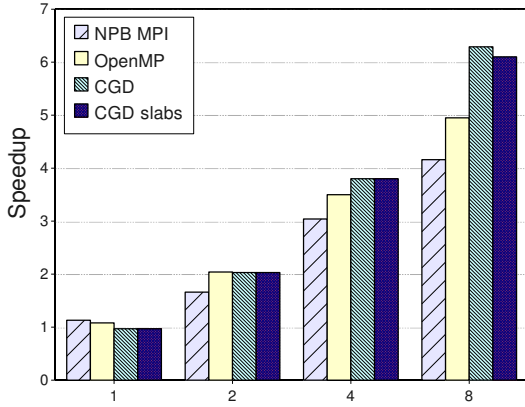


Figure 4: FT A speedup, 2x4 Opteron

the shortest runtime was retained. Time is measured using `gettimeofday` clocks that add a $0.5 \mu\text{sec}$ overhead. All experiments compared against each other were executed by a script on the same machine partition. The second configuration is a dual quad-core machine with 2.3 GHz Shanghai Opteron, 8 GB of memory, kernel 2.6.27-9, and `mpich2-1.0.8`. The MPI library is compiled with shared memory support providing low latency, high bandwidth local messages.

Experiments running on either machine were compiled with version 4.3.3 `gfortran/gcc/gcc -fopenmp` with flags `-O3 -funroll-loops`. For the FT experiment, `total - setup` time is used to compute speedup. When comparing different benchmark implementations all speedups are based on the same runtime: serial NPB Fortran for FT (Altix 4700: FT B 306.98 sec, FT C 1,341 sec; 2x4 Opteron: FT A 7.06 sec), and single CPU handwritten MPI for Stencil (Altix 4700: 4,481 μsec).

NPB FT This benchmark exercises bisection width bandwidth and floating point performance (Section 2.1). We compare four implementations: the original NPB 2.3 MPI Fortran version, the same algorithm implemented in CGD, the “slabs” algorithm in CGD, and the NPB 2.3 OpenMP C version developed by the Omni compiler project. On Altix, the CGD experiments are compiled with a mixed SHMEM and MPI runtime where smaller messages are sent via SHMEM for improved latency; on the SMP, they are compiled with the pthreads CGD runtime.

The CGD C++ computations and the Omni OpenMP version are derived from serial Fortran NPB 2.3. These functions are mostly unchanged and unoptimized, allowing a fair comparison. Table 3 shows sequential `fflow` performance is slightly poorer for C++ vs. Fortran; the same is true when comparing numbers for 1 CPU. `ffcopy` shows a similar side-effect when copying data locally between a 3D vector and a smaller buffer; we assume these are compiler related issues.

In Fig. 5 speedups are nearly linear, thanks to the better cache behavior of the multi-processor code vs. sequential code. CGD is slightly faster than the original NPB Fortran version, which is impressive considering the parallel code is automatically generated and the C++ sequential code has a performance handicap. This performance gap is closed by the

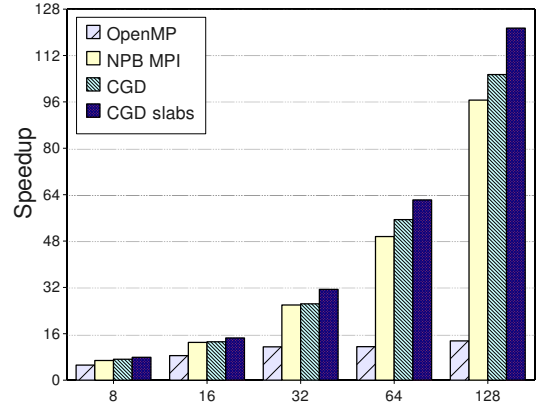


Figure 5: FT C speedup, Altix 4700

	Section	OpenMP	MPI	CGD	CGD slabs
FT B	fft	15.02	2.66	2.26	1.99
	ffflow		1.19	1.23	1.27
	ffcopy		.175	.258	.231
	setup	9.81	.068	.060	.060
	total - setup	17.84	2.98	2.75	2.44
	speedup	17.21	103.10	111.46	125.72
FT C	fft	81.38	12.44	10.70	9.31
	ffflow		5.40	5.73	5.74
	ffcopy		1.99	2.39	1.49
	setup	37.79	0.21	0.22	0.22
	total - setup	99.12	13.88	12.72	11.04
	speedup	13.53	96.61	105.42	121.45

Table 3: FT B, FT C runtime (sec), 128 proc Altix 4700

CGD runtime that runs faster than its NPB MPI counterpart; the improved efficiency is partly due to faster local data handling. The 1D transposition takes 2.58 sec in CGD vs. 5.05 sec in NPB MPI for FT C running on 128 processor Altix.

CGD “slabs” runs significantly faster than NPB MPI due to better communication scheduling and better cache locality: local FFTs will find in cache data copied from other processors. We expect the performance benefit of this algorithm to be greater on interconnects with improved asynchronous messaging support [3].

Fig. 4 shows the original and optimized CGD versions have similar performance on a small SMP, being bound by memory bandwidth. CGD is 27% faster than OpenMP on 8 processors, taking advantage of its explicit data and computation placement and aggregated communication. CGD is 51% faster than MPI since the CGD redistribution copies data directly between vectors, while the MPI transposition rearranges data locally before and after calling `MPI_alltoall`. The later executes an extra copy even for zero copy shared memory MPI implementations.

