TASKWORK: A Cloud-aware Runtime System for Elastic Task-parallel HPC Applications

Stefan Kehrer and Wolfgang Blochinger
Parallel and Distributed Computing Group, Reutlingen University, Germany

Keywords: Cloud Computing, High Performance Computing, Task Parallelism, Elasticity of Parallel Computations.

Abstract: With the capability of employing virtually unlimited compute resources, the cloud evolved into an attractive execution environment for applications from the High Performance Computing (HPC) domain. By means of elastic scaling, compute resources can be provisioned and decommissioned at runtime. This gives rise to a new concept in HPC: Elasticity of parallel computations. However, it is still an open research question to which extent HPC applications can benefit from elastic scaling and how to leverage elasticity of parallel computations. In this paper, we discuss how to address these challenges for HPC applications with dynamic task parallelism and present TASKWORK, a cloud-aware runtime system based on our findings. TASKWORK enables the implementation of elastic HPC applications by means of higher-level development frameworks and solves corresponding coordination problems based on Apache ZooKeeper. For evaluation purposes, we discuss a development framework for parallel branch-and-bound based on TASKWORK, show how to implement an elastic HPC application, and report on measurements with respect to parallel efficiency and elastic scaling.

1 INTRODUCTION

The cloud evolved into an attractive execution environment for High Performance Computing (HPC) workloads with benefits such as on-demand access to compute resources and pay-per-use (Netto et al., 2018; Galante et al., 2016). Recently, Amazon Web Services (AWS) and Microsoft Azure introduced new cloud offerings optimized for HPC workloads. Whereas traditional HPC clusters rely on static resource assignment, cloud offerings allow applications to scale elastically, i.e., compute resources can be provisioned and decommissioned at runtime. This gives rise to a new concept in HPC: *Elasticity of parallel computations* (Rajan and Thain, 2017; Galante et al., 2016; Da Rosa Righi et al., 2016; Haussmann et al., 2018). By leveraging elasticity, HPC applications benefit from fine-grained cost control per application run. For instance, processing time and/or the quality of results can be related to costs, allowing versatile optimizations at runtime (Rajan and Thain, 2017; Haussmann et al., 2018). But this novel opportunity comes with many challenges that require research efforts on all levels of parallel systems.

During the last years, there has been a growing interest to make HPC applications cloud-aware (Gupta et al., 2016; Gupta et al., 2013b; Da Rosa Righi et al., 2016; Rajan et al., 2011). In particular, applications have to cope with the effects of virtualization and resource pooling leading to fluctuations in processing times (Gupta et al., 2013b). Existing research also shows how to employ elasticity for HPC applications with simple communication and coordination patterns (e.g., iterative-parallel workloads) (Rajan et al., 2011; Da Rosa Righi et al., 2016). In these cases, problems are iteratively decomposed into a set of independent tasks, which can be farmed out for distributed computation. However, it is still an open research question to which extent other application classes from the field of HPC can benefit from cloud-specific properties, how to leverage elasticity of parallel computations in these cases, and how to ensure cloud-aware coordination of distributed compute resources.

In this paper, we discuss how to address these challenges for HPC applications with dynamic task parallelism. These applications are less sensitive to heterogeneous processing speeds when compared to data-intensive, tightly-coupled HPC applications (Gupta et al., 2016; Gupta et al., 2013a), but comprise unstructured interaction patterns and complex coordination requirements. Prominent examples of this application class include constraint satisfaction solving, graph search, n-body simulations, and raytracing with applications in artificial intelligence, biochem-

Kehrer, S. and Blochinger, W.
TASKWORK: A Cloud-aware Runtime System for Elastic Task-parallel HPC Applications.
DOI: 10.5220/0007795501980209
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 198-209
ISBN: 978-989-758-365-0

istry, electronic design automation, and astrophysics. We discuss the challenges that have to be addressed to make these applications cloud-aware and present TASKWORK - a cloud-aware runtime system that provides a solid foundation for implementing elastic task-parallel HPC applications. In particular, we make the following contributions:

- We present the conceptualization of a cloud-aware runtime system for task-parallel HPC applications.
- We show how to leverage elasticity of parallel computations and how to solve corresponding coordination problems based on Apache ZooKeeper[1].
- We discuss a development framework for implementing elastic branch-and-bound applications with only minor effort required at the programming level.
- We describe the design and implementation of TASKWORK, an integrated runtime system based on our findings, and report on performance experiments in our OpenStack-based cloud environment.

This paper is structured as follows. Sect. 2 describes our conceptualization of a cloud-aware runtime system for task-parallel HPC in the cloud. In Sect. 3, we present TASKWORK - our integated runtime system for elastic task-parallel HPC applications. We elaborate on a branch-and-bound development framework and describe its use in Sect. 4. The results of several experiments related to parallel performance and elastic scaling are presented in Sect. 5. Related work is discussed in Sect. 6. Sect. 7 concludes this work.

2 CONCEPTUALIZATION OF A CLOUD-AWARE RUNTIME SYSTEM

To benefit from cloud-specific characteristics, developing elastic HPC applications is a fundamental problem that has to be solved (Galante et al., 2016). At the core of this problem lies the required dynamic adaptation of parallelism. At all times, the degree of logical parallelism of the application has to fit the physical parallelism given by the number of processing units to achieve maximum efficiency. Traditionally, the number of processing units has been considered as static. In cloud environments, however, the number of processing units can be scaled at runtime by employing cloud management tooling. As a result, applications have to dynamically adapt the degree of logical parallelism based on a dynamically changing physical parallelism. At the same time, adapting the logical parallelism and mapping the logical parallelism to the physical parallelism incurs overhead (in form of excess computation, communication, and idle time). Consequently, elastic HPC applications have to continuously consider a trade-off between the perfect fit of logical and physical parallelism on the one side and minimizing overhead resulting from the adaptation of logical parallelism and its mapping to the physical parallelism on the other. Hence, ensuring elasticity of parallel computations is a hard task as it introduces new sources of overhead and thus leads to many system-level challenges that have to be addressed to ensure a high parallel efficiency.

