DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines

Mihai Budiu, Daniel Delling, and Renato F. Werneck
Microsoft Research Silicon Valley
Mountain View, CA, USA
Email: {mbudiu,dadellin,renatow}@microsoft.com

***Abstract*—We introduce DryadOpt, a library that enables massively parallel and distributed execution of optimization algorithms for solving hard problems. DryadOpt performs an exhaustive search of the solution space using branch-and-bound, by recursively splitting the original problem into many simpler subproblems. It uses both parallelism (at the core level) and distributed execution (at the machine level). DryadOpt provides a simple yet powerful interface to its users, who only need to implement sequential code to process individual subproblems (either by solving them in full or generating new subproblems). The parallelism and distribution are handled automatically by DryadOpt, and are invisible to the user. The distinctive feature of our system is that it is implemented on top of DryadLINQ, a distributed data-parallel execution engine similar to Hadoop and Map-Reduce. Despite the fact that these engines offer a constrained application model, with restricted communication patterns, our experiments show that careful design choices allow DryadOpt to scale linearly with the number of machines, with very little overhead.**

***Keywords*-combinatorial optimization; branch-and-bound; distributed computation; Dryad; distributed data-parallel execution engines**

I. Introduction

Distributed data-parallel execution engines (DDPEE) such as Dryad [1], MapReduce [2], and Hadoop [3] have become widely popular recently, in both theory [4] and practice [5], [6]. A DDPEE provides a restricted computation model that composes in parallel many *completely independent* sequential computations. The simple programming model and broad availability (particularly with the Hadoop open-source implementation) have caused worldwide adoption of such engines. While exploiting parallelism using DDPEEs is easy, they impose some important restrictions on the algorithms that can be implemented efficiently. Algorithms mapped on DDPEEs need to process a large number of independent data items, grouped in relatively large batches (partitions or shards), with a relatively uniform execution time per batch. These restrictions are a consequence of the limited communication model: processes run in separate isolated address spaces for their complete lifetime, and data exchanges can occur only when a round of processes terminates and a new one starts. The use of large data batches is required to amortize the expensive cost of communication and process creation/destruction. These restrictions make it hard to solve general optimization problems on a DDPEE.

A well-known approach to solving large (NP-hard) optimization problems is based on traversing *branch-and-bound* trees [7]. The root of the tree is the original problem, and the other nodes represent subproblems to be solved. The total number of nodes can be exponential in the problem size in the worst case. In practice, algorithms attempt to prove that certain branches of the tree cannot possibly contain the optimal solution, and can therefore be *pruned* without being visited explicitly. Even so, small input problems can lead to huge search trees. Distributed execution using a computer cluster is an obvious strategy to reduce running times, as evidenced by the wealth of different parallel and distributed solvers proposed over the years [8]–[22]. All these solvers are based on parallel computation frameworks that are more flexible than DDPEEs.

This paper introduces DryadOpt, a library for the automatic distributed execution of branch-and-bound algorithms on large clusters. Its unique feature is that it runs on top of the Dryad/DryadLINQ framework. We show how we leverage the Dryad platform despite its restrictive computation model. In particular, we adapt the impedance of the platform entirely at the application level (i.e., without changing the cluster infrastructure or compiler) to match the specific needs of our algorithm: data-parallel tree-traversal, load balancing, coordination, handling nondeterminism, and reexecution for reliability. We obtain speedups that scale linearly with the number of machines, and achieve excellent cluster utilization even when running on multiprogrammed clusters offering unpredictable resources (i.e., dynamically variable number of available machines).

DryadOpt is a library that user programs can link against. A key feature of DryadOpt is that it factors out the code responsible for distribution and parallelism, making both com-

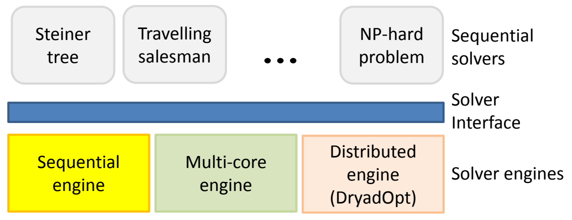

Fig. 1. Modular software architecture for solving branch-and-bound optimization problems.

pletely transparent to the user. It does so by a modular software architecture (shown in Figure 1), which can actually be used to implement generic branch-and-bound solvers. Our target users are expert developers in traditional sequential algorithms, but who have no expertise in writing parallel code.

To serve as a running example in this paper, we implemented a branch-and-bound solver for the NP-hard *Steiner problem in graphs*. The input to the Steiner problem is an undirected graph $G = (V, E)$ with positive edge costs, together with a set $T \subseteq V$ of *terminals*. The goal is to find the lowest-cost subgraph of G containing all terminals. The solution is a tree, and it may contain so-called *Steiner vertices* (i.e., vertices in $V \setminus T$), as illustrated in Figure 2. Although the Steiner vertices do not have to be part of the solution, inserting them may allow terminals to be connected more cheaply.

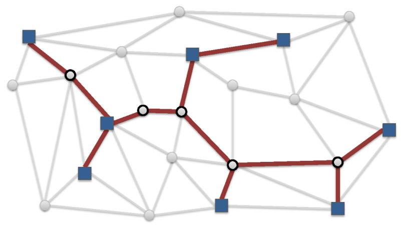
Fig. 2. An instance of the Steiner problem. Circles represent standard vertices and squares represent terminals. The highlighted tree is a possible solution to this instance.

Several techniques for solving this problem have been published, including heuristics, exact algorithms, and logic-based reduction techniques. For comprehensive overviews of some of the best existing algorithms, see [23], [24]. In particular, for a wide class of unstructured instances, the best-known solution method is branch-and-bound [23]–[25].

We emphasize that the DryadOpt library is not specific to the Steiner problem, and we have used it to implement other branch-and-bound algorithms; we report results for the Steiner problem in this paper for concreteness only.

This paper is organized as follows. In Section II we review the branch-and-bound approach in more detail, while Section III describes the Dryad/DryadLINQ framework. A high-level description of DryadOpt is given in Section IV. The solver interface API is presented in Section V, and details of the DryadOpt engine implementation are provided in Section VI. We support our claims of efficiency by demonstrating linear speedup using up to 512 cores in the experimental evaluation in Section VII. Finally, we conclude our work with lessons learned in Section VIII.

II. Branch-and-Bound

Branch-and-bound [7] is a universal and well-known algorithmic technique to solve optimization problems. In a nutshell, it interprets the input problem as the root of a *search tree*. Then, two basic operations are recursively executed: *branch* the problem (node) into several smaller (easier) problems or *bound* (prune) the search tree. The bounding can happen due to two reasons. Either the problem has become easy enough to be directly solved or one can prove that this node, and hence its descendants, cannot contribute to the optimal solution. Note that branch-and-bound is a framework, and that most of the algorithmic challenges are hidden in the pruning and solving mechanism, which are problem-specific.

Without loss of generality, in the remainder of this paper we assume we are dealing with a minimization problem. At all times, the algorithm maintains the best (minimum) known feasible solution (the *incumbent*) to the problem. In addition, each node of the branch-and-bound tree computes its own *lower bound*. If the lower bound of a subproblem is not lower than the incumbent solution, the algorithm will *fathom* (discard) the node, thus pruning the search tree. Figure 3 shows an example search tree.

