Compiling Affine Loop Nests for a Dynamic Scheduling Runtime on Shared and Distributed Memory

ROSHAN DATHATHRI, RAVI TEJA MULLAPUDI, and UDAY BONDHUGULA,
Department of Computer Science and Automation, Indian Institute of Science

Current de-facto parallel programming models like OpenMP and MPI make it difficult to extract *task-level dataflow parallelism* as opposed to *bulk-synchronous parallelism*. Task parallel approaches that use point-to-point synchronization between dependent tasks in conjunction with dynamic scheduling dataflow runtimes are thus becoming attractive. Although good performance can be extracted for both shared and distributed memory using these approaches, there is little compiler support for them.

In this article, we describe the design of compiler–runtime interaction to automatically extract coarse-grained dataflow parallelism in affine loop nests for both shared and distributed-memory architectures. We use techniques from the polyhedral compiler framework to extract tasks and generate components of the runtime that are used to dynamically schedule the generated tasks. The runtime includes a distributed decentralized scheduler that dynamically schedules tasks on a node. The schedulers on different nodes cooperate with each other through asynchronous point-to-point communication, and all of this is achieved by code automatically generated by the compiler. On a set of six representative affine loop nest benchmarks, while running on 32 nodes with 8 threads each, our compiler-assisted runtime yields a geometric mean speedup of 143.6× (70.3× to 474.7×) over the sequential version and a geometric mean speedup of 1.64× (1.04× to 2.42×) over the state-of-the-art automatic parallelization approach that uses *bulk synchronization*. We also compare our system with past work that addresses some of these challenges on shared memory, and an emerging runtime (Intel Concurrent Collections) that demands higher programmer input and effort in parallelizing. To the best of our knowledge, ours is also the first automatic scheme that allows for dynamic scheduling of affine loop nests on a cluster of multicores.

Categories and Subject Descriptors: D.3.4 [**Programming Langauges**]: Compilers, Run-time Environments

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Parallelization, dynamic scheduling, task parallelism, dataflow runtime, compiler-runtime framework, distributed-memory architectures, polyhedral model

ACM Reference Format:
Roshan Dathathri, Ravi Teja Mullapudi, and Uday Bondhugula. 2016. Compiling affine loop nests for a dynamic scheduling runtime on shared and distributed memory. ACM Trans. Parallel Comput. 3, 2, Article 12 (July 2016), 28 pages.
DOI: http://dx.doi.org/10.1145/2948975

1. INTRODUCTION

The design of new languages, compilers, and runtime systems are crucial to provide productivity and high performance while programming parallel architectures. Clusters of multicore processors and accelerators such as GPUs have emerged as the parallel

Authors' addresses: R. Dathathri, R. T. Mullapudi, and U. Bondhugula, Dept of CSA, Indian Institute of Science, Bangalore 560012, India; emails: {roshan, ravi.mullapudi, uday}@csa.iisc.ernet.in.

DOI: http://dx.doi.org/10.1145/2948975

architectures of choice both for medium and high-end high-performance computing. Integrating language and programming model design with compiler and runtime support is naturally a powerful approach owing to the amount of information available to the compiler and runtime in generating and executing efficient code. Several systems [Kulkarni et al. 2007; Bosilca et al. 2010, 2012; Ragan-Kelley et al. 2013; Song and Dongarra 2012] have been designed in an integrated manner to various degrees.

Prevalent parallel programming models like OpenMP and MPI have several limitations that prevent them from delivering high parallel performance for shared and distributed memory without an extraordinary amount of programming effort. Though OpenMP and MPI are the most commonly used programming models for shared-memory and distributed-memory architectures, respectively, using them to extract *task-level dataflow parallelism* as opposed to *bulk-synchronous parallelism* is very difficult to almost infeasible. Bulk-synchronous parallelism is one in which all iterations of a parallel loop are synchronized in bulk after their execution and before moving onto the next set of parallel iterations. Task-level dataflow parallelism is the parallelism that can be extracted from the dynamic dependence graph of tasks (a directed acyclic graph)—each task itself can correspond to a block of iterations of a parallel loop or, more generally, a piece of computation that is to be executed in sequence atomically, that is, synchronization or communication is performed only before and after execution of the task but not during it. *Asynchronous parallelization* enabled by explicit point-to-point synchronization between tasks that are actually dependent is known to provide better performance than *bulk-synchronous parallelization* [Buttari et al. 2009; Baskaran et al. 2009; Chandramowlishwaran et al. 2010; Theobald 1999].

Several programming models and runtimes have been proposed to support task-level dataflow parallelism. Some recent works that address this include that of Baskaran et al. [2009], Song and Dongarra [2012], DAGuE [Bosilca et al. 2012], DPLASMA [Bosilca et al. 2010], Concurrent Collections (CnC) [Budimlic et al. 2009], the EARTH model [Theobald 1999], the codelet model [Zuckerman et al. 2011], and SWARM [Lauderdale and Khan 2012], though in varying contexts. The work of Baskaran et al. [2009] is the only one that takes sequential code as input and requires no additional programming effort (i.e., it is fully automatic), but it is applicable for affine loop nests only on shared-memory architectures.

Although good performance can be extracted for both shared and distributed memory using the other task-parallel approaches, it still requires considerable programming effort. As an example, one of the key issues in leveraging dynamic scheduling dataflow runtimes such as CnC is in determining the right decomposition of tasks; the decomposition and granularity of tasks impacts load balance and synchronization overheads. Choosing the right decomposition can improve the performance by orders of magnitude. The decomposition into tasks has a direct connection with loop transformations such as tiling, making a strong case for integration of compiler support.

In this article, we make a contribution towards the design of compiler support and the necessary compiler-runtime interaction for dynamic scheduling dataflow runtimes. For this purpose, we also develop our own runtime. However, the focus of our work is in effectively exploiting runtime support and features through powerful compile-time analysis and transformation to provide a fully automatic solution. This is done so efficient execution on shared as well as distributed memory is achieved with no programmer input. Hence, a distributed-memory cluster of multicores is a typical target. This work's objective is not to develop a generic runtime that replaces existing dynamic scheduling ones like SWARM [Lauderdale and Khan 2012] or CnC [Budimlic et al. 2009]. The choice to develop our own runtime, instead of using SWARM or CnC, in conjunction with our compiler was driven by the need to allow sufficient customization for affine loop nests. Our runtime enables communication that is precise at the

granularity of array elements. Dependences between tasks that execute on the same node (local) and those between tasks that execute on different nodes (remote) are allowed to be handled differently. We use these runtime features by exploiting information extracted from affine loop nests through static analysis.

We describe the design of compiler–runtime interactions to automatically extract coarse-grained dataflow parallelism in affine loop nests on both shared and distributed memory. We use techniques from the polyhedral compiler framework to extract tasks and generate components of the runtime that are used to dynamically schedule the generated tasks. The runtime components are lightweight helper functions generated by the compiler. The task dependence graph is also encapsulated in such compiler-generated functions. This allows the same generated code to execute in parallel on shared memory, distributed memory, or a combination of both. The runtime includes a distributed decentralized scheduler that dynamically schedules tasks on a node. The schedulers on different nodes cooperate with each other through asynchronous point-to-point communication of data required to preserve program semantics. We are also able to automatically obtain overlap of computation and communication, and load-balanced execution. All of this is achieved by code automatically generated by the compiler.

We build a source-to-source transformation tool that automatically generates code targeting a dataflow runtime. While running on 32 nodes with 8 threads each, our compiler-assisted runtime yields a geometric mean speedup of 143.6× (70.3× to 474.7×) over the sequential version and a geometric mean speedup of 1.64× (1.04× to 2.42×) over the state-of-the-art automatic parallelization approach that uses *bulk synchronization*. We also compare our system with past work that addresses some of these challenges on shared memory and an emerging runtime (Intel Concurrent Collections) that demands higher programmer input and effort in parallelizing. When coupled with compiler support including recent advances in automatically generating MPI code for affine loop nests [Bondhugula 2013a; Dathathri et al. 2013], ours is also the first system that allows fully automatic dynamic scheduling for affine loop nests on a cluster of multicores.

The main contributions of this article can be summarized as follows:

—representing the dynamic dependence graph of tasks in a compact manner using helper functions generated by the compiler,
—designing the compiler–runtime interface as a set of compiler-generated functions required by the dataflow runtime,
—designing a novel compiler-assisted dataflow runtime framework that achieves cooperation without coordination in distributed dynamic schedulers,
—implementing the compiler-assisted dataflow runtime in a source-level transformer to allow for dynamic scheduling of affine loop nests on a cluster of multicores,
—an experimental evaluation of the developed system and comparison with a state-of-the-art parallelization approach that uses *bulk synchronization*, demonstrating better load balance and communication-computation overlap, which directly translates into significantly better scaling and performance,
—comparing our fully automatic framework with manually optimized Intel CnC codes making a strong case to develop compiler support for dataflow runtimes.

The rest of this article is organized as follows. Section 2 provides background on runtime design issues. Section 3 describes the design of our runtime in detail and Section 4 presents implementation details. Experimental evaluation is provided in Section 5. Section 6 discusses related work and conclusions are presented in Section 7.

2. MOTIVATION AND DESIGN CHALLENGES

This section describes our objectives and their implications on the design to be proposed.

2.1. Dataflow and Memory-Based Dependences

It is well known that flow or read-after-write (RAW) dependences lead to communication when parallelizing across nodes with private address spaces. On the other hand, memory-based dependences, namely anti or write-after-read (WAR) and output or write-after-write (WAW) dependences, do not lead to communication. Previous work [Bondhugula 2013a; Dathathri et al. 2013] has shown that when multiple writes to an element occur on different nodes before a read to it, only the *last write* value before the read can be communicated using non-transitive flow dependences. Previous work [Bondhugula 2013a] has also shown that the *last write* value of an element across the iteration space can be determined independently (*write-out set*). Hence, memory-based dependences can be ignored when parallelizing across nodes but they have to be preserved when parallelizing across multiple cores of a node that share the address space. We will see that a compiler that targets a runtime for a distributed-memory cluster of multicores should pay special attention to these.

2.2. Terminology

Tasks: A task is a part of a program that represents an atomic unit of computation. A task is to be atomically executed by a single thread, but multiple tasks can be simultaneously executed by different threads in different nodes. Each task can have multiple accesses to multiple shared data variables. A flow (RAW) data dependence from one task to another would require the data written by the former to be communicated to the latter if they will be executed on different nodes. In any case, it enforces a constraint on the order of execution of those tasks, that is, the dependent task can only execute after the source task has executed. Anti (WAR) and output (WAW) data dependences between two tasks are memory based and do not determine communication. Since two tasks that will be executed on different nodes do not share an address space, memory-based data dependences between them do not enforce a constraint on their order of execution. On the other hand, for tasks that will be executed on the same node, memory-based data dependences do enforce a constraint on their order of execution, since they share the local memory. There could be many data dependences between two tasks with source access in one task and target access in the other. All these data dependences can be encapsulated in one inter-task dependence to enforce that the dependent task executes after the source task. So, it is sufficient to have only one inter-task dependence from one task to another that represents all data dependences whose source access is in the former and target access is in the latter. In addition, it is necessary to differentiate between an inter-task dependence that is only due to memory-based dependences and one that is also due to a flow dependence. If two tasks will be executed on different nodes, an inter-task dependence between them that is only due to memory-based dependences does not enforce a constraint on the order of execution. Finally, our notion of task here is same as that of a "codelet" in the codelet execution model [Zuckerman et al. 2011].