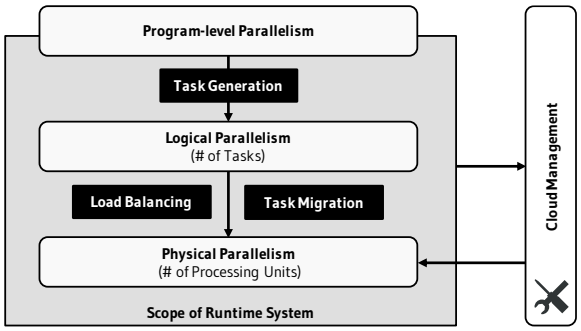

Figure 1: Conceptualization of a cloud-aware runtime system that adapts the logical parallelism, handles load balancing and task migration, and thus enables elastic scaling for task-parallel HPC applications.

Because we specifically focus on dynamic task parallelism, the degree of logical parallelism can be defined as the current number of tasks. We argue that a cloud-aware runtime system is required that transparently controls the parallelism of an HPC application to ensure elastic scaling. Fig. 1 shows our conceptualization of such a runtime system. It allows developers to mark parallelism in the program, automatically adapts the logical parallelism by generating tasks whenever required, and exploits available processing units with maximum efficiency by mapping the logical parallelism to the physical parallelism. An application based on such a runtime system is elastically scalable: Newly added compute nodes automatically receive tasks by means of load balancing and a task migration mechanism releases compute nodes that have been selected for decommissioning (cf. Fig. 1). Our approach is not limited to any specific cloud management approach or tooling: Cloud management may comprise any kind of external decision making logic (e.g., based on processing time, the

[1] https://zookeeper.apache.org.

quality of results, or costs) that finally adapts the number of processing units (i.e., the physical parallelism). An example for such a cloud management approach is given in (Haussmann et al., 2018), where costs are considered to control the physical parallelism. In this work, we focus on elasticity of parallel computations and address related system-level challenges.

Besides elasticity, the characteristics of cloud environments lead to new architectural requirements that have to be considered by HPC applications (Kehrer and Blochinger, 2019). Due to virtualization and resource pooling (leading to CPU timesharing and memory overcommitment), fluctuations in processing times of individual processing units are the common case (Gupta et al., 2016). Thus, in cloud environments processing units should be coupled in a loosely manner by employing asynchronous communication mechanisms (e.g., for load balancing). Similarly, inter-node synchronization should be loosely coupled while guaranteeing individual progress. A runtime system built for the cloud has to provide such asynchronous communication and synchronization mechanisms thus releasing developers from dealing with these low-level complexities.

In this work, we show how to leverage elasticity for applications with dynamic task parallelism. These applications rely on dynamic problem decomposition and thus support the generation of tasks at runtime. Moreover, they are ideal candidates for cloud adoption because they are less sensitive to heterogeneous processing speeds when compared to data-intensive, tightly-coupled HPC applications (Gupta et al., 2016; Gupta et al., 2013a).

3 TASKWORK

In this section, we present TASKWORK, our cloud-aware runtime system for task-parallel HPC applications, designed according to the principles discussed in Sect. 2. TASKWORK comprises several interacting components that enable elasticity of parallel computations (cf. A, Fig. 2) and solve corresponding coordination problems based on ZooKeeper (cf. B, Fig. 2). Based on these system-level foundations, higher-level development frameworks and programming models can be built (cf. C, Fig. 2), which facilitate the implementation of elastic HPC applications.

TASKWORK is implemented in Java and employs distributed memory parallelism by coordinating a set of distributed compute nodes. TASKWORK's components are described in the following. At first, we briefly describe the well-known task pool execution model that we use to manage tasks. Thereafter, we provide details on ZooKeeper, which we employ to solve coordination problems. Finally, we discuss the components of TASKWORK.

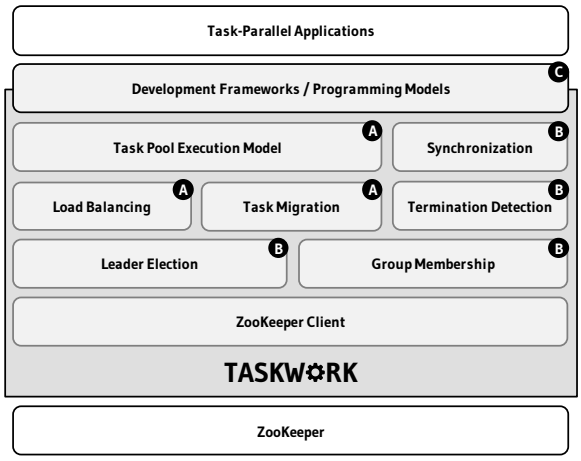

Figure 2: TASKWORK is a cloud-aware runtime system for elastic task-parallel HPC applications. Its components support the construction of higher-level development frameworks and programming models.

3.1 Task Pool Execution Model

The task pool execution model (Grama et al., 2003) decouples task generation and task processing by providing a data structure that can be used to store dynamically generated tasks and to fetch these tasks later for processing. The task pool execution model is fundamental for our runtime system to implement the concepts depicted in Fig. 1: It manages tasks generated at runtime (defining the logical parallelism) and provides an appropriate interface for load balancing and task migration mechanisms that enable elasticity of parallel computations.

In general, a task pool can be implemented following a centralized or a distributed approach. A centralized task pool is located at a single node and accessible by all other nodes. To enable parallel processing, tasks have to be transferred over the network. By following this approach, there is a single instance that has complete knowledge on the state of the system, e.g., which node is executing which tasks. This makes load balancing and coordination simple. However, the centralized approach becomes a sequential bottleneck for a large number of nodes. On the other hand, a distributed implementation leads to multiple task pool instances, local to each node, forming a distributed task pool. The distributed task pool model decouples compute nodes from each other thus leading to a highly scalable and asynchronous system. Further, it enables local accesses to the task pool. On the contrary, coordination becomes a non-trivial task by

following a distributed approach because nodes only have partial knowledge. This specifically holds in cloud environments, where compute nodes are provisioned and decommissioned at runtime.

We favor the distributed task pool model, which, in general, provides a better scalability (Poldner and Kuchen, 2008) and leads to an asynchronous system, thus matching the characteristics of cloud environments. To deal with the drawbacks of the distributed task pool model, we enhance it with scalable coordination and synchronization mechanisms based on ZooKeeper.