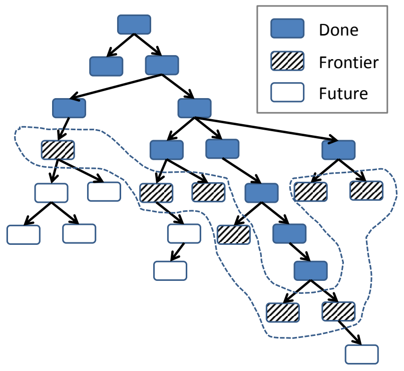

Fig. 3. The frontier is a cut in the search tree separating the completed nodes from the not-yet-explored nodes.

A. The Steiner Problem

A branch-and-bound algorithm for the Steiner problem is conceptually simple. Each subproblem can be defined as a triple $P_i = (V_i, E_i, T_i)$ representing a graph with vertex set V_i, edge set E_i, and a set of terminals $T_i \subseteq V_i$. If the subproblem is not easy enough to be fully solved, we split it into two or more subproblems with disjoint solution spaces.

A natural approach to doing so is to branch on *vertices*. We can use a fixed nonterminal $v \in V_i \setminus T_i$ to divide P_i into two subproblems. The first contains all solutions that include v, which we can achieve by setting $P_i^+ = (V_i, E_i, T_i \cup \{v\})$. The second subproblem contains all solutions that exclude v: we set $P_i^- = (V_i \setminus \{v\}, E_i \setminus E_i^v, T_i)$, where E_i^v is the set of edges in E_i having v as an endpoint.

Our sequential algorithm uses a combination of constructive algorithms and local searches [26] to compute upper bounds. For lower bounds, we use a greedy combinatorial algorithm to find dual-feasible solutions to a directed-cut-based linear programming formulation of the problem. This *dual ascent* algorithm [27] has been shown to find extremely good solutions at a fraction of the running time of a linear-programming solver [24], [25]. The sequential solver is by far the most complicated part of the implementation: it has 13 KLOC, compared to about 4 KLOC for the DryadOpt engine itself. It is thus quite remarkable that DryadOpt can create a highly efficient parallel implementation using the sequential algorithm essentially as a black box.

B. Distributed Branch-and-Bound

At any point during the search tree traversal, the *search frontier*, i.e., all open subproblems, can be processed independently. The only shared resource is the incumbent. Hence, processing the search tree in distributed fashion is very natural and has been studied for decades [28], [29]. The main challenge for such frameworks is to keep all computational resources busy. At a glance, this seems trivial: one could simply keep generating and distributing subproblems until each machine has exactly one subproblem, which is then solved sequentially. This approach fails in practice because search trees often are highly unbalanced, even exponentially so. Hence, all recent approaches [8]–[22] rely either on work stealing (a machine can ask other machines for new subproblems in case it runs dry) or on a central scheduler. Since a DDPEE does not support inter-machine communication, these approaches are not feasible in our setting.

III. Dryad and DryadLINQ

We have built DryadOpt on top of a distributed computation framework for large clusters. The software stack that we use is shown in Figure 4. In this section we focus on two layers in the stack: Dryad and DryadLINQ.

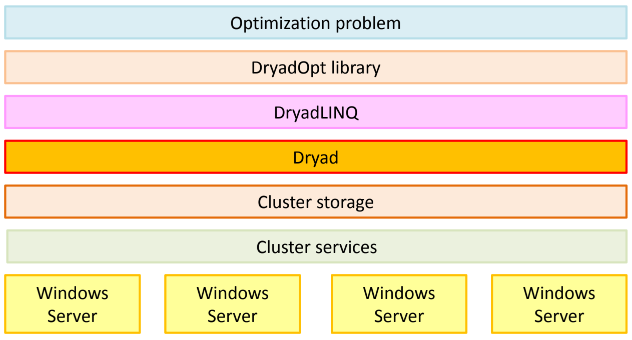

Fig. 4. DryadOpt software stack.

A. Dryad

Dryad [1] is a software layer that coordinates the execution of multiple dependent programs (processes) running on a computer cluster. A Dryad job is a collection of processes that communicate with one another through unidirectional *channels*. Each Dryad job is a directed acyclic multigraph, in which nodes represent processes and edges represent communication channels. Requiring the graphs to be acyclic may seem restrictive, but it enables Dryad to provide fault-tolerance automatically, without any knowledge of the application semantics. Figure 5 shows a hypothetical example of a Dryad job graph.

Dryad handles the reliable execution of the graph on a cluster. Dryad schedules computations to computers, monitors their execution, collects and reports statistics, and handles

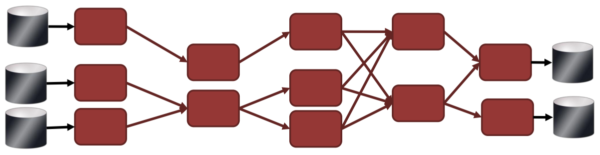

Fig. 5. Example of a hypothetical Dryad job graph; the nodes represent programs that execute, possibly on different computers, while the edges are channels transporting data between the processes. The input and output of the computation reside on the cluster storage medium.

transient failures in the cluster by reexecuting failed or slow computations. Dryad jobs execute in a *shared-nothing* environment: there is no shared memory or disk state between the various processes in a Dryad job; vertices cannot open network connections to each other; the only communication medium between processes are the channels.

B. DryadLINQ

DryadLINQ [6] is a compiler which translates LINQ .Net computations into Dryad job graphs that can be executed on a cluster by Dryad. LINQ is essentially a set of operators that perform computations on *collections* of values; the LINQ language is similar to the SQL database language, and the LINQ collections are the equivalent of database tables; unlike SQL, LINQ is “embedded” within the other .Net languages (i.e., there are LINQ operators for C#, VB, F#). The essential LINQ operations are: apply a transformation (function) to all elements in a collection, filter elements according to a predicate, group elements by a common key, aggregate the elements according to some function (e.g., addition), and join the elements in two collections using a common key. DryadLINQ collections are partitioned, with different parts residing on different machines, as shown in Figure 6. For the programmer the main benefit of using DryadLINQ is that of using a single high-level programming language (.Net) to write the application, blending seamlessly the local and distributed parts of the computation in a single program, due to the tight embedding of LINQ in .Net languages.

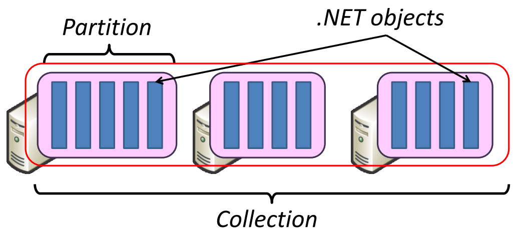

Fig. 6. DryadLINQ data model: collections of typed values partitioned among several computers.

In general, the collection elements must be moved between computers during the computation, so the in-memory data structures need to be *serialized* to a shared physical medium, either a disk or the network. DryadLINQ automatically generates serialization and de-serialization code, but the user can replace the default serialization routines with custom ones.

Not only does DryadLINQ generate job graphs, but it can also generate parallel multi-threaded code for each of the processes in a job graph, using multiple cores. The parallelization across cores uses very similar techniques to the distribution across machines, noting that each partition of a collection is just a smaller collection itself. DryadLINQ translates LINQ operations on the large collections into LINQ operators on individual partitions, which are further partitioned across cores and processed in parallel.