Scheduling tasks: Consider the example shown in Figure 1, where there are five tasks, Task-A, Task-B, Task-C, Task-D, and Task-E. The inter-task dependences determine when a task can be scheduled for execution. For instance, the execution of Task-A, Task-B, Task-C, Task-D, and Task-E in that order by a single thread on a single node is valid since it does not violate any inter-task dependence. Let Task-A, Task-B, and Task-D be executed on Node2, while Task-C and Task-E be executed on Node1, as shown in Figure 1. On Node2, Task-A can be scheduled for execution since it does not depend on any task. Since Task-B depends on Task-A, it can only be scheduled for execution after Task-A has finished execution. Task-C in Node1 depends on Task-B in Node2, but the dependence is only due to WAR or WAW data dependences. So Task-C can be scheduled for execution immediately. Similarly, Task-E in Node1 can ignore its WAR

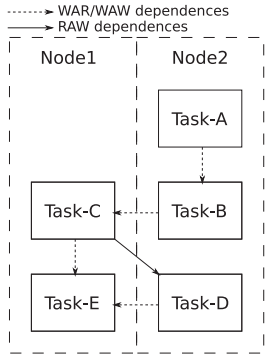


Fig. 1. Inter-task dependences example.

or WAW dependence on Task-D in Node2, but it has to wait for Task-C's completion before it can be scheduled for execution. On the other hand, Task-D in Node2 depends on Task-C in Node1, and it can only be scheduled for execution once it receives the required data from Task-C.

2.3. Synchronization and Communication Code

On shared-memory, threads use synchronization constructs to coordinate access to shared data. *Bulk synchronization* is a common technique used in conjunction with loop parallelism to ensure that all threads exiting it are able to see writes performed by others. For distributed memory, data are shared typically through message passing communication code. Nodes in a distributed-memory cluster are typically shared-memory multicores. Bulk synchronization of threads running on these cores could lead to under-utilization of threads. Dynamically scheduling tasks on threads within each node eliminates bulk synchronization and balances the load among the threads better. It also enables asynchronous point-to-point communication that not only reduces runtime overhead over globally synchronized communication, but also allows overlapping computation with communication.

To dynamically schedule tasks, inter-task dependences are used at runtime. If the task dependence graph is built and maintained in shared memory, then the performance might degrade as the number of tasks increase. So the semantics of the task dependence graph (i.e., all tasks and dependences between tasks) should be maintained without building the graph in memory. In a distributed cluster of nodes, maintaining a consistent semantic view of the task dependence graph across nodes might add significant runtime overhead, thereby degrading performance as the number of tasks increase. To reduce this overhead, each node can maintain its own semantic view of the task dependence graph, and the required communication between nodes can help them to cooperatively maintain their semantics without any centralized coordination.

2.4. Objectives

Our key objectives are as follows:

(1) extraction of coarse-grained dataflow parallelism,
(2) allowing load-balanced execution on shared and distributed-memory architectures,
(3) overlap of computation and communication, and
(4) exposing sufficient functionality that allows the compiler to exploit all of these features automatically including generation of communication code.

For the application of loop transformations and parallelism detection, and subsequent generation of communication sets, we leverage recent techniques developed using the polyhedral framework [Bondhugula 2013a; Dathathri et al. 2013]. The polyhedral framework is used to model computation in sequences of arbitrarily nested loop nests for purposes of analysis, transformation, and code generation. If the accesses are affine functions of surrounding loop iterators and program parameters, they are known as affine accesses. Affine loop nests are loop nests with affine accesses and loop bounds. A statement's domain is represented by a polyhedron with its dimensions corresponding to loops surrounding the statement. The set of integer points in it is the execution domain or the index set of the statement. For affine loop nests, the dependences can be represented by dependence polyhedra. A dependence polyhedron is a relation between the source iterators and target iterators (from same or different statements) that are in dependence. Both the statement's domain and dependence polyhedra are characterized by a set of linear inequalities and equalities.

3. COMPILER-ASSISTED DATAFLOW RUNTIME

This section describes the design issues, our choices, and the details of our solution.

3.1. Overview

A task is a portion of computation that operates on a smaller portion of data than the entire iteration space. Tasks exhibit better data locality, and those that do not depend on one another can be executed in parallel. With compiler assistance, tasks can be automatically extracted from affine loop nests with precise dependence information. Given a distributed-memory cluster of multicores, a task is executed atomically by a thread on a core of a node. A single task's execution itself is sequential with synchronization or communication performed only before and after its execution but not during it. Our aim is to design a distributed decentralized dataflow runtime that dynamically schedules tasks on each node effectively.

Each node runs its own scheduler without centralized coordination. Figure 2 depicts the scheduler on each node. Each node maintains a status for each task and a queue for the tasks that are ready to be scheduled for execution. There are multiple threads on each node, all of which can access and update these data structures. Each thread maintains its own pool of buffers that are reused for communication. It adds more buffers to this pool if all the buffers are busy in communication.

A single dedicated thread on each node receives data from other nodes. The rest of the threads on each node compute tasks that are ready to be scheduled for execution. The computation can update data variables in the local shared memory. After computing a task, for each node that requires some data produced by this task, the thread packs the data from the local shared memory to a buffer from its pool that is not being used, and asynchronously sends this buffer to the node that requires it. After packing the data, it updates the status of the tasks that are dependent on the task that completed execution. The receiver thread preemptively posts anonymous asynchronous receives using all the buffers in its pool and continuously checks for new completion messages. Once it receives the data from another node, it unpacks the data from the buffer to the local shared memory. After unpacking the data, it preemptively posts another anonymous asynchronous receive using the same buffer and updates the status of the tasks that are dependent on the task that sent the data. When the status of a task is updated, it is added to the queue if it is ready to be scheduled for execution.

Each compute thread fetches a task from the task queue and executes it. While updating the status of tasks, each thread could add a task to the task queue. A concurrent task queue is used so the threads do not wait for each other (lock-free). Such dynamic scheduling of tasks by each compute thread on a node balances the load shared by the

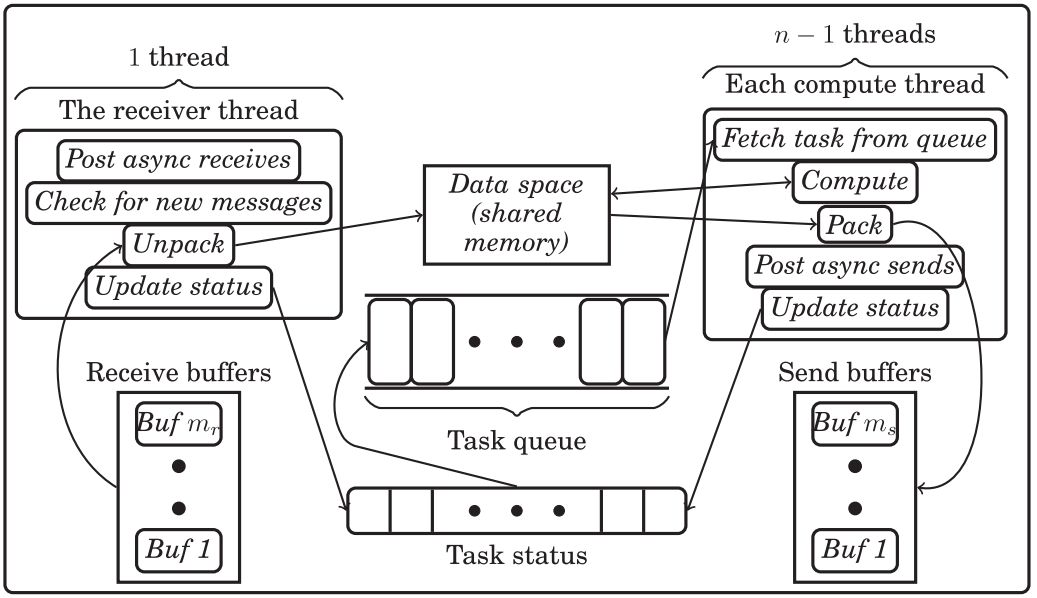


Fig. 2. Overview of the scheduler on each node.

threads better than a static schedule and improves resource utilization [Buttari et al. 2009; Baskaran et al. 2009; Chandramowlishwaran et al. 2010; Theobald 1999]. In addition, each compute thread uses asynchronous point-to-point communication and does not wait for its completion. After posting the non-blocking send messages, the thread progresses to execute another task from the task queue (if it is available) while some communication may still be in progress. In this way, the communication is automatically overlapped with computation, thereby reducing the overall communication cost.

Each node asynchronously sends data without waiting for confirmation from the receiver. Each node receives data without prior coordination with the sender. There is no coordination between the nodes for sending or receiving data. The only messages between the nodes is that of the data that is required to be communicated to preserve program semantics. These communication messages are embedded with metadata about the task sending the data. The metadata are used to update the status of dependent tasks and schedule them for execution. The schedulers on different nodes use the metadata to cooperate with each other. In this way, the runtime is designed for cooperation without coordination.

3.2. Synthesized Runtime Interface (SRI)

The status of tasks are updated based on dependences between them. A task can be scheduled for execution only if all its dependent tasks have finished execution. Since building and maintaining the task dependence graph in memory could have excessive runtime overhead, our aim is to encapsulate the semantics of the task dependence graph to yield minimal runtime overhead. To achieve this, we rely on the observation that, for affine loop nests, the incoming or outgoing edges of a task in a task dependence graph can be captured as a function (code) of that task using dependence analysis. In other words, the semantics of the task dependence graph can be encapsulated at compile time in functions parametric on a task. These functions are called at runtime to dynamically schedule the tasks. The set of parameterized task functions (PTFs)

generated for a program form the Synthesized Runtime Interface (SRI) for that program. We now define the SRI that is required and show that it can be generated using static analysis techniques.

A task is an iteration of the *innermost parallelized loop* that should be executed atomically. The *innermost parallelized loop* is the innermost among loops that have been identified for parallelization, and we will use this term in the rest of this section. The loops that surround the *innermost parallelized loop* may or may not be parallel. A task is uniquely identified using the iteration vector of the innermost parallelized loop, that is, the tuple task_id of integer iterator values ordered from the outermost iterator to the innermost iterator. In addition to task_id, some of the PTFs are parameterized on a data variable and a node. A data variable is uniquely identified by an integer data_id, which is its index position in the symbol table. A node is uniquely identifiable by an integer node_id, which is typically the rank of the node in the global communicator.