3.2 ZooKeeper

ZooKeeper eases the implementation of primitives for coordination, data distribution, synchronization, and meta data management (Hunt et al., 2010). Its interface allows clients to read from and write to a tree-based data structure (consisting of data registers called *znodes*). This data structure is replicated across a set of ZooKeeper servers. Each server accepts client connections and executes requests in FIFO order per client session. Additionally, the API provides so-called *watches*, which enable a client to receive notifications of changes without polling. ZooKeeper guarantees writes to be atomic; read operations are answered locally by each server leading to eventual consistency (Hunt et al., 2010). ZooKeeper's design ensures both high availability of stored data and high-performance data access.

As the cloud is a highly dynamic and distributed execution environment, coordination primitives such as leader election and group membership are essentially required. Based on ZooKeeper, leader election and group membership can be implemented in a straightforward manner (Junqueira and Reed, 2013). However, specific challenges arise in the context of the distributed task pool model: Global variables have to be synchronized across tasks, which imposes additional dependencies, and as tasks can be generated at each node, a termination detection mechanism is required to detect when the computation has been completed. We argue that ZooKeeper provides the means to compensate the missing global knowledge in the distributed task pool model and show how to solve these problems.

3.3 Load Balancing

Load balancing is a fundamental aspect in cloud environments to exploit newly added compute resources efficiently. Moreover, it is a strong requirement of applications with dynamic task parallelism due to dynamic problem decomposition. Load balancing can be accomplished by either sending tasks to other compute nodes (work sharing) or by fetching tasks from other nodes (work stealing) (Blumofe et al., 1996). As the transferral of tasks leads to overhead we favor work (task) stealing because communication is only required when a compute node runs idle. In TASKWORK, load balancing is accomplished by observing changes in the local task pool. Whenever the local task pool is empty and all worker threads are idle, *task stealing* is initiated. Task stealing is an approach where idle nodes send work requests to other nodes in the cluster. These nodes answer the request by sending a task from their local task pool to the remote node.

Because the distributed task pool model lacks knowledge about which compute nodes are busy and which are idling, randomized task stealing has been proven to yield the best performance (Blumofe and Leiserson, 1999). However, in the cloud, the number of compute nodes changes over time. Thus, up-to-date information on the currently available compute nodes is required. This information is provided by the group membership component (cf. Fig. 2). Newly added compute nodes automatically register themselves in ZooKeeper. Changes are obtained by all other compute nodes by means of ZooKeeper *watches* (Hunt et al., 2010).

3.4 Task Migration

To benefit from elasticity, applications also have to deal with the decommissioning of compute resources at runtime. Hence, compute nodes that should be decommissioned have to send unfinished work to remaining nodes. This is ensured by the task migration component. Whenever a compute node should be decommissioned, the task migration component stores the current state of tasks being executed, stops the worker thread, and sends all local tasks to remaining nodes. To support task migration, developers simply specify an optimal interruption point in their program. Therefore, the `migrate` operation can be used to check if a task should be migrated (for an example see Sect. 4.2). TASKWORK employs weak migration of tasks. This means that a serialized state generated from a task object is transferred across the network. To facilitate the migration process, application-specific snapshotting mechanisms can be provided by developers.

3.5 Termination Detection

Traditionally, distributed algorithms (wave-based or based on parental responsibility) have been preferred for termination detection due to their superior scalability characteristics (Grama et al., 2003). However, maintaining a ring (wave-based) or tree (parental responsibility) structure across compute nodes in the context of an elastically scaled distributed system imposes significant overhead. To deal with this issue, we propose a novel termination detection algorithm based on ZooKeeper's design principles.

Termination detection has to consider that tasks are continuously generated at any time and on any compute node in the system. However, an algorithm can make use of the fact that new tasks are always split from existing tasks, ultimately leading to a tree-based task dependency structure. Our termination detection algorithm employs this structure as follows: Every task in the system maintains a list of its children. During the lifecycle of a task, the *taskID* of a child task is appended to this list when it is split from the parent task. Moreover, we make use of a global task list that is updated whenever a task is completed. To initialize this global task list, the *taskID* of the root task is added to the list. At runtime, we update the global task list for every completed task considering its *taskID* as well as the *taskIDs* of all its child tasks. Such an update procedure works as follows: Each *taskID* is either added to the global list if it is not contained in the global list or removed from the global list if it is contained. By following this update procedure, termination can be deduced from an empty global task list because all *taskIDs* are added and removed exactly once: Either by an update procedure triggered after the task has been completed or by an update procedure triggered by the completion of the task's parent task. To ensure that each *taskID* leads to exactly two updates, TASKWORK guarantees that tasks are completed exactly once. Based on this assumption, our termination detection algorithm ensures that the update procedure leads to exactly one create and exactly one delete operation per *taskID*.

In the following, we describe our ZooKeeper-based implementation that is comprised of two parts: One for a coordinator that is determined with the leader election component (cf. Algorithm 1) and one for compute nodes (cf. Algorithm 2). The global task list is stored in ZooKeeper and initialized with the *taskID* of the root task (cf. Algorithm 1, C-1). This is required because the root task has no parent task, i.e., the update procedure with its *taskID* is triggered only once: After the completion of the root task. At runtime, a finite number of update operations add or remove *taskIDs*. Termination can be deduced by the coordinator from an empty global task list. Therefore, a ZooKeeper *watch* is set to receive notifications on changes of the list. C-2 of Algorithm 1 triggers a termination event if all *taskIDs* have been removed.