DryadLINQ provides a generalization of the popular Map-Reduce computation model, implemented in the Google proprietary stack [2] and the open-source Hadoop stack [3]. A map-reduce computation is just a particular sequence of LINQ operators (SelectMany/GroupBy/Aggregate).

IV. DryadOpt Overview

In this section we outline the strategy used by DryadOpt for parallelizing the exploration of a search tree.

The entire execution is orchestrated by the user's machine (the *client workstation*), which coordinates the execution of many rounds of computation on the cluster by launching multiple DryadLINQ computations.

Initially the client workstation reads the problem instance. This is now a search tree with a single (root) element. The client workstation then repeatedly runs the sequential solver locally, in an attempt to generate a large frontier, to provide enough work for the cluster machines.

This frontier becomes the inductive basis for the algorithm, which proceeds in rounds. Each round is executed on the cluster; a round starts from the current frontier and explores a new set of nodes in the search tree, resulting in a new frontier. After a round of computation on the cluster, control is returned to the client workstation, which decides whether to start a new round or terminate. The client workstation declares the algorithm over when the frontier is empty.

The nodes in the frontier that is input and output to each round are partitioned into disjoint sets. Each partition is manipulated by an independent machine in the cluster; we attempt to keep partitions relatively large (many tree nodes), to allow machines to compute independently for a long time. After each round the nodes in the output frontier are redistributed among machines randomly to provide load-balancing of the work.

The cluster execution is handled by the Dryad runtime in a reliable way: Dryad handles the job initialization, schedules the data movement, reexecutes computations if they fail or are too slow, and allocates cluster machines as they become available.

In order to exploit the many cores available on each cluster machine, we have implemented a separate multi-threaded solver engine, which partitions the work of each machine on multiple cores. The structure of the multi-threaded engine is actually extremely similar to DryadOpt itself; the only difference is that the multi-threaded engine also uses a simple form of work-stealing, which is easily implemented between threads that share a single address space.

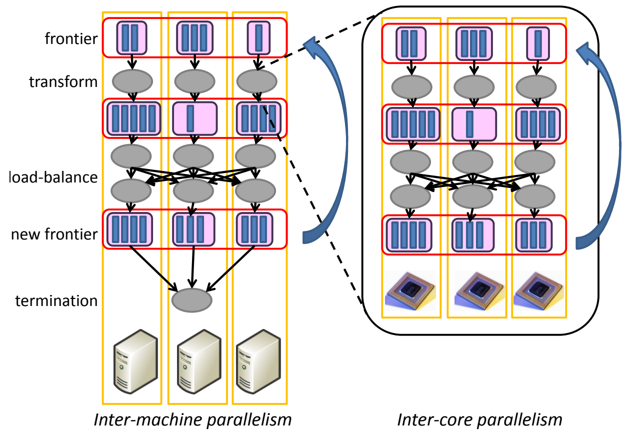

Fig. 7. Parallelization strategy uses nested parallelism with a similar approach across machines and cores.

Figure 7 shows the resulting structure of the computation. The cluster-level computation is shown on the left; each vertical stripe is a machine. Each "transform" stage is shown blown-up on the right: each vertical stripe is a core. Note that the structures of the two computations are very similar.

The handling of both cluster and multi-core executions is completely hidden from the user. The user only has to implement a sequential solver, which is repeatedly invoked during the transform stage.

We next describe the API offered by our infrastructure to the users, followed by more details on the DryadOpt implementation and caveats.

V. The Solver Interface

This section describes the interface between the sequential solvers (provided by the user) and the execution engines, shown in the middle of Figure 1. To use any of the engines the user must only define three classes, representing an *instance* (subproblem), the state of the computation, and the actual function that processes subproblems. The Solver API is represented as a set of three C# interfaces to which the users must adhere (these are similar to the Java interfaces and C++ abstract base classes). We discuss each interface in turn, then show how to invoke the distributed branch-and-bound solver.

A. Instance

The user must create a class to represent each subproblem to be processed, implementing the following interface:

```
[Serializable]
public interface IBBInstance {}
```

Note that `IBBInstance` has no required methods. It must be serializable, however, since objects of this kind will be shipped between machines by DryadLINQ. Besides representing the subproblem itself, an object of this class may also contain various pieces of subproblem-specific information, such as a lower bound.

An instance does not need to be self-contained: instead, we can describe it as a set of operations to be applied to its parent in the branch-and-bound tree. We call this an *incremental representation*.

For our example, the Steiner tree problem, the class that implements this interface is called `SteinerInstance`, and it represents a graph in incremental form. The instance at the root of the tree (the original problem input) describes the whole graph, with its list of nodes and weighted edges, together with the original set of terminals. An instance elsewhere in the tree contains a list of terminal insertions and vertex or edge deletions to be applied to the parent.

We note, however, that in general the user is by no means required to use incremental representations: subproblems can be represented in full if desired.

B. Global State

This class is a container for the global state of the computation. It must implement the following interface:

```
[Serializable]
public interface IBBGlobalState {
   void Merge (IBBGlobalState s);
   void Copy (IBBGlobalState s);
}
```

The user will implement objects of this class to contain global, problem-specific information about the computation. An obvious example of a global state field is the value of the best upper bound found so far. In our Steiner application, the class implementing this interface (`SteinerBBLocalState`) maintains the upper bound and the corresponding best solution.

During the distributed computation, each machine receives a private copy of the global state, and updates it based only on local information. The state will thus diverge between machines. Periodically, DryadOpt will collect the various versions of the global state and *merge* them into a single one.

The two required methods of `IBBGlobalState` enable these operations to be performed. If `s` and `t` implement the `IBBGlobalState` interface, `s.Copy(t)` copies the contents of `t` into `s`. Calling `s.Merge(t)` changes `s` by merging the state it represents with the one represented by `t` (which remains unchanged). Note that `merge` must be *idempotent*, i.e., repeatedly merging `s` with `t` several times should produce the same result as merging just once.

C. The Sequential Solver

Finally, the user must write a *sequential solver* class implementing the following interface:

```
public interface IBBSolver {
   List<IBBInstance> Solve (
      List<IBBInstance> incrementalSteps,
      IBBGlobalState state,
      BBConfig config
   )
}
```

The `Solve` method can be arbitrarily complicated, and is entirely up to the user. Normally this is a very sophisticated and carefully engineered sequential piece of code which executes very efficiently; most of the code written by the user is expected to reside in (or be called from) this function. This function is invoked by our framework as an *upcall*. `Solve` receives as input a single subproblem (expressed as a chain of incremental steps), and outputs a list of child subproblems. The output list may be empty, indicating that the user does not wish to explore this branch of the computation further. This can happen either because the subproblem was solved to completion, or because of a user-implemented heuristic choice.

Note that the first input to `Solve` (`incrementalSteps`) is actually represented as a *list* of objects of type `IBBInstance`. The ordered list contains all instances on the path from the root of the search tree to the open subproblem we actually need to solve. The input to `Solve` is thus a list of increments. If needed, the `Solve` function uses the increments to compute a complete (nonincremental) description of the subproblem internally.

The second parameter of `Solve` (`state`) is an object representing (the local version of) the global state of the computation. The `Solve` function may update its contents as necessary.

Finally, the third parameter (`config`) is a (read-only) object containing various hints that may be useful for `Solve`. `Solve` may completely ignore this parameter without compromising the correctness of the application. For example, the configuration contains a seed (used to coordinate random number generators across machines), the current error verbosity level, and the desired branching factor.