Note that there are alternatives with respect to the choice of the loop depth that defines the task. As per the previous paragraph, the innermost among parallelized loops was chosen as the one demarcating a task from loops that surround it. However, the framework we describe will work for any choice of loop depth that defines a task. A tradeoff clearly exists in its choice. Considering a loop nested deeper and carrying a completely serializing dependence will yield no additional parallelism and incur additional synchronization overhead due to smaller tasks. On the other hand, considering a partially or fully parallel loop at a deeper level will increase parallelism and provide better performance if the reduction in granularity of tasks does not hurt synchronization overhead and locality. We consider the choice of the *innermost parallelized loop* as the one defining tasks as reasonable, especially in conjunction with tiling. With tiling and multi-dimensional parallelization, the *innermost parallelized loop* will be the innermost tile space loop, since we do not parallelize loops that traverse iterations within a tile. The tile size chosen controls the task granularity in order to obtain sufficient parallelism and locality while reducing synchronization overhead. Note that any transformations that alter the schedule of loops surrounding the task do not affect the outcome of dynamic scheduling. There may be loops carrying dependences surrounding the *innermost parallelized loop*, and dynamic scheduling will enable extraction of parallelism from across tasks generated from all iterations of the loop nest surrounding the task and from across tasks pertaining to different statements as well.

The PTFs can access and update data structures that are local to the node and are shared by the threads within the node. The PTFs we define can access and update the following locally shared data structures:

(1) **readyQueue** (task queue): a priority queue containing task_id of tasks that are ready to be scheduled for execution.
(2) **numTasksToWait** (task status): a hash map from task_id of a task to a state or counter, indicating the number of tasks that the task has to wait before it is ready to be scheduled for execution.

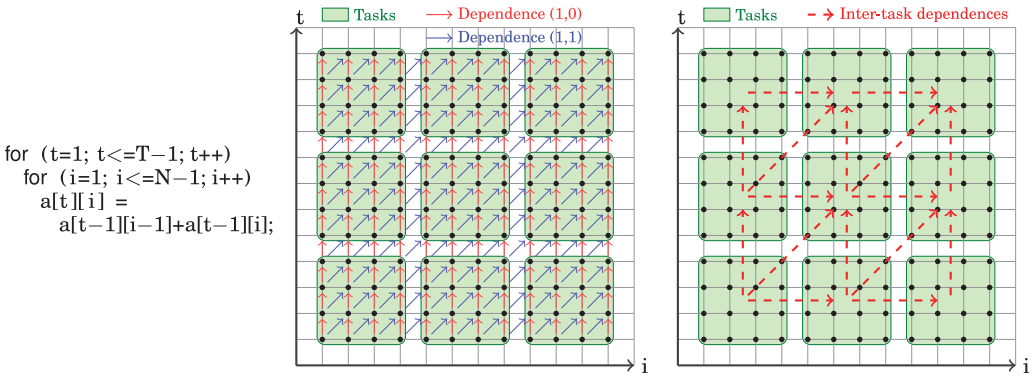
The PTFs do not coordinate with other nodes to maintain these data structures, since maintaining a consistent view of data structures across nodes might add significant runtime overhead. So all operations within a PTF are local and non-blocking.

The name, arguments, and operation of the PTFs in the SRI are listed in Table I. The PTFs are categorized into those that assist scheduling, communication, placement, and computation.

Inter-task dependences: Baskaran et al. [2009] describe a way to extract inter-tile dependences from data dependences between statements in the transformed iteration space. Inter-task dependences can be extracted in a similar way. Figure 3 illustrates

Table I. Synthesized Runtime Interface (SRI)

Function call	Category	Operation
incrementForLocalDependent(task_id)	Scheduling	Increment numTasksToWait of the task task_id for each local task that it is dependent on
incrementForRemoteDependent(task_id)	Scheduling	Increment numTasksToWait of the task task_id for each remote task that it is dependent on
decrementDependentOfLocal(task_id)	Scheduling	Decrement numTasksToWait of the tasks that are dependent on the local task task_id
decrementDependentOfRemote(task_id)	Scheduling	Decrement numTasksToWait of the local tasks that are dependent on the remote task task_id
countLocalDependent(task_id)	Scheduling	Returns the number of local tasks that are dependent on the task task_id
countRemoteDependent(task_id)	Scheduling	Returns the number of remote tasks that are dependent on the task task_id
isReceiver(node_id,data_id,task_id)	Communication	Returns true if the node node_id is a receiver of elements of data variable data_id from the task task_id
pack(data_id,task_id, node_id, buffer)	Communication	Packs elements of data variable data_id from local shared-memory into the buffer, that should be communicated from the task task_id to the node node_id
unpack(data_id,task_id, node_id, buffer)	Communication	Unpacks elements of data variable data_id to local shared-memory from the buffer, that has been communicated from the task task_id to the node node_id
pi(task_id)	Placement	Returns the node node_id on which the task task_id will be executed
compute(task_id)	Computation	Executes the computation of the task task_id



(a) Original code (b) Dependencies between iterations (c) Dependencies between tasks

Fig. 3. Illustration of inter-task dependences for an example.

the inter-task dependences for an example. Recall that a task is an iteration of the innermost parallelized loop. For each data dependence polyhedron in the transformed iteration space, all dimensions inner to the innermost parallelized loop in the source domain and the target domain are projected out to yield an *inter-task dependence polyhedron* corresponding to that data dependence. As noted in Section 2.2, it is

sufficient to have only one inter-task dependence between two tasks for all data dependences between them. Therefore, a union of all inter-task dependence polyhedra corresponding to data dependences is taken to yield the inter-task dependence polyhedron.

Note that a single task can be associated with multiple statements in the polyhedral representation. In particular, all statements inside the innermost parallelized loop characterizing the task are the ones associated with the task. A task can also be created for a statement with no surrounding parallel loops, but is part of a sequence of loop nests with parallel loops elsewhere.

We now introduce notation corresponding to background presented on the polyhedral framework in Section 2. Let $S_1, S_2, \ldots, S_m$ be the statements in the polyhedral representation of the program, m_S be the dimensionality of statement S, d_i and d_j be the depths of the innermost parallelized loops corresponding to tasks T_i and T_j respectively, $s(T)$ be the set of polyhedral statements in task T, and D_e be the dependence polyhedron for a dependence edge $e \in E$ between S_p and S_q. Let project_out(P, i, n) be the polyhedral library routine that projects out n dimensions from polyhedron P starting from dimension number i (0-indexed). Then, the inter-task dependence polyhedron for tasks T_i and T_j is computed as follows:

$$\begin{aligned} D'_e &= \text{project\_out}(D_e, m_{S_p} + d_j, m_{S_q} - d_j) \\ D_e^T &= \text{project\_out}\big(D'_e, d_i, m_{S_p} - d_i\big) \\ D^T(T_i \rightarrow T_j) &= \bigcup_{\{e \mid e=(S_p,S_q),\ S_p \in T_i,\ S_q \in T_j\}} \big(\langle \vec{s}, \vec{t} \rangle \in D_e^T\big). \end{aligned} \tag{1}$$

In the above, D_e^T captures dependence instances between S_p and S_q only on the dimensions surrounding the respective tasks, more simply the inter-task dependence instances. Equation (1) thus provides dependences between task instances of T_i and T_j, with the union being performed for edges from any statement in T_i to any statement in T_j.

The inter-task dependence polyhedron is a key compile-time structure. All PTFs that assist in scheduling rely on it. A code generator such as Cloog [Bastoul 2013] is used to generate code iterating over certain dimensions of $D^T(T_i \rightarrow T_j)$ while treating a certain number of outer ones as parameters. For example, if the target tasks need to be iterated over for a given source task, we treat the outer d_i dimensions in D^T as parameters and generate code scanning the next d_j dimensions. If the source tasks are to be iterated over given a target task, then the dimensions are permuted before a similar step is performed.

Constraints on scheduling: As illustrated in the example in Section 2.2, memory-based data dependences, that is, WAR and WAW dependences, do not enforce a constraint on the order of execution of tasks on different nodes since those tasks will not share an address space at execution time. So the inter-task dependence polyhedron between tasks placed on different nodes is extracted using RAW dependence polyhedra alone. On the other hand, memory-based data dependences do enforce a constraint on the order of execution of tasks on the same node. So the inter-task dependence polyhedron between tasks on the same node is extracted using RAW, WAR, and WAW dependence polyhedra. For a PTF that traverses the incoming edges of a task, the target task in the inter-task dependence polyhedron is treated as a parameter, and code is generated to enumerate the source tasks. For a PTF that traverses the outgoing edges of a task, the source task in the inter-task dependence polyhedron is treated as a parameter, and code is generated to enumerate the target tasks. Each PTF can check if the enumerated task is local or remote (using the placement PTF) and then

Table II. Synthesized Runtime Interface (SRI) That Assists Dynamic Scheduling: Generated by Analyzing Inter-Task Dependences (decrementDependentOfRemote() Should Be Called for Remote Tasks While the Rest Should Be Called for Local Tasks)

Type of dependence	Parameterized on	Iterates over	Condition on enumerated task	Conditional action
RAW, WAR or WAW	target task	source tasks	local task	numTasksToWait[target_task_id]++

(a) incrementForLocalDependent

Type of dependence	Parameterized on	Iterates over	Condition on enumerated task	Conditional action
RAW	target task	source tasks	remote task	numTasksToWait[target_task_id]++

(b) incrementForRemoteDependent

Type of dependence	Parameterized on	Iterates over	Condition on enumerated task	Conditional action
RAW, WAR or WAW	source task	target tasks	none	numTasksToWait[target_task_id]−− If target task is local AND numTasksToWait[target_task_id] == 0: readyQueue.push(target_task_id)

(c) decrementDependentOfLocal

Type of dependence	Parameterized on	Iterates over	Condition on enumerated task	Conditional action
RAW	source task	target tasks	local task	numTasksToWait[target_task_id]−− If numTasksToWait[target_task_id] == 0: readyQueue.push(target_task_id)

(d) decrementDependentOfRemote

Type of dependence	Parameterized on	Iterates over	Condition on enumerated task	Conditional action
RAW, WAR or WAW	source task	target tasks	local task	return_count++

(e) countLocalDependent

Type of dependence	Parameterized on	Iterates over	Condition on enumerated task	Conditional action
RAW	source task	target tasks	remote task	return_count++

(f) countRemoteDependent

perform an action dependent on that. Table II summarizes this for each PTF that assists scheduling.