Algorithm 1: Termination Detection - Coordinator.

```
Task Pool, instance tp.
Application, instance app.
ZooKeeper Client, instance zk.
 1: upon event <Init> do                                          ▷ C-0
 2:     if !zk.exists('/globallist') then
 3:         zk.create('/globallist');
 4:     end if
 5:     if !zk.exists('/txnIDs') then
 6:         zk.create('/txnIDs');
 7:     end if
 8: upon event <tp, RootTaskEnqued | task> do                     ▷ C-1
 9:     zk.getChildren('/globallist');
10:     zk.create('/globallist/' + TaskID(task));
11: upon event <zk, Watch | w> do                                 ▷ C-2
12:     if Type(w) = NodeChildrenChanged then
13:         if zk.getChildren('/globallist') = ∅ then
14:             trigger <app, Terminated>
15:         end if
16:     end if
```

Algorithm 2 implements the mechanisms to update the task list at runtime. An update is executed by the `TrackTasks` procedure, which is called by a worker thread whenever a task is completed (cf. Algorithm 2, CN-0). If a specific *taskID* is contained in the task list, we request the removal of this *taskID* (cf. Algorithm 2, line 12). If a *taskID* is not contained, we request its creation (line 14). To avoid false positive termination decisions, we have to ensure that the update procedure is executed atomically. We employ ZooKeeper transactions to enforce an atomic update of the global task list. This guarantees that termination cannot be detected due to an intermediate system state, in which the global task list is empty until the full update procedure (related to a single completed task) has been executed. As the transaction might fail, e.g., if a *taskID* should be created that has been created by another compute node before our transaction gets processed by ZooKeeper, we pass an asynchronous callback to the *commit* operation that handles potential failures (cf. Algorithm 2, line 17). In the following, we discuss these potential failures.

We constructed the transaction as a set of *create* and *delete* operations. Each *create* operation might fail. In case of such a failure, another compute node added the *taskID* before the local request was processed. Moreover, delayed reads might lead to a falsely constructed create operation. On the other hand, *delete* operations are executed exactly once because they are only requested if the corresponding

Algorithm 2: Termination Detection - Compute Node.

```
Worker Thread, instance wt.
ZooKeeper Client, instance zk.
 1: upon event <wt, completedTask | task> do        ▷ CN-0
 2:     taskIDs ← TaskID(task) ∪ ChildTaskIDs(task);
 3:     TRACKTASKS(TASKIDS, NULL);
 4: procedure TRACKTASKS(taskIDs, txnID)            ▷ CN-1
 5:     txn ← zk.transaction();
 6:     if txnID = null then
 7:         txnID ← generateTxnID();
 8:     end if
 9:     txn.create('/txnIDs/' + txnID);
10:     for each taskID ∈ taskIDs do
11:         if zk.exists('/globallist/' + taskID) then
12:             txn.delete('/globallist/' + taskID);
13:         else
14:             txn.create('/globallist/' + taskID);
15:         end if
16:     end for
17:     txn.commit(TXNCALLBACK, taskIDs, txnID);
18: end procedure
19: procedure TXNCALLBACK(taskIDs, txnID)           ▷ CN-2
20:     if failed ∨ lost connection then
21:         if !zk.exists('/txnIDs/' + txnID) then
22:             TRACKTASKS(TASKIDS, TXNID);
23:         end if
24:     end if
25: end procedure
```

taskID is contained in the list (cf. Algorithm 2, line 11-12). Additionally, connection losses might also lead to a failed transaction and, by default, handling connection losses is a non-trivial task with ZooKeeper (Junqueira and Reed, 2013). There are cases, in which we cannot know if a transaction, which is interrupted by a connection loss, has been processed or not. Simply executing the transaction again could compromise our termination detection algorithm, finally leading to a false positive or missing termination event. In the following, we describe how our algorithm deals with these issues.

CN-2 of Algorithm 2 shows the implementation of the `TxnCallback` procedure, which handles the failures described above. Therefore, transactions that fail are simply retried by calling the `TrackTasks` procedure again. To deal with connection losses, however, we enhanced the presented algorithm to include an additional *create* operation in each transaction constructed by the `TrackTasks` procedure. This operation creates a unique znode */txnIDs/txnID*, where *txnID* is a globally unique ID generated for each transaction (cf. Algorithm 2, line 6-8). This allows us to check if a transaction has been processed or not in case of a connection loss. CN-2 of Algorithm 2 employs the *txnID* to avoid a repeated execution of committed transactions thus making them idempotent. Even if the *txnID* exists but we read an outdated ZooKeeper server state (in which it seems to be nonexistent), the *txnID* passed to the `TrackTasks` procedure avoids a repeated commit thus finally maintaining a consistent system state. As ZooKeeper guarantees that clients read up-to-date values within a certain time bound, retrying a commit of a transaction with an existing *txnID* stops eventually (cf. Algorithm 2, CN-2).

Our termination detection algorithm makes use of ZooKeeper's design principles by employing (1) *watches* and notifications to be informed on relevant updates to the system state, (2) fast read operations for ensuring high-performance transaction construction, (3) asynchronous execution of operations for non-blocking system behavior[2], and (4) ZooKeeper's atomic writes and transactions in conjunction with TASKWORK's exactly once completion to ensure a consistent system state.

3.6 Synchronization of Global Variables

Many non-trivial task-parallel applications require the synchronization of global variables across tasks at runtime. This synchronization leads to additional task interaction patterns that the runtime system has to cope with. TASKWORK's synchronization component allows developers to easily define global variables that are transparently shared across tasks.

Synchronization considers three hierarchy levels: (1) task-level variables, which are updated for each task executed by a worker thread, (2) node-level variables, which are updated on each compute node, and (3) global variables. Task-level variables are typically updated by the implemented program and thus managed by the developer. To synchronize node-level variables, we provide two operations: `getVar` for obtaining node-level variables and `setVar` for setting node-level variables. Whenever a node-level variable changes its value, we employ ZooKeeper to update this variable globally, which enables synchronization across all distributed compute nodes. These generic operations allow developers to address application-specific synchronization requirements, while TASKWORK handles the process of synchronization.

3.7 Development Frameworks

TASKWORK is specifically designed for applications with dynamic task parallelism and provides a generic

[2]Note that we employed ZooKeeper's asynchronous API for all interactions with ZooKeeper. In some cases, we used a synchronous-looking notation in the algorithmic descriptions only for the purpose of better readability.

task abstraction that allows the specification of custom task definitions. As described in Sect. 2, developers only mark program-level parallelism while task generation, load balancing, and task migration are handled automatically thus ensuring elasticity of parallel computations. Therefore, developers specify an application-specific `split` operation based on the generic task abstraction to split work from an existing task. Afterwards, this `split` operation can be used for implementing any application program that dynamically creates tasks at runtime (for an example see Sect. 4.2). To enable elasticity of parallel computations, TASKWORK provides an execution mode called *potential splitting* that adapts the logical parallelism (number of tasks) in an automated manner. In this mode, developers also implement the `split` operation, but only specify a potential splitting point in their application program with the `potentialSplit` operation. Thus, the `potentialSplit` operation is used to mark program-level parallelism. At runtime, TASKWORK decides whether to create new tasks or not depending on the current system load. Potential splitting automatically adapts the amount of tasks generated and thus controls the logical parallelism of the application (cf. Fig. 1). As a result, TASKWORK manages the trade-off between perfect fit of logical and physical parallelism and minimizing overhead resulting from task generation and task mapping as discussed in Sect. 2.