For example, in our Steiner application the solver class (called `SteinerBBSolver`) uses the branching factor to decide how many child instances to generate, in a trade-off between parallelism and memory utilization (described in detail in Section VI-B).

D. The Engine

The actual optimization engine is a generic parameterized class that implements the following interface:

```
public interface BranchBoundEngine
   <TInstance, TSolver, TGlobalState>
   where TInstance : IBBInstance
   where TSolver : IBBSolver
   where TGlobalState : IBBGlobalState
{
   TGlobalState
      Run(TInstance i, TGlobalState g);
}
```

To prove the flexibility of the engine itself we have implemented three versions of the branch-and-bound optimizer, shown in Figure 1: a sequential version, a multi-threaded version, and DryadOpt, a distributed version that runs on top of the DryadLINQ execution engine. DryadOpt also invokes the multi-threaded engine on each machine in the cluster.

E. Putting Everything Together

To solve an instance of the Steiner tree problem we have to initialize two objects, representing a problem to solve and the initial state:

```
SteinerInstance problem;
SteinerBBGlobalState uprBd;
```

Then we invoke the DryadOpt solver:

```
SteinerBBSolver seq = new SteinerBBSolver();
DryadOptEngine<SteinerInstance,
   SteinerBBSolver,
   SteinerBBGlobalState>
   dryadopt = new DryadOptEngine(seq);
TGlobalState
   result = dryadopt.Run(problem, uprBd);
```

The global state resulting from this computation contains the solution instance.

VI. Implementation

Although the basic structure of the DryadOpt search algorithm is quite simple, there are several details we have to deal with in order to ensure correctness and to obtain good performance. This is the scope of this section.

A. Distribution

In order to use the LINQ language, we have to express the program as a chain of computations on collections. DryadOpt manipulates collections of *work units*. A *work unit* is a container packaging: (1) a collection of open subproblems to be solved, including their position in the search tree, (2) a local version of the *global state*, and (3) computation statistics. If subproblems are represented incrementally, a work unit also maintains the ancestors of these subproblems in the branch-and-bound tree. Therefore, each work unit actually represents an entire subtree, whose leaves are open subproblems. Each search node of a work unit subtree is represented only once (otherwise we could face an exponential space blow-up), as shown in Figure 8.

DryadOpt maintains two invariants on the collections of work units at the end of each computation round: first, different work units represent disjoint sets of open subproblems. Second, the union of all work units is a frontier of the computation, containing all open subproblems.

From the point of a view of a single machine, one round of DryadOpt is quite straightforward. The machine receives a collection of work units (one from each machine in the previous round) and *merges* them into a single work unit. It then *transforms* it into another work unit by processing some (or all) of the open subproblems in the corresponding subtree, possibly generating new subproblems. (Section VI-B explains how DryadOpt chooses which subproblems to process first.) The work unit thus created is then *split* (partitioned) into k work units for the next round. We discuss the three basic operations (*merge*, *transform*, and *split*) in turn.

To *merge* work units, the algorithm simply combines the subtrees represented in each of them. We know their sets of leaves are disjoint, but some ancestors may appear more than once. Since these nodes have already been processed (by the inductive invariant), duplicates can be discarded. The *merge* operation also combines statistics and global states of the individual work units.

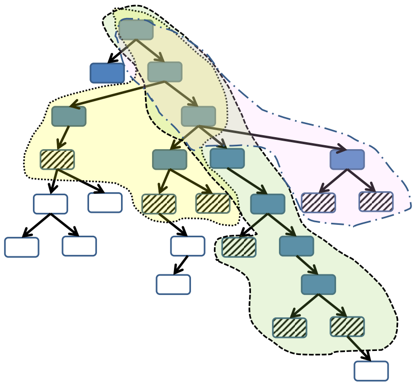
Fig. 8. Search tree with three work units highlighted with different dashed and dotted lines. A work unit contains a complete subtree ending in a subset of the nodes of the frontier.

To *transform* a work unit, DryadOpt repeatedly selects a leaf (open subproblem) from its current subtree and calls the user-defined `Solve` function on it. The resulting subproblems (if any) are then attached to the tree. This process is repeated for a certain amount of time (as detailed in Section VI-D), or until there are no more subproblems to solve. The resulting subtree (even if empty) is then represented as a new work unit.

Finally, DryadOpt *splits* a work unit into k work units by partitioning its open problems (leaves) into k groups. The partition is always balanced: if there are ℓ leaves, the algorithm produces k sets with either $\lfloor \ell/k \rfloor$ or $\lceil \ell/k \rceil$ elements, with leaves assigned to these sets at random.

Note that keeping k trees in memory could be rather expensive. Fortunately, we can leverage the fact that DryadLINQ performs streaming computation on the collections, only loading what is strictly necessary from the channel storage into memory. The *split* operation generates work units one at a time, committing them immediately to the output channel and garbage-collecting them from memory. Similarly, *merge* does not need to keep all the k input trees in memory at once: it reads the k inputs in streaming fashion and constructs incrementally a global tree by merging the inputs one at a time. We stress that all these operations (including the work unit abstraction) are completely hidden from the user.

B. Traversing the Tree

There are two standard ways of traversing a search tree. *Breadth-first search* (BFS) processes subproblems that are closer to the root first. This allows broad exploration of the search space and may eventually lead to a large (exponential) number of open problems. In contrast, *depth-first search* explores deeper nodes first. This rule ensures that there are at most $h\beta$ open problems at any time, where h is the maximum height of the tree and β the maximum *branching factor* (number of children) of a node. DFS tends to be more memory-efficient, since the height of a branch-and-bound tree is typically polynomial (even linear) on the input size. On our Steiner solver, for example, the height is bounded by $|V| - |T|$.

To ensure that our computational resources are fully utilized, DryadOpt must generate enough subproblems to keep all machines occupied. This favors BFS, which tends to generate more problems. However, BFS is very memory-intensive. A machine switches from BFS to DFS when it is used near its maximum capacity. If a processor already has more than τ open subproblems (where τ is a user-defined threshold), it runs DFS; otherwise, it runs BFS.

A third popular traversal policy for branch-and-bound trees is *best-first* (BeFS), which prefers subproblems that are more likely to lead to good solutions according to some problem-specific metric [28]. DryadOpt could easily be augmented to support BeFS. However, we deliberately kept the Solver API simple, and thus it does not currently provide a way to compare subproblems for cost.

C. Load Balancing

In essence, the goal of DryadOpt is to traverse a rooted tree in parallel. With k machines, a simple algorithm to achieve this would locally split the initial problem into k subproblems, send each subproblem to a different machine, then wait for them to finish. Because a typical branch-and-bound tree is extremely unbalanced, however, some machines will complete much faster than the others.

Given the restrictive communication model imposed by DDPEEs, DryadOpt must plan ahead of time to avoid such situations. If a machine runs out of problems in the middle of a round, it needs to wait for the round to end. This is in contrast with most other distributed branch-and-bound solvers [8]–[22], where communication is often costly, but never impossible. In the following we discuss the techniques we employed to overcome this problem.

First, by working in rounds and performing *periodic redistributions*, we ensure all machines start each round with roughly the same number of open subproblems.

Second, by making sure these redistributions are random, we try to attenuate the correlations between nearby subproblems in the search tree. It is often the case the entire subtrees are “easier” than others; in other words, the heights of the subtrees rooted at two siblings are often similar. In general we do not know whether a node is relatively easy or hard until we actually process its entire subtree, but randomization ensures that most machines have access to nodes of both kinds.