Communication and placement: Dathathri et al. [2013] generate data movement code for distributed-memory architectures by parameterizing communication on an iteration of the innermost parallelized loop. Since the data to be communicated could be discontiguous in memory, the sender packs it into a buffer before sending it, and the receiver unpacks it from the buffer after receiving it. We adapt the same techniques (*flow-out partitioning*) to parameterize communication on a task. A PTF is generated to pack elements of a data variable written in a task from local shared memory into a buffer that should be communicated to a node. Similarly, a PTF is generated to unpack elements of a data variable written in a task to local shared-memory from a buffer that has been communicated to a node. In addition, a PTF is generated to determine if a node is a receiver of the elements of a data variable written in a task. The *pi* function (Table I) provides the placement of tasks. Section 3.3 will discuss when the placement can be determined and specified (in essence, our framework supports determining the placement of all tasks at runtime before any task has been executed).

ALGORITHM 1: Distributed Function-Based Dynamic Scheduling (DFDS)

```
1 ⟨numTasksToCompute, numTasksToReceive⟩ ←initTasks()
2 begin parallel region
3   if thread_id == 0 then
      // single dedicated receiver thread
4
5     receiveDataFromTasks(numTasksToReceive)
    // compute threads
6   computeTasks(numTasksToCompute)
```

Computation: We enumerate all tasks and extract computation for a parameterized task using techniques described by Baskaran et al. [2009]. For each innermost parallelized loop in the transformed iteration space, from the iteration domain of a statement within the loop, all dimensions inner to the innermost parallelized loop are projected out. The code generated to traverse this domain will enumerate all tasks in that distributed loop at runtime. To extract the computation PTF, the iteration domain of all statements within the innermost parallelized loop is considered. All outer dimensions up to and including the innermost parallelized loop are treated as parameters, and code is generated to traverse dimensions inner to the innermost parallelized loop.

Thread-safety: A concurrent priority queue is used as the readyQueue. Atomic increments and decrements are used on the elements of numTasksToWait. *unpack* is the only PTF that modifies original data variables in local shared-memory. So the runtime has to ensure that the unpack PTF of a (remote) task is not called while a local task that the task depends on or a local task that depends on the task (through flow, anti, or output dependence) is being executed. As long as the unpack PTF respects the local inter-task dependence constraints, all PTFs can be simultaneously called with different parameters by different threads in a node without affecting program semantics.

3.3. Distributed Function-Based Dynamic Scheduling (DFDS)

Compiler assistance or hints can make a runtime more efficient by reducing runtime overhead. A runtime that a compiler can automatically generate code for is even more useful since efficient parallel code is directly obtained from sequential code, thereby eliminating programmer burden in parallelization. As mentioned earlier, our goal is to build a runtime that is designed to be targeted by a compiler. In particular, we design a distributed decentralized runtime that uses the SRI generated by a compiler to dynamically schedule tasks on each node. Hence, we call this runtime Distributed Function-Based Dynamic Scheduling (DFDS). Algorithm 1 shows the high-level code generated for DFDS that is executed by each node. Initially, each node initializes the status of all tasks. It also determines the number of tasks it has to compute and the number of tasks it has to receive from. After initialization, a single dedicated thread receives data from tasks executed on other nodes, while the rest of the threads compute tasks that are assigned to this node and these could send data to other nodes.

Algorithm 2 shows the code generated to initialize the status of tasks. For each local task, its numTasksToWait is initialized to the sum of the number of local and remote tasks that it is dependent on. If a local task has no tasks that it is dependent on, then it is added to the readyQueue. For each remote task, a counter numReceivesToWait is determined, which indicates the number of data variables that this node should receive from that remote task. If any data are going to be received from a remote task, then its numTasksToWait is initialized to the number of local tasks that it is dependent

ALGORITHM 2: initTasks()

```
1  my_node_id ← node_id of this node
2  numTasksToCompute ← 0
3  numTasksToReceive ← 0
4  foreach task_id do
5      if pi(task_id) == my_node_id then // local task
6          numTasksToCompute++
7          incrementForLocalDependent(task_id)
8          incrementForRemoteDependent(task_id)
9          if numTasksToWait[task_id] == 0 then
10             readyQueue.push(task_id)
11     else // remote task
12         numReceivesToWait[task_id] ← 0
13         foreach data_id do
14             if isReceiver(my_node_id,data_id,task_id) then
15                 numReceivesToWait[task_id]++
16         if numReceivesToWait[task_id] > 0 then
17             numTasksToReceive++
18             incrementForLocalDependent(task_id)
   Result: ⟨numTasksToCompute, numTasksToReceive⟩
```

ALGORITHM 3: computeTasks()

```
   Data: numTasksToCompute
1  while numTasksToCompute > 0 do
2      (pop_succeeded, task_id) ← readyQueue.try_pop()
3      if pop_succeeded then
4          compute(task_id)
5          sendDataOfTask(task_id)
6          decrementDependentOfLocal(task_id)
7          atomic numTasksToCompute−−
```

on. This is required since the unpack PTF cannot be called on a remote task until all the local tasks it depends on (through flow, anti, or output dependences) have completed. Note that the *for-each* task loop can be parallelized with numTasksToCompute and numTasksToReceive as reduction variables and atomic increments to elements of numReceivesToWait.

Algorithms 3 and 4 show the generated code that is executed by a compute thread. A task is fetched from the readyQueue and its computation is executed. Then, for each data variable and receiver, the data that have to be communicated to that receiver are packed from local shared-memory into a buffer that is not in use. If all the buffers in the pool are being used, then a new buffer is created and added to the pool. The task_id of this task is added as metadata to the buffer. The buffer is then sent asynchronously to the receiver, without waiting for confirmation from the receiver. Note that the pack PTF and the asynchronous send will not be called if all the tasks dependent on this task due to RAW dependences will be executed on the same node. A local task is considered to be complete from this node's point of- view only after the data it has to communicate are copied into a separate buffer. Once a local task has completed, numTasksToWait of its dependent tasks is decremented. This is repeated until there are no more tasks to compute.

ALGORITHM 4: sendDataOfTask()

```
   Data: task_id
1  my_node_id ← node_id of this node
2  foreach data_id do
3      foreach node_id ≠ my_node_id do
4          if isReceiver(node_id,data_id,task_id) then
5              Let i be the index of a send_buffer that is not in use
6              Put task_id to send_buffer[i]
7              pack(data_id, task_id, node_id, send_buffer[i])
8              Post asynchronous send from send_buffer[i] to node_id
```

ALGORITHM 5: receiveDataFromTasks()

```
   Data: numTasksToReceive
1  my_node_id ← node_id of this node
2  foreach data_id and index i of receive_buffer do
3      Post asynchronous receive to receive_buffer[i] with any node_id as source
4  while numTasksToReceive > 0 do
5      foreach data_id and index i of receive_buffer do
6          if asynchronous receive to receive_buffer[i] has completed then
7              Extract task_id from receive_buffer[i]
8              if numTasksToWait[task_id] == 0 then
9                  unpack(data_id,task_id, my_node_id, receive_buffer[i])
10                 numReceivesToWait[task_id]−−
11                 if numReceivesToWait[task_id] == 0 then
12                     decrementDependentOfRemote(task_id)
13                     numTasksToReceive−−
14             Post asynchronous receive to receive_buffer[i] with any node_id as source
```

Algorithm 5 shows the generated code that is executed by the receiver thread. Initially, for each data variable, an asynchronous receive from any node (anonymous) is preemptively posted to each buffer for the maximum number of elements that can be received from any task. Reasonably tight upper bounds on the required size of buffers are determined from the communication set constraints, which are all affine, and thus amenable to static analysis. This is used to determine the maximum number of elements that can be received from any task.

Each receive is checked for completion. If the receive has completed, then the metadata task_id is fetched from the buffer. If all the local tasks that task_id depends on (through flow, anti, or output dependences) have completed, then the data that have been received from the task task_id is unpacked from the buffer into local shared-memory,[1] and numReceivesToWait of task_id is decremented. A data variable from a task is considered to be received only if the data has been updated in local shared

[1]Tasks need not be unpacked in the order in which they are received because they may have to wait until the local tasks they depend on have completed. In this way, the unpack PTF of a (remote) task respects the dependence constraints between local tasks and the task. On the other hand, the unpack PTF of a (remote) task ignores the dependence constraints between the task and other remote tasks. If there are multiple writes to an element in different remote tasks, then only the *last write* before the read in a local task is communicated (using non-transitive flow dependences). So the order of unpacking (remote) tasks is inconsequential.

memory, that is, only if the data has been unpacked. Once the data have been unpacked from a buffer, an asynchronous receive from any node (anonymous) is preemptively posted to the same buffer. A remote task is considered to be complete from this node's point of view only if it has received all the data variables it needs from that task. Once a remote task has completed, numTasksToWait of its dependent tasks is decremented. If all the receive buffers have received data, but have not yet been unpacked, then more buffers are created and an asynchronous receive from any node (anonymous) is preemptively posted to each new buffer. This is repeated until there are no more tasks to receive from.

While evaluating our runtime, we observed that a dedicated receiver thread is under-utilized since almost all its time is spent in busy-waiting for one of the non-blocking receives to complete. Hence, we believe that a single receiver thread is sufficient to manage any amount of communication. To avoid under-utilization, the generated code was modified such that the receiver thread also executed computation (and its associated functions) instead of busy-waiting. We observed that there was almost no difference in performance between a dedicated receiver thread and a receiver thread that also computed. There is a tradeoff: Although a dedicated receiver thread is under-utilized, it is more responsive since it can unpack data (and enable other tasks) soon after a receive. The choice might depend on the application. Our tool can generate code for both such that it can be chosen at compile time. The algorithms are presented as is for clarity of exposition.

Priority: Priority on tasks can improve performance by enabling the priority queue to choose between many ready tasks more efficiently. There are plenty of heuristics to decide the priority of tasks to be executed. Though this is not the focus of our work, we use PTFs to assist in deciding the priority. A task with more remote tasks dependent on it (countRemoteDependent()) has higher priority since data written in it are required to be communicated to more remote tasks. This helps initiate communication as early as possible, increasing its overlap with computation. For tasks with the same number of remote tasks dependent on it, the task with more local tasks dependent on it (countLocalDependent()) has higher priority since it could enable more tasks to be ready for execution. We plan to explore more sophisticated priority schemes in the future.

Dynamic *a priori* placement: When the status of the tasks are being initialized at runtime, DFDS expects the placement of all tasks to be known, since its behavior depends on whether a task is local or remote. The placement of all tasks can be decided at runtime before initializing the status of tasks. In such a case, a hash map from a task to the node that will execute the task should be set consistently across all nodes before the call to initTasks() in line 1 of Algorithm 1. The placement PTF would then read the hash map. DFDS is thus designed to support dynamic *a priori* placement. To find the optimal placement automatically is not the focus of this work. In our evaluation, we use a block placement function except in cases where non-rectangular iteration spaces are involved; in such cases, we use a block-cyclic placement. This placement strategy yields good strong scaling on distributed-memory for the benchmarks we have evaluated, as we will see in Section 5.2. Determining more sophisticated placements including dynamic *a priori* placements is orthogonal to our work. Recent work by Reddy and Bondhugula [2014] explores this independent problem.