Stencil This micro-kernel measures nearest neighbor communication performance. Three implementations of the 512×512 2D stencil are used for comparison: MPI “manual” and CGD use the basic one halo exchange per step algorithm, while CGD “merged step” aggregates two communication steps by replicating data (Section 2.2). The MPI code is hand-

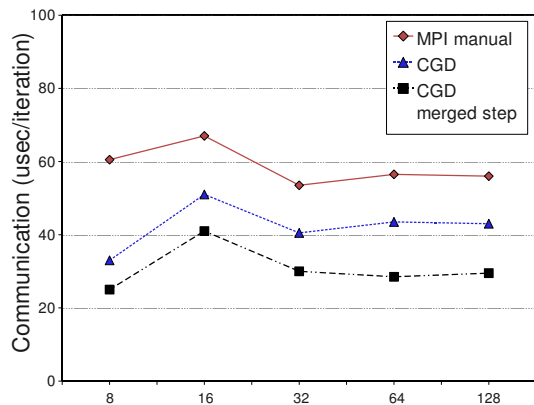


Figure 6: Stencil 512 communication, Altix 4700

written using asynchronous calls: `IRecv`, `encode` and `ISend`, `Waitany` for `IRecv` and `decode`, then `Waitall`. Both CGD versions rely on the SHMEM CGD implementation.

Communication is significantly faster for CGD even for the basic algorithm due to SHMEM improved latency (Fig. 6). The message aggregation optimization further reduces synchronization overhead. Fig. 7 and Table 4 show both CGD versions have speedups better than an already fast MPI implementation.

6. Conclusions and Future Work

We have presented a programming model that describes parallel computations at high level by relying on coarse grain dataflow semantics. Compared to message passing, CGD applications take less effort to write and optimize. The dataflow abstraction allows both general and architecture specific optimizations be implemented in the library not the application. Improving algorithms by changing data distributions is straightforward. The optimized CGD FT size B running on 128 processors has a 27% speed advantage over original Fortran MPI.

We're currently working on porting other NPB benchmarks to our system, and consider implementing benchmarks featuring non-array irregular datastructures. Given the wide support and amount of fine-tuning provided by GASNet, we plan to provide a CGD implementation based on this library.

References

- [1] D. H. Bailey, E. Barszcz, and J. T. Barton et al. The NAS parallel benchmarks. Technical report, The Int. Journal of Supercomputer Applications, 1991.
- [2] V. Balaji and R. W. Numrich. A Uniform Memory Model for Distributed Data Objects on Parallel Architectures. In *Use of High-Performance Computing in Meteorology*, pages 272–294. World Scientific Publishing Co., 2005.
- [3] Christian Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proc. of the 20th IPDPS*, 2006.
- [4] Dan Bonachea. GASNet specification, v1.1. Berkeley, CA, USA, 2002. University of California at Berkeley.

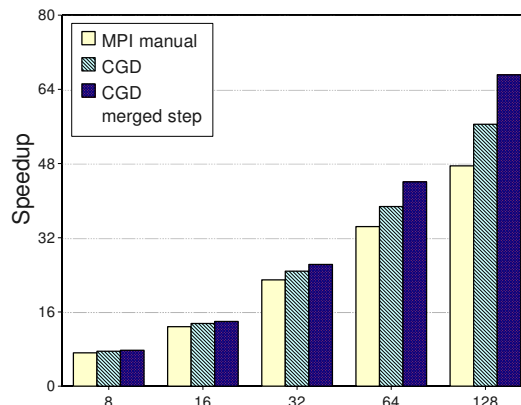


Figure 7: Stencil 512 speedup, Altix 4700

Section	MPI manual	CGD	CGD mrg. step
communication	56	43	29.5
total	94.3	79.35	66.75
speedup	47.51	56.47	67.13

Table 4: Stencil 512 iteration (μ sec), 128 proc Altix

- [5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *Int. Journal of High Performance Computing*, 21:231–312, 2007.
- [6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN*, 2005.
- [7] Wei Yu Chen, P. Husbands, and D. Bonachea et. al. A performance analysis of the Berkeley UPC compiler. In *Proc. of the 17th Int. Conf. on Supercomputing (ICS)*, 2003.
- [8] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *18th Int. Workshop on Languages and Compilers for Parallel Computing*, 2005.
- [9] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Proc. of the Supercomputing Conference (SC)*, pages 1–26, 2002.
- [10] K. Frlinger and M. Gerndt. Analyzing overheads and scalability characteristics of OpenMP applications. In *Proc. of Int. Meeting on High Performance Computing for Computational Science (VECPAR)*, 2006.
- [11] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE Computer Society Press, 1995.
- [12] Wesley M. Johnston, J. R. Paul Hanna, Richard, and J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 2004.
- [13] M. Krishnan and J. Nieplocha. SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *Proc. of the Parallel and Distributed Processing Symposium*, 2004.
- [14] J Nieplocha, R J Harrison, and R J Littlefield. Global arrays: A non-uniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10:197–220, 1996.
- [15] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proc. of 11th IPPS/SPDP*, 1999.
- [16] D. S. Nikolopoulos and E. Ayguadé. The trade-off between implicit and explicit data distribution in shared-memory programming paradigms. In *Proc. of the 15th Int. Conf. on Supercomputing (ICS)*. ACM, 2001.
- [17] I. Patel and J.R. Gilbert. An empirical study of the performance and productivity of two parallel programming models. In *Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7, 2008.
- [18] Keith H. Randall, Charles E. Leiserson, and H. Randall. Cilk: Efficient multithreaded computing. Technical report, 1998.