Third, DryadOpt tries to boost the effectiveness of the first two techniques by maximizing the number of subproblems available. One obvious way in which it does so is by preferring BFS over DFS whenever possible, as described above. It also uses *hints* to encourage the user-defined `Solve` function to generate more child nodes at once when necessary.

D. Synchronization

As described, DryadOpt works in rounds, with k machines operating in parallel. Ideally all machines should be busy at all times, which means they should all finish simultaneously to prevent machines to idle waiting for laggards. Unfortunately, the DDPEE framework provides no communication channel among processes on the same round to be used to synchronize their termination.

Instead, DryadOpt allocates (up front) a *budget* to each machine indicating how much work it can perform. A machine stops as soon as it reaches its budget limit, even if it still has open problems to solve. (Of course, the machine also stops if it runs out of subproblems.) An obvious way to specify the budget would be in terms of invocations of the sequential solver or some other deterministic operation count. Unfortunately, the time to process a subproblem can vary widely within the same branch-and-bound tree, making this approach unsuitable for load-balancing.

Alternatively, we could define the budget in terms of *real elapsed time*; each node is given a certain number of seconds b to run, and stops when this limit is reached. The limit should be high enough to amortize the communication and setup costs between rounds, but low enough to ensure reasonably frequent redistributions for load-balancing purposes. Note that b is user-defined, and can be set appropriately depending on the properties of the system and the problem.

This approach works well as long as all k processes start at the same time and run to completion. However, on a shared cluster infrastructure, where multiple jobs compete for resources (or in the presence of failures), the number of machines available for a job can fluctuate randomly around k.

To maximize utilization, the budget includes not only a computation time but also a *deadline*. When starting a new round, the client workstation uses the current time t_0 and gives to each machine a deadline of $t_0 + b$. To avoid pathological cases when clocks are not properly synchronized, each cluster machine uses $\min(t_0 + b, t_0^l + b)$, where t_0^l is its local time.

E. Nondeterminism

As a side effect of using deadlines as the stopping criterion, the program becomes nondeterministic: different runs of the same program could lead to different results[1]. In the presence of Dryad's reexecution for fault tolerance, this can affect correctness.

We illustrate what can go wrong with an example. Suppose machine N runs to completion during the first round and produces three work units during its allotted time: W_1, W_2, and W_3. When the next round starts, machine N_1 reads W_1, machine N_2 reads W_2, and machine N_3 attempts to read W_3 but fails due to data corruption. At this point, Dryad must reexecute the computation of N to generate a new copy of W_3. This nondeterministic reexecution produces three work units, W_1', W_2', and W_3'. Now the computation of machine N_3 is executed again, using W_3' as an input. (N_1 and N_2 do not need to be reexecuted.)

In general, however, $W_1 \cup W_2 \cup W_3' \neq W_1 \cup W_2 \cup W_3$. This could cause the computation to miss some parts of

[1]Note that the user-supplied function `Solve` could also be nondeterministic; DryadOpt will work correctly in this case as well.

the search space (the search tree nodes in $W_3 \setminus W'_3$). This violates the invariant that the system maintains a frontier of the computation at all times.

To work around this issue, DryadOpt *materializes* (by writing to the cluster storage medium) the output of each round. Once outputs are written, the processes that generated them cannot be reexecuted. Note that we still allow Dryad reexecutions within a round: the program is correct regardless of whether N outputs (W_1, W_2, W_3) or (W'_1, W'_2, W'_3). What it must not do is mix the outputs of two executions.

Materializing the results has another benefit: it provides "free" computation *checkpointing*. This enables DryadOpt to resume computations that fail (or are stopped) from the most recent checkpoint, allowing us to run computations for multiple months of wall clock time.

F. Parallelism

DryadOpt supports multi-processor and multi-core executions within the same machine (either standalone or within the cluster). The parallel execution follows the same approach as the distributed one, in rounds. Each machine partitions its input into c work units, each for a different thread. Threads use a local time budget, a fraction of the machine budget. The resulting work units are then (sequentially) merged, and the work unit thus generated is randomly partitioned among all c cores, starting a new round. This is repeated until the entire machine (as opposed to individual threads) runs out of time, or until it has no more open subproblems to solve.

As in the distributed case, a core may run out of subproblems before its time expires. When this happens, it sets a bit in shared memory stating that it is finished. By polling this bit, the other threads stop as soon as they finish processing the subproblem they are currently working on, thus allowing an early redistribution of the available subproblems. This is essentially a simple implementation of work stealing within a single machine.

G. Statistics

During the computation, DryadOpt automatically keeps problem-independent statistics about the computation, reporting them to the client workstation periodically. Statistics include the total number of subproblems solved, the number of open subproblems, the total time spent, and the total (sequential) time spent on the user-defined `Solve` function.

VII. Experimental Evaluation

We tested DryadOpt on a cluster with 240 computers, each with 16GB of RAM and two 2.6 GHz dual-core AMD Opteron processors, running Windows Server 2003. Jobs on the cluster are initiated by and report to our *client workstation*, an Intel Core 2 Duo E8500 at 3.16 GHz with 4 GB of RAM, running Windows 7 64-bit.

As already mentioned, our example application is the Steiner problem. Like DryadOpt itself, our Steiner solver was implemented in C# using .Net 3.5 and Visual Studio 2008.

We focus our experiments on *incidence instances*, a well-known benchmark for the Steiner problem consisting of random graphs with adversarial distribution of edge weights [30]. On these instances, the sequential performance of our solver is competitive with the best previously reported in the literature [23]–[25]. (For some other classes, such as VLSI instances, state-of-the-art performance requires linear programming [23], [25], [30], which for simplicity we have not implemented in our prototype.)

The best available results on incidence instances were reported in detail by Polzin [24] and obtained by an algorithm by Polzin and Daneshmand [23]. Of the 400 incidence instances, almost all can be solved in a few seconds or minutes by their algorithm. On the nine hardest instances they solved, the sequential time of their C++ code ranged from 5 hours (instance i320-312) to almost 5 days (instance i640-342) on a Sunfire 15000 with a 900 MHz SPARC III+ CPU. (One core of our machines is roughly twice as fast.)

The purpose of our experiments is to illustrate how sophisticated combinatorial algorithms can benefit from DryadOpt. The sequential Steiner solver itself is not novel—it just reimplements ideas already used in other solvers [24]–[26]. We did try some other test problems, such as a toy Sudoku solver and a simple algorithm for Capacitated Vehicle Routing [31], and they scaled equally well. Unlike our Steiner solver, however, their sequential versions are far from the state of the art. Hence, we do not report these results here.

A. Basic Experiments

In our first experiment, we ran our solver on each of the 9 hardest instances using 32 machines, with 4 cores each. In each execution, the maximum branching factor β was set to 8, the threshold τ was set to 2000 (a machine switches from BFS to DFS when there are τ open problems), the client workstation was allowed to run for two minutes (during initialization), and the time budget b allocated to each round on the cluster was 10 minutes. The results are reported in Table I.

TABLE I
DryadOpt runs on selected incidence instances using 32 machines with 4 cores each.