4. IMPLEMENTATION

We implement our compiler-assisted runtime as part of a publicly available source-to-source polyhedral tool chain. Clan [Bastoul 2012], ISL [Verdoolaege 2014], Pluto [Bondhugula 2013b], and Cloog-isl [Bastoul 2004] are used for polyhedral extraction,

dependence testing, automatic transformation, and code generation, respectively. Polylib [2010] is used to implement the polyhedral operations in Section 3.2.

The input to our compiler-assisted runtime is sequential code containing arbitrarily nested affine loop nests. The sequential code is tiled and parallelized using the Pluto algorithm [Bondhugula et al. 2008; Bandishti et al. 2012]. Loop tiling helps reduce the runtime overhead and improve data locality by increasing the granularity of tasks. The tiling transformation, that is, the shape of tiles, dictates the decomposition of tasks and the tile size controls the granularity of tasks. The SRI is automatically generated using the parallelized code as input. The DFDS code for either shared-memory or distributed-memory systems is then automatically generated. The code generated can be executed either on a shared-memory multicore or on a distributed-memory cluster of multicores. Thus, ours is a fully automatic source transformer of sequential code that targets a compiler-assisted dataflow runtime.

The concurrent priority queue in Intel Thread Building Blocks [Intel TBB 2014] is used to maintain the tasks that are ready to execute. Parametric bounds of each dimension in the task_id tuple are determined, and these, at runtime, yield bounds for each of the outer dimensions that were treated as parameters. A multi-dimensional array of dimensionality equal to that of the task_id tuple is allocated at runtime. The extent of each dimension of this array corresponds to the difference of the upper and lower bounds of the corresponding dimension in the task_id tuple. This is used to maintain the task statuses numTasksToWait and numReceivesToWait as arrays instead of hash maps. The status of a task_id can then be accessed by offsetting each dimension in the array by the lower bound of the corresponding dimension in the task_id tuple. Thus, the memory required to store the task status is minimized, while accessing it is efficient. Asynchronous non-blocking MPI primitives are used to communicate between nodes in the distributed-memory system.

5. EXPERIMENTAL EVALUATION

Benchmarks: We present results for Floyd-Warshall (`floyd`), LU Decomposition (`lu`), Cholesky Factorization (`cholesky`), Alternating Direction Implicit solver (`adi`), 2d Finite Different Time Domain Kernel (`fdtd-2d`), Heat 2d equation (`heat-2d`), and Heat 3d equation (`heat-3d`) benchmarks. The first five are from the publicly available Polybench/C 3.2 suite [Polybench 2012]; `heat-2d` and `heat-3d` are widely used stencil computations [Tang et al. 2011]. All benchmarks use double-precision floating-point operations. The compiler used for all experiments is ICC 13.0.1 with options "-O3 -ansi-alias -ipo -fp-model precise." These benchmarks were selected from a larger set since (a) their parallelization involves communication and synchronization that cannot be avoided, and (b) they capture different kinds of communication patterns that result from uniform and non-uniform dependences, including near-neighbor, multicast, and broadcast style communication. Table III lists the problem sizes and tile sizes used. We found the variation in performance over multiple runs negligible—the standard deviation over at least three runs was less than 1.8% of the mean in all cases. The results presented hereafter are for a single execution of a benchmark.

Intel CnC implementations: To compare our automatically generated codes against a manually optimized implementation, we implemented `heat-2d`, `heat-3d`, `fdtd-2d`, and `lu` using Intel Concurrent Collections [Intel CnC 2013]. We include `floyd` and `cholesky` from the Intel CnC samples; the `cholesky` benchmark we compare against does not use MKL routines. The Intel CnC version used is 0.9.001 for shared-memory experiments and 0.8.0 for distributed-memory experiments.

The `cholesky` implementation is detailed in a previous performance evaluation of Intel CnC [Chandramowlishwaran et al. 2010]. Our Intel CnC implementations use computation and data tiling for coarsening task granularity and improving locality.

Table III. Problem Sizes and Tile Sizes

	Shared memory			Distributed memory		
		Tile sizes			Tile sizes	
Benchmarks	Problem sizes	auto	manual-CnC	Problem size	auto	manual-CnC
adi	—	—	—	N = 8192, T = 128	256 (2d)	—
heat-2d	N = 8192, T = 1024	64 (3d)	256 (2d)	N = 8192, T = 1024	16 (3d)	256 (2d)
heat-3d	N = 512, T = 256	16 (4d)	64 (3d)	—	—	—
fdtd-2d	N = 4096, T = 1024	16 (3d)	256 (2d)	N = 8192, T = 1024	32 (3d)	256 (2d)
floyd	N = 4096	256 (2d)	256 (2d)	N = 8192	512 (2d)	512 (2d)
cholesky	N = 8192	8 (3d)	128 (3d)	N = 16384	128 (3d)	128 (3d)
lu	N = 8192	64 (3d)	128 (3d)	N = 16384	256 (3d)	128 (3d)

—auto represents auto-DFDS, auto-static, and auto-graph-dynamic;
—data is tiled in manual-CnC but not in auto;
—tiling transformations for auto and manual-CnC may differ;
—auto uses load-balanced (diamond) tiling transformations for heat-2d, heat-3d, and fdtd-2d (stencil codes);
—on distributed memory, auto uses block-cyclic placement for lu and cholesky (non-rectangular iteration spaces) and block placement for the rest.

Table III shows the tile sizes chosen for each benchmark. In case of distributed memory, we also ensure that communication is precise, that is, we only communicate that which is necessary to preserve program semantics. Tiling and precise data communication constitute most of the programming effort. Additionally, we specify the nodes that consume the data produced in a task (the consumed_on() tuner), which helps the runtime to push the data to be communicated to the nodes that require it. We observe that the push model performs much better than the default pull model (pulling data when it is required) in our context. We also provide the exact number of uses for each data buffer so the CnC runtime can efficiently garbage collect it and reduce memory footprint. For each benchmark, we assign higher priority to tasks that communicate to other nodes. Thus, our Intel CnC implementations have been tuned with considerable effort to extract high performance. Note that all these components that are tedious and error prone to write manually are automatically generated by our framework.

5.1. Shared-Memory Architectures

Setup: The experiments were run on a four-socket machine of AMD Opteron 6136 2.4GHz, 128KB L1, 512KB L2, and 6MB L3 cache. The memory architecture is NUMA, and we use numactl to bind threads and pages suitably for all our experiments.

Evaluation: We compare the performance of our automatic approach (auto-DFDS) with:

—hand-optimized Intel CnC codes (manual-CnC),
—state-of-the-art automatic dynamic scheduling approach [Baskaran et al. 2009] that constructs the entire task dependence graph in memory (auto-graph-dynamic), and
—state-of-the-art automatic static scheduling approach [Bondhugula et al. 2008; Bandishti et al. 2012] that uses *bulk-synchronization* (auto-static).

For auto-graph-dynamic, the graph is constructed using Intel Thread Building Blocks [Intel TBB 2014] Flow Graph (TBB is a popular work stealing based library for task parallelism). All the automatic schemes use the same polyhedral compiler transformations (and the same tile sizes). The performance difference in the automatic schemes thus directly relates to the efficiency in their scheduling mechanism.

Analysis: Figure 4 shows the scaling of all approaches relative to the sequential version (seq) that is the input to our compiler. Note that the performance of auto-DFDS and auto-static on a single thread differs from that of seq due to automatic

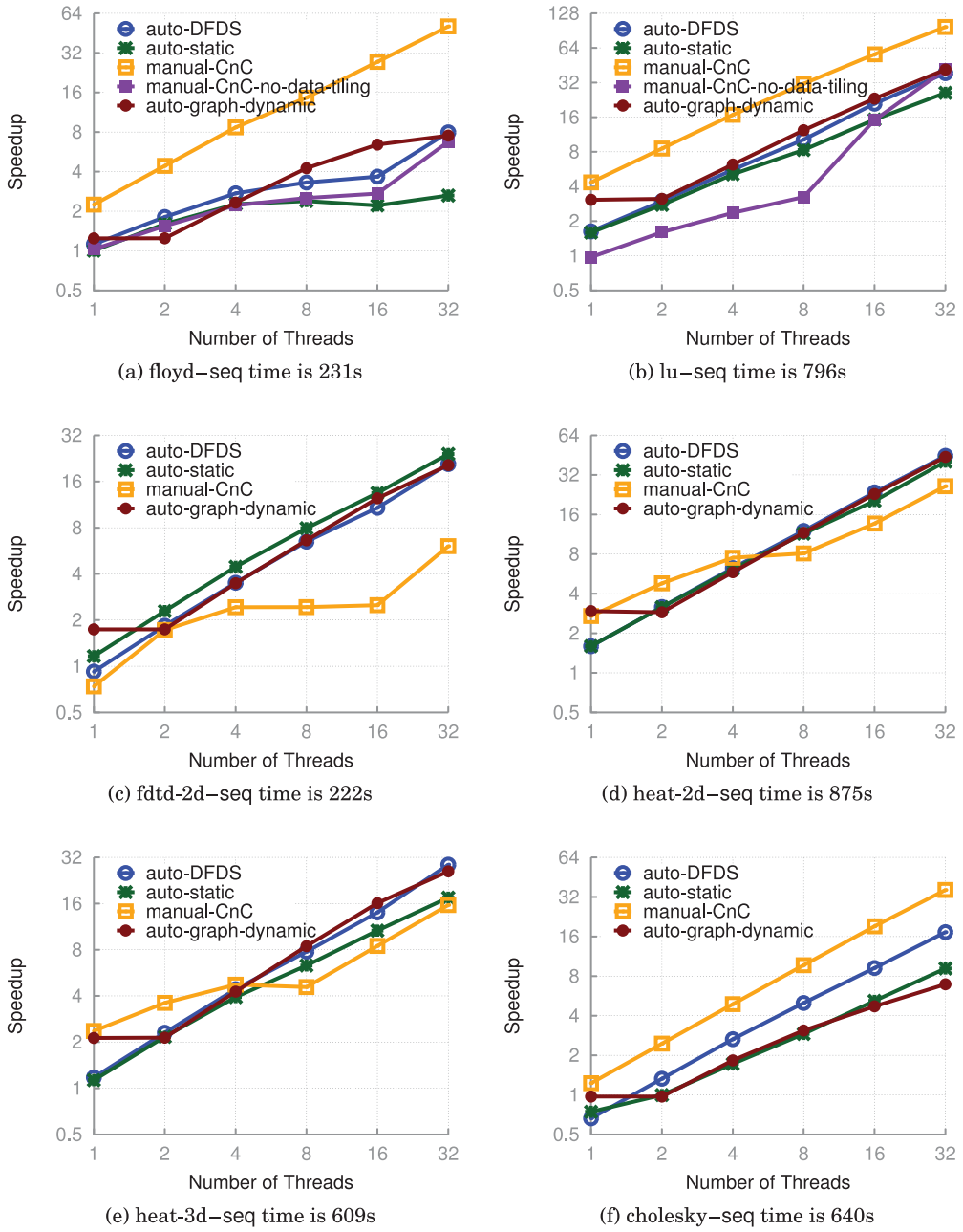


Fig. 4. Speedup of auto-DFDS, auto-static, and manual-CnC over seq on a shared-memory multicore.

transformations for the former. auto-DFDS scales well with an increase in the number of threads and yields a geometric mean speedup of 22.8× (7.9× to 44.7×) on 32 threads over the sequential version. The runtime overhead of auto-DFDS (to create and manage tasks) on 32 threads is less than 1% of the overall execution time for all benchmarks, except `cholesky`, for which it is less than 3%.