4 BRANCH-AND-BOUND DEVELOPMENT FRAMEWORK

We describe a development framework for parallel branch-and-bound applications based on TASKWORK. Branch-and-bound is a well-known meta-algorithm for solving search and optimization problems with numerous applications in biochemistry, pattern recognition, finite geometry, model checking, and fleet and vehicle scheduling. In the following, we briefly explain the branch-and-bound approach and employ our framework to develop an example application.

4.1 Branch-and-Bound Applications and the TSP

We explain the branch-and-bound approach by employing the Traveling Salesman Problem (TSP) as example application. We chose the TSP as a representative application due to its wide use in research and

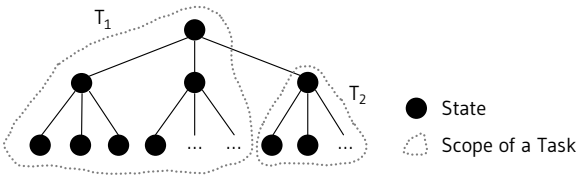

Figure 3: For parallelization, we cut the state space tree of a problem into several tasks, each capturing a subproblem of the initial problem. However, because states in the state space tree are explored as the computation proceeds, tasks containing these states have to be created at runtime to avoid load imbalance.

industry. The TSP states that a salesman has to make a tour visiting n cities exactly once while finishing at the city he starts from. The problem can be modeled as a complete graph with n vertices, where each vertex represents a city and each edge a path between two cities. A nonnegative cost $c(i, j)$ occurs to travel from city i to city j. The optimization goal is to find a tour whose total cost, i.e., the sum of individual costs along the paths, is minimum (Cormen et al., 2009).

All feasible tours can be explored systematically by employing a *state space tree* that enumerates all states of the problem. The initial state (root of the state space tree) is represented by the city the salesman starts from. From this (and all following states), the salesman can follow any path to travel to one of the unvisited cities, which leads to a new state. At some point in time, all cities have been visited thus leading to a leaf state, which represents a tour. Each state can be evaluated with respect to its cost by summing up the individual costs of all paths taken. This also holds for leaf states representing a tour. A search procedure can be applied that dynamically explores the complete state space tree and finally finds a tour with minimum cost. However, brute force search cannot be applied to large state space trees efficiently. Instead of enumerating all possible states, branch-and-bound makes use of existing knowledge to search many paths in the state space tree only implicitly. The following four operations enable an efficient search procedure:

Branching: If the current state is not a leaf state, the next level of child states is generated by visiting all unvisited cities that are directly accessible. Each of these child states leads to a set of disjoint tours.

Evaluation: If the current state is a leaf state, we evaluate the tour represented by this state with respect to its total cost.

Bounding: At runtime, the tour whose total cost is known to be minimum at a specific point in time defines an upper bound for the ongoing search procedure. Any intermediate state in the state space tree that evaluates to a higher cost can be proven to lead

to a tour with higher total costs and thus has not to be explored any further. Moreover, lower bounds can be calculated by solving a relaxed version of the problem based on the current state (Sedgewick, 1984).

Pruning: We can prune parts of the state space tree if the calculated lower bound of the current state is larger or equal to the current upper bound. The pruning operation is essential to make branch-and-bound efficient.

Following a branch-and-bound approach, a problem is decomposed into subproblems at runtime. Each of these subproblems captures several states of the state space tree and can be solved in parallel. Technically, these subproblems are captured in a set of tasks, which can be distributed across available compute nodes. However, several challenges arise when we map branch-and-bound applications to parallel and distributed architectures: Pruning introduces work anomalies, which means that the amount of work differs between sequential and parallel processing as well as across parallel program runs due to non-determinism of message delivery. Additionally, the workload of branch-and-bound applications is highly irregular, i.e., task sizes are not known a priori and unpredictable by nature. Consequently, solving the TSP requires the runtime system to cope with dynamic problem decomposition and load balancing to avoid idle processors. Every task that captures a specific subproblem can produce new child tasks (cf. Fig. 3). Thus, termination detection is another strong requirement to detect if a computation has been completed. Additionally, updates on the upper bound have to be distributed fast to enable efficient pruning for subproblems processed simultaneously in the distributed system.

4.2 Design and Use of the Development Framework

In the following, we describe a development framework for parallel branch-and-bound on top of TASKWORK. We employ the TSP as an example application to show how to use the framework. Parallel applications can be built with this framework without considering low-level, technical details.

TASKWORK provides a generic task abstraction that can be used to build new development frameworks. In this context, we define a task as the traversal of the subtrees rooted at all unvisited input states. Additionally, each task has access to the graph structure describing the cities as vertices and the paths as edges. This graph structure guides the exploratory construction of the state space tree. All visited states are marked in the graph. This representation allows

```
1  public void search() {
2    while(!openStates.isEmpty()){
3      if(migrate()) return;
4
5      State currentState = openStates.getNext();
6
7      getUpperBound();
8
9      State[] children = currentState.branch();
10     for(State child : children) {
11       if(child.isLeafState()) {
12         if(child.getCost() < current_best_cost){
13           current_best_cost = child.getCost();
14           current_best_tour = child.getPath();
15           setUpperBound();
16         }
17       }else if(child.getLowerBound() < current_best_cost){
18         openStates.add(child);
19       }
20     }
21
22     potentialSplit();
23   }
24 }
```

Figure 4: The branch-and-bound development framework allows developers to implement parallel search procedures without considering low-level details such as concurrency, load balancing, synchronization, and task migration.

to split the currently traversed state space tree to generate new tasks.