INSTANCE		SEARCH TREE		RUNNING TIME		
NAME	OPT	NODES	RND	CPU	WALL	RATIO
i320-312	18122	682705	2	72472	963	75.2
i320-314	18088	998566	2	110236	1216	90.7
i320-315	17987	1183504	2	127438	1378	92.5
i640-211	11984	751215	4	170450	2234	76.3
i640-341	32042	480730	9	571043	5984	95.4
i640-342	31978	57795	2	67593	942	71.7
i640-343	32015	325522	7	393501	4083	96.4
i640-344	31991	224639	6	333203	3558	93.6
i640-345	31994	184744	4	243652	2585	94.3

For each instance, the table shows the optimum solution (OPT), the size of the branch-and-bound tree traversed (NODES), and the number of rounds in the distributed execution (RND). We also report the time spent by the execution in two ways. The *CPU time* (CPU) is the total time spent on the problem-specific, user-defined `Solve` function. The *wall clock* *time* (WALL) is the the total execution time of the program, i.e., the time elapsed (as measured by the client workstation) from the moment it reads the original instance until it writes the final output. Times are given in seconds. We also report the ratio between these times (RATIO).

Note that most instances can be solved in less than one hour, and all under two, even though the total CPU time is as high as a week. The ratio between the total CPU time and the wall clock time is usually higher than 90, indicating that DryadOpt can use the resources at its disposal fairly efficiently (the theoretical best possible, 128, is not too far).

In a few cases, however, the efficiency is lower, closer to 70. This tends to happen for smaller, easier instances, which can be solved in very few rounds (as low as two). In such cases, during a given round, many cluster machines tend to run out of subproblems early, and must idle until all others are done.

B. Scalability

To better assess the performance of DryadOpt, we focus on i640-341, the hardest instance in our test set. It has 640 vertices, 40896 edges, and 160 terminals. We reran our solver on this instance several times, varying the number of machines (from 16 to 128) and the number of cores per machine (1, 2, or 4). To have more control over the execution, in this experiment we provided the value of the optimum solution (34042) to the algorithm as an input. Of course, the algorithm must still visit the remainder of the tree to prove that this upper bound is indeed optimal.

The initialization phase (on the client workstation) was allowed to run for 2 minutes using both cores. Cluster machines were allowed to run for 10 minutes in each round. Finally, the maximum branching factor β was set to 8. The threshold τ for switching from BFS to DFS was set to 10 000, high enough to ensure BFS was always used. As a result, all runs solved the exact same subproblems (470 795 in total). Note that, for this particular instance, the number of subproblems increases by only 2% if we do not provide the optimal primal solution in advance (see Table I), indicating that our algorithm can find it rather quickly by itself.

We start our analysis of these runs with Table II, which reports the total CPU time of each execution (i.e., the total time spent by the `Solve` function).

TABLE II
TOTAL CPU TIME FOR I640-341 WHEN VARYING THE NUMBER OF MACHINES AND CORES.

	CORES		
MACHINES	1	2	4
16	410468	430204	495351
32	406853	425902	495421
64	407472	428154	499458
96	408178	430574	506130
128	407871	431546	510926

If a single core per machine is allowed, the total CPU time is roughly 410 thousand seconds (4.75 days), and is mostly independent of the number of machines used. This would be the total running time if the entire execution were serialized and there were no overhead for communication, selecting the next problem to solve, or assembling each new subproblem from its incremental definition (as a subpath of the search tree).

Note that using multiple cores per machine actually increases the total CPU time, since there is always some contention between multiple cores on the same machine (for memory access, for instance). The slowdown is rather small (about 5%) when we go from one to two cores, but higher than 15% when going from two to four. Recall that each machine in the cluster actually has two dual-core CPUs. If only two cores are in use, each thread has an entire CPU and memory controller at its disposal; this is not the case with four cores. The overhead of DyradOpt-initiated communication is negligible in the parallel execution, and the speedups it obtains are still significant.

These numbers should be compared with those on Table III, which shows the actual (wall clock) execution time of the algorithm, including the two minutes spent on the client workstation. Table IV contains the same information, but given as a speedup relative to a hypothetical sequential execution of 407 thousand seconds. They confirm that, in general, increasing the number of machines is better than increasing the number of CPUs per machine. For example, the algorithm is about 20% faster when running on a single core of 128 machines than on 4 cores of 32 machines. With 512 cores, this instance can be solved in less than half an hour, almost 230 times faster than a sequential execution.

TABLE III
TOTAL (WALL CLOCK) TIME FOR I640-341, IN SECONDS.

	CORES		
MACHINES	1	2	4
16	34694	18795	11003
32	16342	9008	5298
64	8107	4311	2681
96	5509	3030	2038
128	4188	2551	1780

TABLE IV
SPEEDUP RELATIVE TO A SEQUENTIAL RUN ON I640-341.

	CORES		
MACHINES	1	2	4
16	11.7	21.6	37.0
32	24.9	45.2	76.8
64	50.2	94.4	151.8
96	73.8	134.3	199.7
128	97.1	159.5	228.6

With a single core, increasing the number of machines leads to almost perfect speedup. On 128 machines, the algorithm is about four times faster than on 32. At first sight, this appears not to be the case with four cores: quadrupling the number of machines (from 32 to 128) reduces the total time by a factor of three. Note, however, that with 128 machines there is enough time for only three distributed rounds, as Table V shows.

TABLE V
NUMBER OF ROUNDS NECESSARY TO SOLVE *i640-341*.

	CORES		
MACHINES	1	2	4
16	51	28	17
32	25	14	8
64	13	7	4
96	9	5	3
128	7	4	3

As we have argued, one should expect the computation to be more imbalanced during the first and last rounds, with the "middle" rounds perfectly balanced. As the results with one core show, DryadOpt can enable linear speedups for relatively longer runs.

Another way to observe this phenomenon is given by Figure 9, which shows the total resource usage (given by the number of cores multiplied by the total wall clock times in minutes) as a function of the total number of cores used.

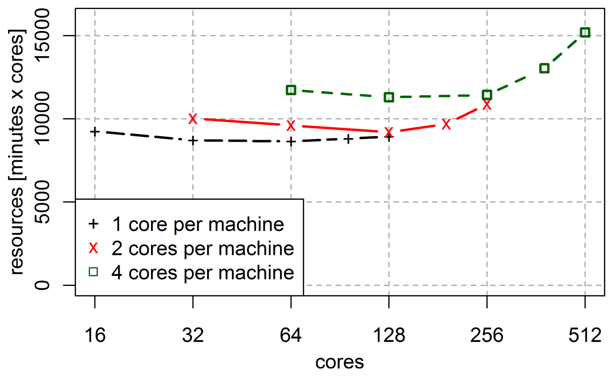

Fig. 9. Total resource usage for i640-341.

While the number of cores is relatively small, the total usage remains roughly constant, indicating that more cores can be added to the computation with no loss in efficiency. As the number of rounds gets significantly smaller, however, the imbalance between machines becomes more pronounced.

C. Branching Factor

We have established that, although DryadOpt can achieve reasonable speedups when the number of rounds is very small, it does even better when they are higher. Simply having more rounds is not enough, however. Since machines cannot communicate directly, DryadOpt relies on the availability of a large number of open subproblems at the beginning of each round. Higher branching factors help achieve this goal.

In the experiments above, the branching factor was set to $\beta = 8$. To measure its importance, we ran DryadOpt on the same instance, with $\beta = 2$, 4, and 8. On all cases, the maximum computation time per machine was set to 480 seconds and each machine was allowed to use a single core. We do not use deadlines in this experiment.