Table IV. Standard-Deviation over Mean of Computation Times of All Threads in %: Lower Value Indicates Better Load Balance

	Shared-memory (32 threads)		Distributed-memory (32 nodes)	
Benchmarks	**static**	**DFDS**	**static**	**DFDS**
adi	—	—	2.58	3.12
heat-2d	23.74	0.65	3.67	2.22
heat-3d	46.66	0.09	—	—
fdtd-2d	22.28	1.14	3.52	1.29
lu	37.48	1.36	67.45	16.13
cholesky	57.34	0.39	48.96	8.09
floyd	102.35	0.08	174.78	3.25

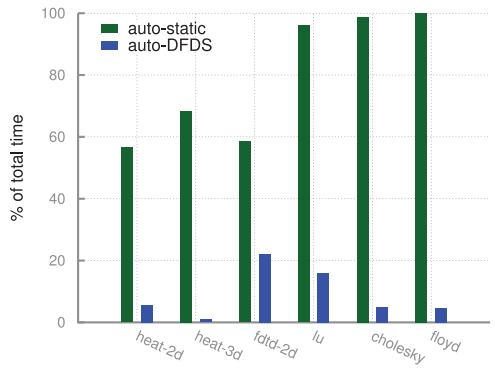


Fig. 5. Maximum idle time across 32 threads on a shared-memory multicore.

auto-DFDS scales better than or comparably to both auto-graph-dynamic and auto-static. For auto-DFDS and auto-static, we measured the computation time of each thread and calculated the mean and standard deviation of these values. Table IV shows the standard deviation divided by mean, which provides a fair measure of the load balance. auto-DFDS balances load much better than auto-static, thereby decreasing the overall execution time. We also measured the maximum idle time across threads for both auto-DFDS and auto-static, which includes the synchronization time. Figure 5 shows that all threads are active for most of the time in auto-DFDS, unlike auto-static.

Figure 6 shows the speedup of auto-DFDS over manual-CnC on both 1 thread and 32 threads. The speedup on 32 threads is as good as or better than that on 1 thread, except for `floyd`. This shows that auto-DFDS scales as well as or better than manual-CnC. In the CnC model, programmers specify tasks along with data they consume and produce. As a result, data are decomposed along with tasks, that is, data are also tiled. For example, a two-dimensional (2D) array when 2D tiled yields a 2D array of pointers to a 2D sub-array (tile) that is contiguous in memory. Such explicit data tiling transformations yield better locality at all levels of memory or cache. Due to this, manual-CnC outperforms auto-DFDS for `floyd`, `lu`, and `cholesky`. manual-CnC also scales better for `floyd` because of privatization of data tiles with increase in the number of threads; privatization allows reuse of data along the outer loop, thereby achieving an effect similar to that of 3D tiling. To evaluate this, we implemented manual-CnC without the data tiling optimizations for both `floyd` and `lu`. Figure 4 validates our hypothesis by showing that manual-CnC-no-data-tiling versions perform similar to auto-DFDS, indicating the need for improved compiler transformations for data tiling. For `fdtd-2d`, `heat-2d`, and `heat-3d`, automatic approaches find load-balanced computation

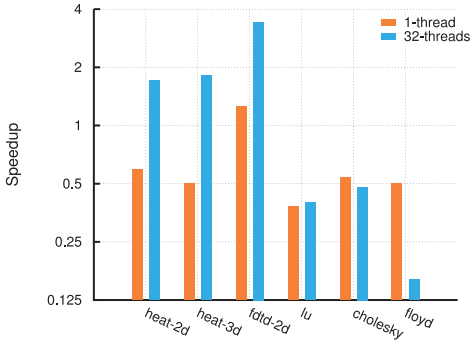


Fig. 6. Speedup of auto-DFDS over manual-CnC–shared-memory.

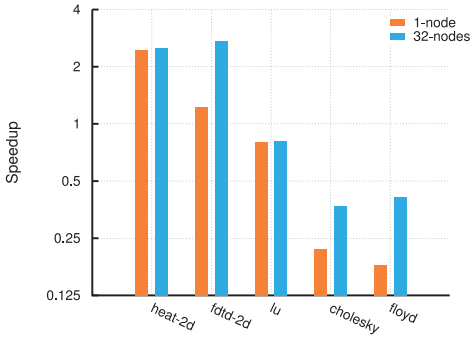


Fig. 7. Speedup of auto-DFDS over manual-CnC–distributed-memory.

tiling transformations [Bandishti et al. 2012] that also tile the outer serial loop. These are hard and error prone to implement manually and almost never done in practice. Consequently, manual-CnC codes only tile the parallel loops and not the outer serial loop. In these cases, auto-DFDS significantly outperforms manual-CnC: This highlights the power of automatic task generation frameworks used in conjunction with runtimes. Storage mapping or automatic data tiling transformations [Yuki and Rajopadhye 2013; Reddy and Bondhugula 2014] can make our approach even more effective and match the performance of manual implementations like CnC.

5.2. Distributed-Memory Architectures

Setup: The experiments were run on a 32-node InfiniBand cluster of dual-SMP Xeon servers. Each node on the cluster consists of two quad-core Intel Xeon E5430 2.66GHz processors with 12MB L2 cache and 16GB RAM. The InfiniBand host adapter is a Mellanox MT25204 (InfiniHost III Lx HCA). All nodes run 64-bit Linux kernel version 2.6.18. The cluster uses MVAPICH2-1.8.1 as the MPI implementation. It provides a point-to-point latency of 3.36μs, unidirectional and bidirectional bandwidths of 1.5GB/s and 2.56GB/s respectively. The MPI runtime used for running CnC samples is Intel MPI as opposed to MVAPICH2-1.8.1, as CnC works only with the Intel MPI runtime.

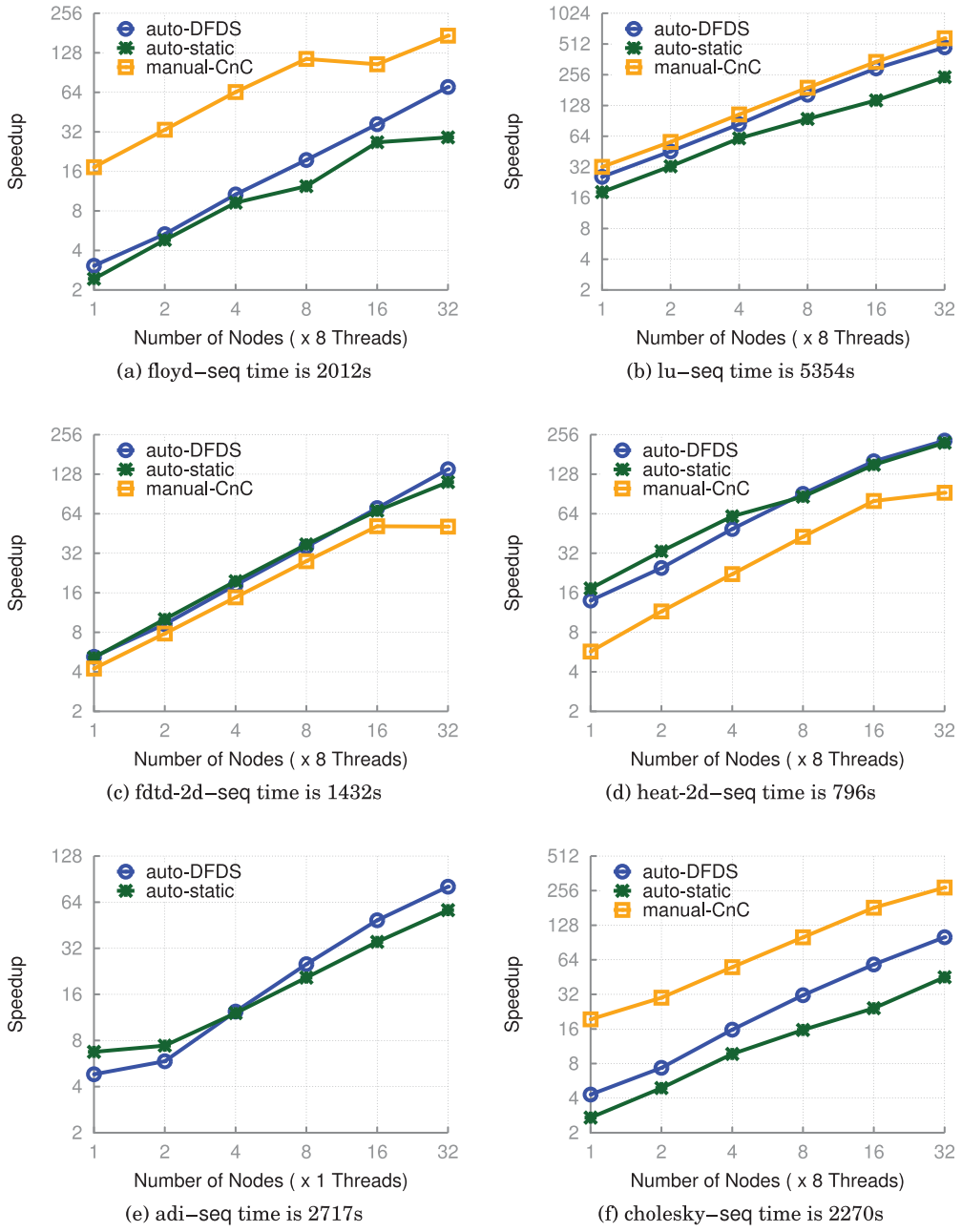
Evaluation: We compare our fully automatic approach (auto-DFDS) with:

—hand-optimized Intel CnC codes (manual-CnC), and
—a recent automatic parallelization approach on distributed-memory [Dathathri et al. 2013] that uses bulk synchronization (auto-static).

All the automatic schemes use the same polyhedral compiler transformations, and the same tile sizes. The performance difference in the automatic schemes thus directly relates to the efficiency in their scheduling mechanism.