New tasks have to be created at runtime to keep idling processors (and newly added ones) busy. Therefore, the branch-and-bound task definition allows the specification of an application-specific `split` operation. This operation branches the state space tree by splitting off a new task from a currently executed task. This split-off task can be processed by another worker thread running on another compute node. To limit the amount of tasks generated, we make use of TASKWORK's potential splits, i.e, the `split` operation is only triggered, when new tasks are actually required. As depicted in Fig. 4, here, the `potentialSplit` operation is executed after a state has been evaluated. TASKWORK decides if a split is required. If so, it executes the application-specific `split` operation that takes tasks from the `openStates` list to create a new (disjoint) task. Otherwise it proceeds normally, i.e., it evaluates the next state in the state space tree.

Task Migration: To enable task migration, developers check if migration is required (cf. Fig. 4). In this case, a task simply stops its execution. The migration process itself is handled by TASKWORK (cf. Sect. 3.4).

Bound Synchronization: Pruning is based on a global upper bound. In case of the TSP, the total cost of the best tour currently known is used as the global upper bound. The distribution of the current upper bound is essential to eliminate excess computation (due to an outdated value). By employing the synchronization component (cf. Sect. 3.6), we initiate an update of the global upper bound when-

Table 1: Performance measurements of TSP instances.

Problem Instance	T_s [s] (1 VM)	T_p [s] (60 VMs)	Speedup (60 VMs)	Efficiency (60 VMs)
$TSP35_1$	1195	32.9 ± 2.0	36.3	0.60
$TSP35_2$	1231	55.7 ± 4.0	22.1	0.37
$TSP35_3$	2483	103.5 ± 2.1	24.0	0.40
$TSP35_4$	3349	115.5 ± 6.3	29.0	0.48
$TSP35_5$	10286	167.4 ± 12.4	59.5	0.99

ever the local upper bound is better then the current global upper bound observed. Technically, we specify an update rule that compares the total costs of two tours. If a better upper bound has been detected, TASKWORK ensures that the new upper bound is propagated through the hierarchy levels. `getUpperBound` and `setUpperBound` (cf. Fig. 4) are implemented based on the `getVar` and `setVar` operations (cf. Sect. 3.6).

Termination Detection: Termination detection is transparently handled by TASKWORK.

5 EVALUATION

Our evaluation is threefold: (1) We report on the parallel performance and scalability by measuring speedups and efficiencies. (2) We measure the effects of elastic scaling on the speedup of an application. (3) Moreover, because TASKWORK heavily relies on ZooKeeper, we provide several results that show that ZooKeeper fits our architectural requirements. For evaluation purposes, we employ the TSP application implemented with the parallel branch-and-bound development framework.

Setup. TASKWORK compute nodes are operated on CentOS 7 virtual machines (VM) with 1 vCPU clocked at 2.6 GHz, 2 GB RAM, and 40 GB disk. All VMs are deployed in our OpenStack-based cloud environment. The underlying hardware consists of identical servers, each equipped with two Intel Xeon E5-2650v2 CPUs and 128 GB RAM. The virtual network connecting tenant VMs is operated on a 10 GBit/s physical ethernet network. Each compute node runs a worker thread and is connected to one of three ZooKeeper servers (forming a ZooKeeper cluster). Our experiments were performed during regular multi-tenant operation.

Parallel Performance & Scalability. To evaluate the parallel performance, we solved 5 randomly generated instances of the 35 city symmetric TSP. Speedups and efficiencies are based on the runtime T_s of a sequential implementation executed by a single thread on the same VM type. We calculate the lower bound (cf. Sect. 4.1) by adding the weight of a minimum spanning tree (MST) of the not-yet visited cities to the current path (Sedgewick, 1984; Archibald et al., 2018). The MST itself is calculated based on Prim's algorithm (Prim, 1957). Table 1 shows the results of our measurements with three parallel program runs per TSP instance. As we can see, the measured performance is highly problem-specific. Note that, as mentioned in Sect. 4.1, the pruning operation results in work anomalies, which means that the total amount of work differs significantly between sequential and parallel processing as well as across parallel program runs thus rendering a systematic evaluation of the system's scalability infeasible. To deal with this problem, we disabled pruning for our scalability measurements. Fig. 5 shows the measured scalability for two TSP instances with 14 and 15 cities with a sequential runtime T_s of 1170 and 16959 seconds, respectively.

Elastic Scaling. Compute resources can be provisioned or decommissioned easily by employing cloud management tooling. However, to leverage elasticity of parallel computations, the fundamental question is: *How fast* can resources be effectively employed by an HPC application? This is a completely new viewpoint from which cloud-aware parallel systems have to be evaluated. Traditional HPC metrics do not cover this temporal aspect of elasticity. In the following, we present an experiment that shows the capability of TASKWORK to dynamically adapt to a changing number of resources while maximizing the speedup of the computation. To avoid work anomalies, we disabled pruning to evaluate elastic scaling. All measurements are based on the TSP instance with 14 cities.

Our experiment is described in Fig. 6 and com-

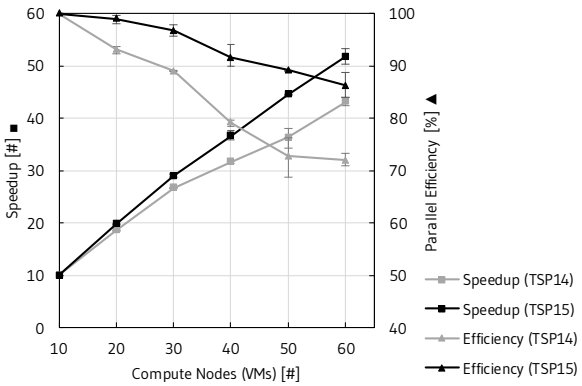

Figure 5: Speedups and efficiencies given are arithmetic means based on 3 parallel program runs for 6 setups leading to 36 measurements in total. The highest and lowest measured values per setup are depicted accordingly.

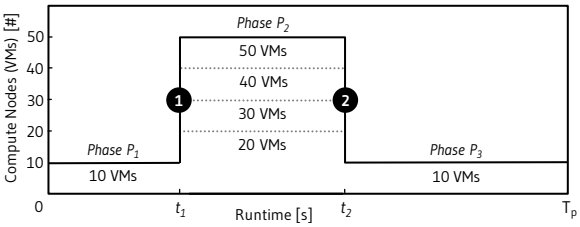

Figure 6: To evaluate the elasticity of parallel computations, we adapt the physical parallelism (given by the compute resources employed) at times t_1 and t_2 and measure the effects on the speedup S_E.