Figure 10 shows that increasing the branching factor tends not only to lead to fewer rounds, but to make these rounds more efficient. With all branching factors, all machines were fully utilized between rounds 3 and 6. With lower branching factors, however, machines were much more likely to run dry in the beginning and at the end.

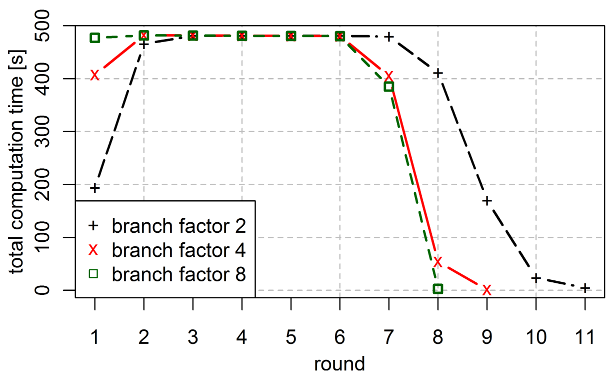

Fig. 10. Impact of branching factor on running times and number of rounds.

In general, the importance of the branching factor depends on the complexity of the `Solve` function, which is responsible for generating new subproblems. On i640-341 (our example instance), it takes about 1 second on average, though this number can be closer to 10 seconds for nodes higher up in the tree. This is a significant fraction of the total time allocated to each round, which makes high branching factors crucial. In contrast, Table I shows that the average subproblem generated from instances i320-31x is solved much quicker, in about a tenth of a second. A lower branching factor is less of a problem in such cases.

Of course, the branching factor β is just an example. Different tradeoffs can be observed for other parameters (such as τ and the length b of a round) as well, depending on the problem to be solved. By specifying individual parameters, users can fine-tune the performance.

D. Scheduling and Fault-Tolerance

Figure 11 shows the benefits of the real-time deadline scheduling in our implementation. The figure, created with the Artemis cluster monitoring tool [32], shows two job schedules. The horizontal axes are aligned, and represent the time elapsed since the start of the job. The vertical axis represents the 240 machines in the cluster. Each horizontal line indicates a machine performing a job-related computation. The top graph uses fixed-time budgets only, while the bottom graph uses deadlines as well. Both jobs have a 10-minute time quota per round. The top job uses only 3 rounds to traverse the complete search tree, while the bottom one requires 5 rounds. It is visible that the 5 rounds are tightly aligned to the real-time deadlines. In these particular runs the top job lost 64 processes due to preemption from the cluster-level scheduler, while the bottom job lost 396 (our cluster uses a fair-share scheduler [33], which can preempt processes when new jobs are submitted; our figures do not show the cluster utilization by jobs from other concurrent users). Despite the more adversarial environment, the real-time scheduling achieves 20% faster execution and a better utilization of the available cluster resources.

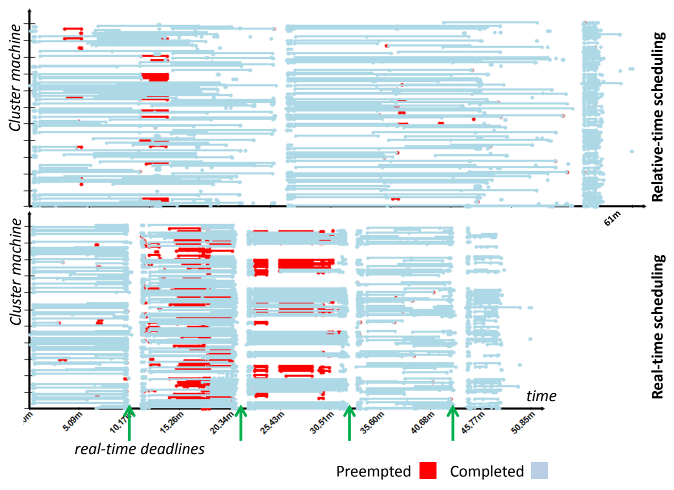

Fig. 11. Influence of scheduling policy on cluster utilization.

E. *Branch-and-Bound as a Heuristic*

Among the 400 incidence instances on the SteinLib [30], 395 have been solved to optimality at the time of writing. Each of the five remaining instances (i640-31[1–5]) has 640 vertices, 4135 edges, and 160 terminals. We tried using our method to solve them exactly, but based on partial executions we estimate each computation would take a few months, even using all 960 cores in our cluster. This makes these instances significantly harder than the others, which can all be solved sequentially in no more than a week (usually much less).

We stress, however, that branch-and-bound algorithms can still be useful in such situations. By simply changing the pruning rule, our Steiner solver can prove that known upper bounds are actually close to the optimal solution. Recall that we normally prune a subproblem P if its lower bound $L(P)$ is at least as high as the best known upper bound U. If, instead, we prune whenever $L(P) \geq (1 - \epsilon)U$, we guarantee that the optimum solution is within a factor of ϵ of the final upper bound found by our algorithm.

We set $\epsilon = 0.02$ and ran this modified version of our algorithm on all five open instances, starting from the best upper bounds listed on the SteinLib (taken from [24]). Table VI reports the result. In each case, we show the upper bound available at the SteinLib, followed by the best solution we found, the number of nodes explored, and total CPU time in seconds. Note that the primal heuristics embedded in our algorithm actually found better solutions in every case. Moreover, the optima of all these instances is guaranteed to be within 2% of the reported values.

Recently, an improved upper bound has been reported[2] for instance i640-312: 35771. Using $\epsilon = 0.01$, our algorithm could prove that the optimum is at most 1% lower than this value; to do so, it visited 23.4 million nodes in 90.5 CPU-days (it actually ran overnight on 64 machines in our cluster).

[2]See *http://areeweb.polito.it/ricerca/cmp/node/383.*

TABLE VI
RESULTS FOR OPEN INCIDENCE INSTANCES WITH $\epsilon = 0.02$.

INSTANCE	STEINLIB	DRYADOPT	NODES	TIME (S)
i640-311	36005	35895	787044	269867
i640-312	35997	35921	737367	232052
i640-313	35758	35697	6793	2525
i640-314	35727	35712	34040	8935
i640-315	35934	35887	110173	33224

VIII. CONCLUSION

While DryadLINQ provides a very high-level abstraction for writing distributed data-parallel programs, not all of the features provided by the DryadLINQ compiler and the Dryad runtime match the needs of our application domain. The system is however flexible enough for us to circumvent the undesired characteristics without undue effort, and, most importantly, without making any changes in the underlying distributed software stack.