Analysis: Figure 8 shows the scaling of all approaches relative to the sequential version (seq) which is the input to our compiler. Note that the performance of auto-DFDS and auto-static on a single thread differs from that of seq due to automatic transformations for the former. auto-DFDS scales well with an increase in the number of nodes and yields a geometric mean speedup of 143.6× (70.3× to 474.7×) on 32 nodes over the sequential version. The runtime overhead of auto-DFDS (to create and manage tasks) on 32 nodes is less than 1% of the overall execution time for all benchmarks.

auto-DFDS yields a geometric mean speedup of 1.64× (1.04× to 2.42×) over auto-static on 32 nodes. For both of them, we measured the computation time of each thread on each



(a) floyd−seq time is 2012s

(b) lu−seq time is 5354s

(c) fdtd-2d−seq time is 1432s

(d) heat-2d−seq time is 796s

(e) adi−seq time is 2717s

(f) cholesky−seq time is 2270s

Fig. 8. Speedup of auto-DFDS, auto-static, and manual-CnC over seq on a cluster of multicores.

node and calculated the mean and standard deviation of these values. Table IV shows the standard deviation divided by mean, which provides a fair measure of the load balance. auto-DFDS achieves good load balance even though the computation across nodes is statically distributed. auto-DFDS balances load much better than auto-static, thereby decreasing the overall execution time.

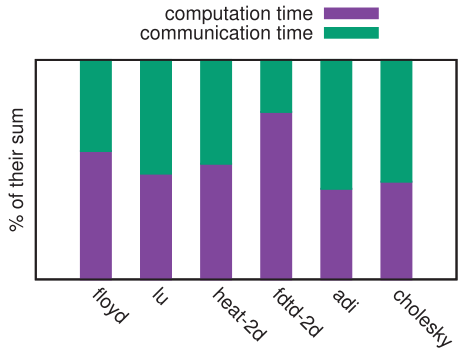


Fig. 9. Maximum computation time and maximum communication time across all threads on 32 nodes in auto-static.

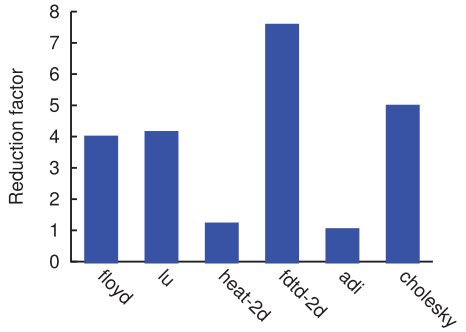


Fig. 10. Non-overlapped communication time reduction: auto-DFDS over auto-static on 32 nodes.

We measured the maximum communication time across all threads in auto-static, and the maximum idle time across all threads in auto-DFDS, which would include the non-overlapped communication time. Figure 9 compares the maximum communication time and the maximum computation time for auto-static on 32 nodes and shows that communication is a major component of the overall execution time. Figure 10 shows the reduction factor in non-overlapped communication time achieved by auto-DFDS on 32 nodes. The graphs show that auto-DFDS outperforms auto-static mainly due to better communication-computation overlap achieved by performing asynchronous point-to-point communication.

Figure 7 shows the speedup of auto-DFDS over manual-CnC on both 1 node and 32 nodes. The speedup on 32 nodes is as good as or better than that on 1 node. This shows that auto-DFDS scales as well as or better than manual-CnC. The performance difference between auto-DFDS and manual-CnC on 32 nodes is due to the performance difference between them on a single node (shared-memory multicore). Hence, as shown in our shared-memory evaluation in Section 5.1, auto-DFDS outperforms manual-CnC when compiler transformations like computation tiling and time skewing [Wonnacott 2000; Song and Li 1999; Bondhugula et al. 2008] are better than manual implementations, and manual-CnC outperforms auto-DFDS in other cases due to orthogonal explicit data tiling transformations.

6. RELATED WORK

This section compares our work with existing automatic parallelization frameworks and other dynamic scheduling dataflow runtime frameworks.

6.1. Automatic Parallelization Frameworks

A large number of distributed-memory code generation algorithms for uniform dependences were explored in the 1990s; we focus our attention on those that handle affine dependences. Previous techniques for automatic parallelization of affine loop nests for distributed-memory architectures [Adve and Mellor-Crummey 1998; Claßen and Griebl 2006; Bondhugula 2013a; Dathathri et al. 2013] perform static scheduling of loops across nodes, either block or block cyclic. They perform communication in a bulk-synchronous manner, that is, after the loop is executed in parallel, communication and synchronization is performed using a "push" approach before the parallel loop is invoked again due to an outer surrounding sequential loop. They do not overlap communication with computation. Our approach uses dynamic scheduling on each node

and communication across nodes is done in an asynchronous fashion, thereby achieving communication-computation overlap. Our evaluation demonstrates that it scales better than bulk-synchronous approaches. Our framework builds on the state-of-the-art automatic distributed-memory parallelization framework [Dathathri et al. 2013] and subsumes it to target a dataflow runtime.

Reddy and Bondhugula [2014] build on the work presented in this article. They use our dynamic scheduling dataflow runtime framework in conjunction with techniques for on-demand data allocation via data tiling and more general computation placement schemes including *sudoku* mappings as well as arbitrary ones. The computation placement schemes used in their work are all dynamic *a priori*, but the choice of the placement function is made at compiler time itself. The placements determined optimize communication volume and maintain load balance. Placement and scheduling are two different aspects: The former determines where iterations will execute while the latter determines when they will execute. Our work enables dynamic scheduling to improve load balance for a given placement. Tasks placed at a node are not moved across nodes but only scheduled dynamically across cores of a node; it is thus complementary to approaches that determine balanced placements across nodes.

Prior works that automatically parallelized code for dynamic scheduling or dataflow runtimes target only shared-memory architectures [Baskaran et al. 2009; Vasilache et al. 2014; Pop and Cohen 2013; Kong et al. 2015]. As a consequence, they only deal with synchronization of tasks unlike ours which also deals with communication of data between tasks. Our framework, unlike past works, has been designed to target distributed memory in addition to shared memory and both in conjunction.

Baskaran et al. [2009] provide techniques for extracting tasks from affine loop nests and constructing the task dependence graph using the polyhedral framework. They construct the entire task dependence graph in memory and then schedule it. Since the task graph is shared and modified across all threads, it becomes a bottleneck when there are a large number of threads. More importantly, their compiler-assisted runtime is limited to shared memory and does not deal with challenges associated with distributed memory. Our work was motivated by theirs, and our goal has been to target a distributed-memory cluster of multicores. In contrast to their technique, our approach does not build the entire task dependence graph in memory but uses compiler generated routines that semantically encapsulate it. We also generate runtime components to manage memory-based dependences and communication across nodes in distributed memory. Our tool generates code that can be executed on shared memory, distributed memory, or a combination of both. Our evaluation shows that the performance of our approach on shared memory is similar to or better than that of their approach. Thus, our framework builds on and subsumes it.

Vasilache et al. [2014] use the polyhedral framework to generate code from affine loop nests that targets CnC [Budimlic et al. 2009], SWARM [Lauderdale and Khan 2012], and Open Community Runtime [OCR 2013], with a goal of comparing these runtimes. Their framework, unlike ours, is applicable only for shared-memory systems. Moreover, their approach conservatively approximates any dependence on permutable loops with distance-one dependences between all adjacent iterations of such loops. Such an over-approximation of dependences restricts scheduling choices, and for loop nests with long dependences, a dynamic scheduling runtime may no longer be able to overcome the imbalance that exists when using bulk-synchronous runtimes (like OpenMP).

Kong et al. [2015] provide techniques for generating OpenStream [Pop and Cohen 2013] code from affine loop nests using the polyhedral framework. Their primary focus was on exploiting temporal reuse across distinct loop nests, that is, dynamically fuse tasks across loop nests by scheduling them to execute one after the other at runtime. Although such reuse can also be exploited in our framework, their main contribution of

selecting the most profitable synchronization idiom (whether to dynamically fuse loop nests or use a soft barrier between loop nests) is orthogonal to our work. In addition, unlike ours, their design and evaluation is for shared-memory systems alone.

6.2. Dataflow Runtime Frameworks

A number of works have focused on building scalable frameworks for a certain class of applications like linear algebra kernels [Bosilca et al. 2010, 2012; Song and Dongarra 2012]. These frameworks are typically driven by a domain-specific language in which the user is expected to express the computation as a DAG of tasks, where the nodes are sequential computation tasks, and the edges are dependences between tasks. Hence, the burden of expressing parallelism and locality is shifted to the user. In contrast, our framework automatically extracts such a DAG of tasks. The Directed Acyclic Graph Engine (DAGuE) [Bosilca et al. 2012] framework does not build or unroll the entire task graph in memory but encapsulates the DAG concisely in memory using a representation conceptually similar to the *Parameterized Task Graph* (PTG) [Cosnard and Loi 1995]. Our techniques to compactly represent the task graph use *Parameterized Task Functions* (PTFs). In contrast to PTG, PTFs encapsulate differences in dependences between tasks on the same node and dependences between tasks on different nodes, thereby allowing handling of memory-based dependences; PTFs also encapsulate precise communication semantics (at the granularity of array elements). Our work has been motivated by these approaches, with the main difference being that we intend to provide a fully automatic solution through compiler support.

In the system designed by Song and Dongarra [2012], the node that will execute a task instance maintains its inputs, output, and the ready status of each input. Each node, therefore, has a partition of the DAG of tasks in memory, using which tasks are scheduled dynamically driven by data availability; the partial task graph is built without any coordination with other nodes. Our framework, on the other hand, does not maintain even a partial task graph in memory but only maintains status on tasks that is built and maintained without any coordination with other nodes. Communication in their system is determined by the dataflow between tasks at the granularity of data tiles, whereas communication in our system is precise at the granularity of array elements. Communication is asynchronous and is overlapped with computation. There is a separate thread each for intra-node and inter-node communication. The inter-node communication thread preemptively posts an anonymous receive and checks whether it has finished with busy-polling. Our communication framework is designed similarly.

There are several recent works that focus on providing high-level programming models or extensions that enable easy expression of parallelism, like CnC [Budimlic et al. 2009], StarPU [Augonnet et al. 2011], the codelet model [Zuckerman et al. 2011; Lauderdale and Khan 2012] (which is inspired by the EARTH system [Theobald 1999; Hendren et al. 1997]), OpenStream [Pop and Cohen 2013], OmpSs [Fernández et al. 2014], and TERAFLUX [Giorgi et al. 2014]. These models decouple scheduling and program specification, which is tightly coupled in current programming models. The notion of a "task" in our work is conceptually similar to that of a "codelet" in the codelet model, a "task" in StarPU, and a "step" in CnC. In these models, the application developer or user specifies the program in terms of tasks along with its inputs and outputs, while the system (or system developer) is responsible for scheduling the tasks efficiently on the parallel architecture. However, the user is not completely isolated from locality aspects of modern architectures. As an example, one of the key issues in leveraging task scheduling runtimes such as CnC is in determining the right decomposition into tasks, that is, the shape of blocks or tiles, and the granularity for the tasks, that is, the block or tile size. The shape of blocks decides available *asynchronous parallelism*, communication, and synchronization costs; a smaller block size increases the degree of available

asynchronous parallelism, thereby improving load balance, but also increases the overhead in synchronization, maintaining tasks, and managing data. Choosing the right decomposition can improve performance by orders of magnitude—this is evident from our experimental results. The decomposition into tasks has a direct connection with loop transformations such as tiling, making a strong case for integration of compiler support.