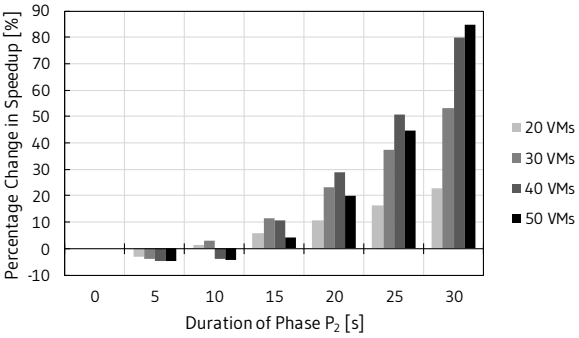

Figure 7: We calculated the percentage change in speedup based on different durations of Phase P_2 and different numbers of VMs added to the computation. The legend shows the total number of VMs employed in Phase P_2.

prises three phases. We start our application with 10 compute nodes (VMs) in Phase P_1. At time t_1, we scale out by adding more VMs to the computation. To abstract from platform-specific VM startup times, we employ VMs that are already running. At time t_2, we decommission the VMs added at t_1. At phase transition ❶, TASKWORK ensures task generation and efficient load balancing to exploit newly added compute nodes. At phase transition ❷, the task migration component ensures graceful decommissioning of compute nodes (cf. Sect. 3.4). We can easily see if newly added compute resources contribute to the computation by comparing the measured speedup S_E (speedup with elastic scaling) with the *baseline speedup* S_{10} that we measured for a static setting with 10 VMs. To see how effectively new resources are employed by TASKWORK, we tested several durations for Phase P_2 as well as different numbers of VMs added (cf. Fig. 6) and calculated the percentage change in speedup $S_\%$ as follows:

$$S_\% = \frac{S_E - S_{10}}{S_{10}} \cdot 100 \qquad (1)$$

$S_\%$ allows us to quantify the relative speedup improvements based on elastic scaling. Both S_E and S_{10} are arithmetic means calculated based on three program runs. Fig. 7 shows the percentage change in speedup achieved for different durations of Phase P_2 and different numbers of VMs added to the computation at runtime. 40 VMs added (leading to 50 VMs in total) can be effectively employed in 15 seconds. Higher speedup improvements can be achieved by increasing the duration of Phase P_2. We also see that for a duration of 10 seconds, adding 40 VMs even leads to a decrease in speedup whereas adding 20 VMs leads to an increase in speedup (for the same duration). This effect results from the higher parallel overhead (in form of task generation, load balancing, and task migration) related to adding a higher number of VMs. On the other hand, as expected, for higher durations of Phase P_2, employing a higher number of VMs leads to better speedups. Note that the percentage of time spent in Phase P_2 (with respect to the total runtime) affects the actual percentage change in speedup, but not the effects that we have described.

ZooKeeper. Because TASKWORK heavily relies on ZooKeeper, we evaluate two fundamental aspects of our implementation: Its read/write ratio and the usefulness of our termination detection algorithm. For the termination detection algorithm, we measured a read/write ratio of 2.6:1 on average. For group membership, load balancing, and variable synchronization each write operation leads to n reads, where n is the number of compute nodes. Consequently, our workload fits ZooKeeper's design principles as it is built for systems with a read/write ratio of at least 2:1 (Hunt et al., 2010). Further, we evaluate our termination detection algorithm. For the termination detection algorithm, performance degradation results from retrying to many operations due to contention thus leading to excess computation. Hence, in our case, we have to ensure that retrying transactions does not dominate the execution time of the algorithm (cf. Sect. 3.5). Additionally, replication of data across ZooKeeper servers amplifies this problem by providing a worse data accuracy. However, for our example application, we measured that roughly 2% of all transactions have been retried on average, which is a negligible overhead and thus emphasizes the advantage of our implementation.

6 RELATED WORK

It has been widely recognized that HPC applications have to be adapted towards cloud-specific characteristics to benefit from on-demand resource access, elasticity, and pay-per-use (Netto et al., 2018; Rajan et al., 2011; Parashar et al., 2013; Rajan and Thain, 2017). The authors of (Galante et al., 2016) propose the de-

velopment of new frameworks for building HPC applications optimized for cloud environments and discuss the importance of application support with respect to elasticity. We follow this approach by presenting a runtime system that does most of the heavy lifting to implement cloud-aware HPC applications.

In (Gupta et al., 2016) an in-depth performance analysis of different HPC applications is presented. Based on their measurements, the authors describe several strategies to make HPC applications cloud-aware. A major issue is the specification of the optimal task size to balance various sources of overhead.

The authors of (Gupta et al., 2013b) address the problem of fluctuations in processing times, which specifically affects tightly-coupled HPC applications. A dynamic load balancer is introduced that continuously monitors the load of each vCPU and reacts to a measured imbalance. Whereas the authors rely on overdecomposition to ensure dynamic load balancing, our runtime system actively controls the logical parallelism of an application to minimize task management overhead. However, it is still an open research question if applications without dynamic task parallelism can benefit from such an approach.

In (Da Rosa Righi et al., 2016), the authors enable elasticity for iterative-parallel applications by employing a master/worker architecture. They make use of an asynchronous elasticity mechanism that enables scaling operations without blocking the application. Whereas we specifically consider dynamic task parallelism, our runtime system also makes use of asynchronous scaling operations. Moreover, our runtime system handles communication and synchronization asynchronously to hide network latencies and fluctuations in processing times.