In particular, we had to override the following features:

- **Serialization:** The DryadLINQ-generated serialization code assumes that the data collections that are processed are composed of independent elements (i.e., the elements do not share any state). For DryadOpt the work units share common subpaths for important space savings; we have thus implemented custom serialization which correctly dismantles and reconstructs shared data structures.
- **Scheduling:** Dryad jobs are "virtualized": a Dryad job can have many more processes than available machines; the Dryad runtime schedules the processes on the existing machines by time-sharing available machines. However, especially in the presence of multiple competing jobs on a cluster, this can lead to inefficient cluster utilization. DryadOpt overrides Dryad's scheduling at the application layer using timers triggered by absolute time deadlines.
- **Reexecution:** Dryad provides fault-tolerant execution on an unreliable cluster by monitoring and reexecuting failed processes. The implicit assumption is that all Dryad job processes are idempotent, generating the same output if they are reexecuted. However, due to DryadOpt's use of timers for scheduling, execution is not deterministic. Reexecution across some computation boundaries can lead to a violation of the correctness invariants of the search. DryadOpt controls reexecution boundaries by periodically materializing partial results to stable storage.
- **Multi-core parallelization:** DryadOpt overrides the default DryadLINQ parallelization algorithm across cores to use work stealing and timers for uniform load-balancing.
- **Partitioning:** DryadOpt achieves load balancing of the computation by repartitioning the work units periodically. DryadOpt assigns keys to work unit elements explicitly in order to achieve round-robin partitioning.
- **Iterated computations:** Dryad graphs are constructed statically, but the depth of the search process cannot be statically bounded. DryadOpt implements iterative computations by creating and launching multiple DryadLINQ jobs (rounds) for each optimization problem.

With all these adaptations the resulting system is remarkably efficient (providing very good speed-ups and scale-ups on clusters up to hundreds of machines), and extremely generic (the API is simple and accommodates a wide variety of optimization problems).

Although we have not performed the exercise of porting our application on top of other DDPEE platforms, we believe that many of the lessons learned are directly transferable. The platform-specific fraction of the DryadOpt code is tiny (only 12 lines are LINQ statements). Moreover, as Figure 7 shows, our execution plan can be interpreted as a sequence of map and reduce operations.

Since the introduction of DDPEEs in the seminal work of [2] the space of problems that can be implemented efficiently on DDPEEs has constantly grown. This work demonstrates that optimization problems belong squarely in this space.

Acknowledgments

The authors would like to thank Dennis Fetterly, Michael Isard, Frank McSherry, and Yuan Yu for helpful discussions.

References

[1] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. European Conference on Computer Systems (EuroSys)*, 2007.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] "Apache Hadoop." [Online]. Available: http://hadoop.apache.org/

[4] H. Karloff, S. Suri, and V. Vassilevska, "A Model of Computation for MapReduce," in *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA)*, M. Charikar, Ed. SIAM, 2010, pp. 938–948.

[5] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[6] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[7] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[8] M. Quinn, "Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 384 –387, mar. 1990.

[9] R. Luling and B. Monien, "Load balancing for distributed branch and bound algorithms," in *Proceedings of the Sixth International Parallel Processing Symposium*, 1992, pp. 543–548.

[10] S. Tschöke, R. Lubling, and B. Monien, "Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network," in *Proceedings of the 9th Parallel Processing Symposium*, 1995, pp. 182 –189.

[11] Y. Shinano, M. Higaki, and R. Hirabayashi, "A generalized utility for parallel branch and bound algorithms," in *IEEE Symposium on Parallel and Distributed Processing*. Los Alamitos, CA, USA: IEEE Computer Society, 1995, p. 392.

[12] S. Tschöke and T. Polzer, "Portable parallel branch-and-bound library PPBB-Lib user manual," Department of Computer Science, University of Paderborn, Tech. Rep., 1996, version 2.0.

[13] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt, "The parallel search bench ZRAM and its applications," *Annals of Operations Research*, vol. 90, pp. 45–63, 1999.

[14] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth, "Masterworker: An enabling framework for applications on the computational grid," *Cluster Computing*, vol. 4, pp. 63–70, 2001.

[15] J. Eckstein, C. A. Phillips, and W. E. Hart, "PICO: An object-oriented framework for parallel branch and bound," in *Proceedings of the Workshop on Inherently Parallel Algorithms in Optimization and Feasibility and their Applications*, ser. Studies in Computational Mathematics. Elsevier Scientific, 2001, pp. 219–265.

[16] Q. Chen and M. C. Ferris, "FATCOP: A fault tolerant Condor-PVM mixed integer programming solver," *SIAM Journal on Optimization*, vol. 11, no. 4, pp. 1019–1036, 2001.

[17] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa, "Mallba: A library of skeletons for combinatorial optimisation," in *Euro-Par 2002 Parallel Processing*, ser. Lecture Notes in Computer Science, B. Monien and R. Feldmann, Eds., vol. 2400. Springer, 2002, pp. 63–73.

[18] R. Lougee-Heimer, "The Common Optimization INterface for Operations Research," *IBM Journal of Research and Development*, vol. 47, pp. 57–66, 2003.

[19] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman, "ALPS: A framework for implementing parallel search algorithms," in *Proceedings of the Ninth INFORMS Computing Society Conference*, 2005, pp. 319–334.

[20] T. K. Ralphs and M. Güzelsoy, "The SYMPHONY callable library for mixed-integer linear programming," in *Proceedings of the Ninth INFORMS Computing Society Conference*, 2005, pp. 61–76.

[21] L. M. A. Drummond, E. Uchoa, A. D. Gonçalves, J. M. N. Silva, M. C. P. Santos, and M. C. S. de Castro, "A grid-enabled distributed branch-and-bound algorithm with application on the Steiner problem in graphs," *Parallel Computing*, vol. 32, no. 9, pp. 629–642, 2006.

[22] A. Djerrah, B. L. Cun, V. D. Cung, and C. Roucairol, "Bob++: Framework for solving optimization problems with branch-and-bound methods," in *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2006, pp. 369–370.

[23] T. Polzin and S. V. Daneshmand, "Improved algorithms for the Steiner problem in networks," *Discrete Applied Mathematics*, vol. 112, no. 1–3, pp. 263–300, 2001.

[24] T. Polzin, "Algorithms for the Steiner problem in networks," Ph.D. dissertation, Universität des Saarlandes, 2003.

[25] M. Poggi de Aragão, E. Uchoa, and R. F. Werneck, "Dual heuristics on the exact solution of large Steiner problems," in *Proc. Brazilian Symposium on Graphs, Algorithms and Combinatorics (GRACO)*, ser. Elec. Notes in Disc. Math., vol. 7, 2001.

[26] E. Uchoa and R. F. Werneck, "Fast local search for steiner trees in graphs," in *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2010, pp. 1–10.

[27] R. Wong, "A dual ascent approach for Steiner tree problems on a directed graph," *Mathematical Programming*, vol. 28, pp. 271–287, 1984.

[28] T. G. Crainic, B. L. Cun, and C. Roucairol, "Parallel branch-and-bound algorithms," in *Parallel Combinatorial Optimization*, E.-G. Talbi, Ed. Wiley, 2006, pp. 1–28.

[29] B. Gendron and T. G. Crainic, "Parallel branch-and-bound algorithms: Survey and synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 1994.

[30] T. Koch, A. Martin, and S. Voß, "SteinLib: An updated library on Steiner tree problems in graphs," Konrad-Zuse-Zentrum für Informationstechnik Berlin, Tech. Rep. ZIB-Report 00-37, 2000. [Online]. Available: http://elib.zib.de/steinlib

[31] P. Toth and D. Vigo, Eds., *The Vehicle Routing Problem*. SIAM, 2001.

[32] G. F. Creţu-Ciocârlie, M. Budiu, and M. Goldszmidt, "Hunting for problems with Artemis," in *USENIX Workshop on the Analysis of System Logs (WASL)*, San Diego, CA, December 7 2008.

[33] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *ACM Symposium on Operating Systems Principles (SOSP)*, J. N. Matthews and T. E. Anderson, Eds. Big Sky, Montana, USA: ACM, October 2009, pp. 261–276.