Many new dynamic scheduling runtimes have emerged in the recent past and are under active development. These include Intel CnC [Budimlic et al. 2009], StarPU [Augonnet et al. 2010, 2011], codelet model runtimes like ETI SWARM [Lauderdale and Khan 2012; Suettlerlein et al. 2013], and Open Community Runtime [OCR 2013]. They share some of the same design objectives as those of our work, like providing load balance and overlapping data movement with computation. Our techniques are sufficiently generic and could possibly be adapted to generate code for these runtimes. However, the thrust of our work is in coupling runtime support with powerful compiler transformation, parallelization, and code generation support to provide a fully automatic solution. This is done so efficient execution on shared as well as distributed memory is achieved with no programmer input.

The choice to develop our own runtime was driven by the need to allow sufficient customization for affine loop nests, as opposed to replacing or competing with existing ones. SWARM [Lauderdale and Khan 2012] is not designed to natively support dependences between remote tasks—such dependences differ from dependences between local tasks. There is also no direct way to make it ignore memory-based dependences at runtime exclusively in the situation that the dependent tasks are scheduled on different nodes. Other existing runtimes are similar. In CnC, there are no memory-based dependences because it adheres to dynamic single assignment. In contrast, our runtime enforces memory-based dependences only for local tasks and ignores it for remote tasks (except to unpack received data). Communication semantics in SWARM and CnC are designed such that tasks can directly operate on data in the buffer communicated. Consequently, if tasks operate on local memory different from communication buffers, it is the programmer's responsibility to copy data. In other words, those runtimes do not intrinsically support packing and unpacking of data from application memory to communication buffer and vice versa, respectively. In our compiler-runtime co-design, data from a remote task are unpacked only once, even if there are several local tasks waiting for the data, and it is only unpacked after all memory-based dependences between that remote task and local tasks are satisfied. It is non-trivial to generate such code targeting runtimes that do not support packing and unpacking intrinsically. On the other hand, our runtime incorporates precise communication semantics by natively supporting packing and unpacking values of array elements to and from buffers.

As our evaluation demonstrated, the performance benefits of using a dataflow runtime are not only due to avoiding bulk synchronization but also due to overlapping computation with communication. There have been plenty of works that overlap computation with communication within a bulk-synchronous schedule to reduce effective communication time [Marjanović et al. 2010; Nguyen et al. 2012; Bao et al. 2012]. The main disadvantage with such works is that the amount of computation that could be overlapped with communication is restricted due to the bulk-synchronous schedule, whereas in our framework, it is limited only by the dataflow dependences.

7. CONCLUSIONS

We described the design and implementation of a new dataflow runtime system for modern parallel architectures that is suitable for target by compilers capable of extracting tasks and dependence information between tasks. We coupled the runtime with a source-to-source polyhedral optimizer to enable fully automatic dynamic scheduling

on a distributed-memory cluster of multicores. The design of the runtime allows load-balanced execution on shared and distributed memory cores, handling of memory-based dependences, and asynchronous point-to-point communication. The resulting system is also the first automatic parallelizer that uses dynamic scheduling for affine loop nests on distributed-memory. On 32 nodes with 8 threads per node, our compiler-assisted runtime yields a geometric mean speedup of 1.64× (1.04× to 2.42×) over the state-of-the-art automatic approach and a geometric mean speedup of 143.6× (70.3× to 474.7×) over the sequential version. On a shared-memory system with 32 cores, our runtime yields a speedup of up to 2.5× over the state-of-the-art dynamic scheduling approach and a geometric mean speedup of 22.8× (7.9× to 44.7×) over the sequential version. Our automatic framework also significantly outperformed hand-optimized Intel CnC codes in some cases due to benefit from complex compiler transformations. We believe our approach is a significant advance to compiler/runtime technology necessary to leverage dynamic scheduling and task-level data flow parallelism automatically on both shared and distributed memory. The detailed experimental evaluation presented offers compelling evidence and quantifies the exact benefits of such an approach.

REFERENCES

Vikram Adve and John Mellor-Crummey. 1998. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, NY, 186–198. DOI: http://dx.doi.org/10.1145/277650.277721

Cédric Augonnet, Samuel Thibault, and Raymond Namyst. 2010. *StarPU: A Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines*. Technical Report 7240. INRIA. http://hal.inria.fr/inria-00467677.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (Feb. 2011), 187–198. Issue 2. DOI: http://dx.doi.org/10.1002/cpe.1631.

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, Article 40, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389051

Bin Bao, Chen Ding, Yaoqing Gao, and Roch Archambault. 2012. Delta send-recv for dynamic pipelining in MPI programs. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012) (CCGRID'12)*. IEEE Computer Society, Washington, DC, 384–392. DOI: http://dx.doi.org/10.1109/CCGrid.2012.113

Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2009. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 219–228. DOI: http://dx.doi.org/10.1145/1504176.1504209

Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE Computer Society, Washington, DC, 7–16. DOI: http://dx.doi.org/10.1109/PACT.2004.11

Cédric Bastoul. 2012. Clan: The Chunky Loop Analyzer. (2012). http://icps.u-strasbg.fr/people/bastoul/public_html/development/clan/.

Cédric Bastoul. 2013. CLooG: The Chunky Loop Generator. (2013). http://www.cloog.org.

Uday Bondhugula. 2013a. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY, Article 33, 12 pages. DOI: http://dx.doi.org/10.1145/2503210.2503289

Uday Bondhugula. 2013b. PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. (2013). http://pluto-compiler.sourceforge.net.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI'08). ACM, New York, NY, 101–113. DOI:http://dx.doi.org/10.1145/1375581.1375595

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim Yarkhan, and Jack Dongarra. 2010. Distibuted dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. *Univ. of Tennessee, CS Technical Report, UT-CS-10-660* (2010).

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.* 38, 1 (2012), 37–51.

Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. 2009. Multi-core implementations of the concurrent collections programming model. In *CPC09: 14th International Workshop on Compilers for Parallel Computers*.

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38–53.

Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. 2010. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. DOI:http://dx.doi.org/10.1109/IPDPS.2010.5470404

Michael Claßen and Martin Griebl. 2006. Automatic code generation for distributed memory architectures in the polytope model. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments, Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*.

M. Cosnard and M. Loi. 1995. Automatic task graph generation techniques. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*. IEEE Computer Society, Washington, DC, 113–. http://dl.acm.org/citation.cfm?id=795695.798168

Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE Press, Piscataway, NJ, 375–386. http://dl.acm.org/citation.cfm?id=2523721.2523771

Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2014. Task-based programming with OmpSs and its application. *Workshop on Software for Exascale Computing (SPPEXA)* (Aug 2014), 602–613. Euro-Par 2014 Workshop, Part II, Lecture Notes in Computer Science vol. 8806.

Roberto Giorgi, Rosa M. Badia, Franois Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R. Gao, Arne Garbade, Rahul Gayatri, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Kolia, Joshua Landwehr, Nhat Minh Lł, Feng Li, Mikel Lujn, Avi Mendelson, Laurent Morin, Nacho Navarro, Tomasz Patejko, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Ian Watson, Sebastian Weis, Stphane Zuckerman, and Mateo Valero. 2014. TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocess. Microsyst.* 38, 8, Part B (2014), 976–990. DOI:http://dx.doi.org/10.1016/j.micpro.2014.04.001

Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Shereen Ghobrial, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. 1997. Compiling C for the EARTH multithreaded architecture. *Int. J. Parallel Programm.* 25, 4 (1997), 305–338.

Intel CnC. 2013. Intel® Concurrent Collections (CnC) for C/C++. (2013). http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

Intel TBB. 2014. Intel® Thread Building Blocks (TBB). (2014). https://www.threadingbuildingblocks.org/.

Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. 2015. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Trans. Archit. Code Optim.* 11, 4, Article 61 (Jan. 2015), 30 pages. DOI:http://dx.doi.org/10.1145/2687652

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 211–222. DOI:http://dx.doi.org/10.1145/1250734.1250759

Christopher Lauderdale and Rishi Khan. 2012. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'12)*. ACM, New York, NY, 21–26. DOI:http://dx.doi.org/10.1145/2185475.2185478

Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2010. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the*

24th ACM International Conference on Supercomputing (ICS'10). ACM, New York, NY, 5–16. DOI:http://dx.doi.org/10.1145/1810085.1810091

Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan, and Scott B. Baden. 2012. Bamboo: Translating MPI applications to a latency-tolerant, data-driven form. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, Article 39, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389050

OCR. 2013. The Open Community Runtime Project. (2013). https://01.org/projects/open-community-runtime.

Polybench. 2012. PolyBench/C - the Polyhedral Benchmark suite. (2012). http://polybench.sourceforge.net.

Polylib. 2010. PolyLib - A library of polyhedral functions. (2010). http://icps.u-strasbg.fr/polylib/.

Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages. DOI:http://dx.doi.org/10.1145/2400682.2400712

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 519–530. DOI:http://dx.doi.org/10.1145/2491956.2462176

Chandan Reddy and Uday Bondhugula. 2014. Effective automatic computation placement and data allocation for parallelization of regular programs. In *International Conference on Supercomputing*.

Fengguang Song and Jack Dongarra. 2012. A scalable framework for heterogeneous GPU-based clusters. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM, New York, NY, 91–100. DOI:http://dx.doi.org/10.1145/2312005.2312025

Y. Song and Z. Li. 1999. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference Programming Language Design and Implementation*. 215–228.

Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. 2013. An implementation of the codelet model. In *Euro-Par 2013 Parallel Processing*. Springer, Berlin, 633–644.

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. ACM, New York, NY, 117–128. DOI:http://dx.doi.org/10.1145/1989493.1989508

Kevin Bryan Theobald. 1999. *Earth: An Efficient Architecture for Running Threads*. Ph.D. Dissertation. Montreal, Quebec, Canada. Advisor(s) Gao, Guang R. AAINQ50269.

Nicolas Vasilache, Muthu Manikandan Baskaran, Thomas Henretty, Benoît Meister, Harper Langston, Sanket Tavarageri, and Richard Lethin. 2014. A tale of three runtimes. *CoRR* abs/1409.1914 (2014). http://arxiv.org/abs/1409.1914

Sven Verdoolaege. 2014. Integer Set Library - an integer set library for program analysis. (2014). http://www.ohloh.net/p/isl.

David Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, Los Alamitos, CA, 171–180.

Tomofumi Yuki and Sanjay Rajopadhye. 2013. Memory allocations for tiled uniform dependence programs. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*. 13–22.

Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*. ACM, New York, NY, 64–69. DOI:http://dx.doi.org/10.1145/2000417.2000424

Received January 2015; revised September 2015; accepted January 2016