Task-based parallelism was originally designed to exploit shared memory architectures and used by systems such as Cilk (Blumofe et al., 1996). A major characteristic of task-parallel approaches is that tasks can be assigned dynamically to worker threads, which ensures load balancing and thus effectively reduces idle time. This approach also provides attractive advantages beyond shared memory architectures and has been adopted for different environments including clusters (Archibald et al., 2018) and grids (Anstreicher et al., 2002). As a result, the distributed task pool model has been actively researched. The authors of (Poldner and Kuchen, 2008) present a skeleton for C++, which supports distributed memory parallelism for branch-and-bound applications. Their skeleton uses MPI communication mechanisms. The authors of (Cunningham et al., 2014) propose a termination detection mechanism based on ZooKeeper. However, in contrast to our algorithm, their approach employs the synchronous API of ZooKeeper. COHESION is a microkernel-based platform for desktop grid computing (Schulz et al., 2008; Blochinger et al., 2006). It has been designed for task-parallel problems with dynamic problem decomposition and tackles the challenges of desktop grids such as limited connectivity. In this work, we present an approach to enable elastic task-parallelism in cloud environments.

7 CONCLUSION AND FUTURE WORK

In this work, we addressed several system-level challenges related to task-parallel HPC in the cloud and presented a novel runtime system that manages the complexities of cloud-aware applications. We showed how to solve several coordination problems with ZooKeeper and enhanced the well-known distributed task pool execution model. Elasticity of parallel computations is enabled by means of load balancing, task migration, and application-specific task generation, which requires only minor effort at the programming level. Whereas our development framework is specifically designed for branch-and-bound applications, other applications with dynamic task parallelism such as n-body simulations (Hannak et al., 2012) might also benefit from TASKWORK's architecture.

Many research challenges are left on the path towards cloud-aware HPC applications. Most interestingly, it is not fully understood how to deal with monetary costs of parallel computations in the cloud. To ultimately benefit from elastic scaling and pay-per-use, new cost models are required. Specifically, irregular task-parallel applications with unpredictable resource requirements may benefit from elasticity. In this context, we are confident that TASKWORK provides a solid foundation for future research activities. We also plan to investigate container virtualization for deploying HPC applications to cloud environments (Kehrer and Blochinger, 2018b; Kehrer and Blochinger, 2018a).

ACKNOWLEDGEMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program *Services Computing*.

REFERENCES

Anstreicher, K., Brixius, N., Goux, J.-P., and Linderoth, J. (2002). Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3):563–588.

Archibald, B., Maier, P., McCreesh, C., Stewart, R., and Trinder, P. (2018). Replicable parallel branch and bound search. *Journal of Parallel and Distributed Computing*, 113:92 – 114.

Blochinger, W., Dangelmayr, C., and Schulz, S. (2006). Aspect-oriented parallel discrete optimization on the cohesion desktop grid platform. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 49–56.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1996). Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69.

Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). Introduction to algorithms (3rd ed.).

Cunningham, D., Grove, D., Herta, B., Iyengar, A., Kawachiya, K., Murata, H., Saraswat, V., Takeuchi, M., and Tardieu, O. (2014). Resilient x10: Efficient failure-aware programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 67–80, New York, NY, USA. ACM.

Da Rosa Righi, R., Rodrigues, V. F., Da Costa, C. A., Galante, G., De Bona, L. C. E., and Ferreto, T. (2016). Autoelastic: Automatic resource elasticity for high performance applications in the cloud. *IEEE Transactions on Cloud Computing*, 4(1):6–19.

Galante, G., De Bona, L. C. E., Mury, A. R., Schulze, B., and Da Rosa Righi, R. (2016). An analysis of public clouds elasticity in the execution of scientific applications: a survey. *Journal of Grid Computing*, 14(2):193–216.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Pearson Education, second edition.

Gupta, A., Faraboschi, P., Gioachin, F., Kale, L. V., Kaufmann, R., Lee, B., March, V., Milojicic, D., and Suen, C. H. (2016). Evaluating and improving the performance and scheduling of hpc applications in cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321.

Gupta, A., Kale, L. V., Gioachin, F., March, V., Suen, C. H., Lee, B. S., Faraboschi, P., Kaufmann, R., and Milojicic, D. (2013a). The who, what, why, and how of high performance computing in the cloud. In *IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 306–314.

Gupta, A., Sarood, O., Kale, L. V., and Milojicic, D. (2013b). Improving hpc application performance in cloud through dynamic load balancing. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 402–409.

Hannak, H., Blochinger, W., and Trieflinger, S. (2012). A desktop grid enabled parallel barnes-hut algorithm. In *2012 IEEE 31st International Performance Computing and Communications Conference*, pages 120–129.

Haussmann, J., Blochinger, W., and Kuechlin, W. (2018). Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. *Cluster Computing*.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA.

Junqueira, F. and Reed, B. (2013). *ZooKeeper: distributed process coordination*. O'Reilly Media, Inc.

Kehrer, S. and Blochinger, W. (2018a). Autogenic: Automated generation of self-configuring microservices. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 35–46. INSTICC, SciTePress.

Kehrer, S. and Blochinger, W. (2018b). Tosca-based container orchestration on mesos. *Computer Science - Research and Development*, 33(3):305–316.

Kehrer, S. and Blochinger, W. (2019). Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions. *SICS Software-Intensive Cyber-Physical Systems*.

Netto, M. A. S., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L. F., and Buyya, R. (2018). Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Computing Surveys (CSUR)*, 51(1):8:1–8:29.

Parashar, M., AbdelBaky, M., Rodero, I., and Devarakonda, A. (2013). Cloud paradigms and practices for computational and data-enabled science and engineering. *Computing in Science Engineering*, 15(4):10–18.

Poldner, M. and Kuchen, H. (2008). Algorithmic skeletons for branch and bound. In Filipe, J., Shishkov, B., and Helfert, M., editors, *Software and Data Technologies*, pages 204–219, Berlin, Heidelberg. Springer.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401.

Rajan, D., Canino, A., Izaguirre, J. A., and Thain, D. (2011). Converting a high performance application to an elastic cloud application. In *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 383–390.

Rajan, D. and Thain, D. (2017). Designing self-tuning split-map-merge applications for high cost-efficiency in the cloud. *IEEE Transactions on Cloud Computing*, 5(2):303–316.

Schulz, S., Blochinger, W., Held, M., and Dangelmayr, C. (2008). Cohesion — a microkernel based desktop grid platform for irregular task-parallel applications. *Future Generation Computer Systems*, 24(5):354 – 370.

Sedgewick, R. (1984). *Algorithms*. Addison-Wesley Publishing Co., Inc., Boston, MA, USA.