SCHEDULING HEURISTICS FOR EXECUTING SCIENTIFIC WORKFLOWS ON HOMOGENEOUS CLUSTERS WITH GLOBALLY- AND LOCALLY-ACCESSIBLE PERSISTENT STORAGE

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

MAY 2018

By

Suraj Pandey

Thesis Committee:

Henri Casanova, Chairperson
Nodar Sitchinava
Lipyeow Lim

Keywords: Scientific Workflows, DAG, Task Replication, Local/Global Storages, Heuristics

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Henri Casanova, for providing guidance and feedback throughout this research. Without his guidance and support, I would not have been able to complete this research. I would also like to thank Prof. Nodar Sitchinava and Prof. Lipyeow Lim for being on my thesis committe. I would also like to express my thankfulness to all the faculty members of Information and Computer Sciences Department at UH Manoa whose direct/indirect guidance helped me to complete my Masters Degree.

ABSTRACT

Many applications in science and engineering today are structured as scientific workflows, i.e., task graphs with data dependencies between graphs, where tasks are implemented as standalone executables and data dependencies are via files read/written from/to stable storage. For many relevant application domains, these workflows are both large and data-intensive. Therefore, optimizing data accesses is crucial for efficient scientific workflow executions. Typical HPC (High Performance Computing) platforms used to run scientific workflows are commodity clusters, in which each compute node has access to private, small, high-bandwidth "local" storage, and to shared, large, low-bandwidth "global" storage. To date, production Workflow Management Systems (WMs), software infrastructures for executing workflows in practice, only use global storage. There is thus an opportunity to improve workflow performance by exploiting local storage. The difficulty, however, is twofold. First, the capacity of local storage is limited and often allows holding only a few workflow files. Second, storing data in local storage reduces parallelism because storage is private to a single node. In this thesis, we design scheduling heuristics to orchestrate workflow execution in this context, with the objective of minimizing workflow execution time. These heuristics decide which files should be stored in which level of storage (local or global) and replicate tasks so as to increase the availability of data across compute nodes and thus maintain parallelism. We implement a simulation framework to evaluate and drive the design of these heuristics using both real-world and synthetic workflow configurations. We also implement a software prototype for using these heuristics on HPC platforms. From experimental results obtained in simulation and on an actual HPC cluster we are able to evaluate the relative merit of our heuristics and draw conclusions about the most promising approaches and remaining challenges.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Scientific workflows have become mainstream for conducting large-scale scientific research [30]. Workflows allow scientists to express multi-step computational tasks, for example: retrieve data from an instrument or a database, reformat the data, run analyses, and post-process results. Astronomers are using workflows to generate science-grade mosaics of the sky [6, 4, 27] and to search for extrasolar planets using data collected by NASA's Kepler mission [5, 33]. The Laser Interferometer Gravitational-Wave Observatory (LIGO) uses workflows to search for binary inspiral gravitational waves [9]. Earthquake scientists use workflows to develop shakemaps of Southern California [11, 10, 12]. Researchers in bioinformatics have embraced workflows for protein folding [14], DNA and RNA sequencing [7, 21, 24], and disease-related research [19, 20]. Given the above and countless other efforts, efficient execution of workflow applications is crucial for scientific advances. These workflow applications consist of computational tasks, provided as opaque executables, that take input from files and produce output to files. Tasks thus have data dependencies, and a workflow can be seen as a Directed Acyclic Graph (DAG) in which vertices are tasks and edges are data dependencies. .

The most common High Performance Computing (HPC) platform on which workflow applications are executed are clusters: (large) numbers of compute hosts (or nodes, servers) interconnected via a fast network. Many of these clusters are commodity clusters, i.e., they use commodity hosts and commodity interconnects. As of November 2017, 87.4% of platforms on the Top500 list [31] are commodity clusters. In all these clusters, each compute host is connected to a *global storage* system with enormous capacity. There are various ways in which this connection is implemented (e.g., a separate "storage area network", dedicated "I/O nodes"). But conceptually, the compute hosts all share some bandwidth to the storage system. In addition, in many clusters, each compute host also is connected to a *local storage* device (i.e., a disk). The bandwidth to this local storage is typically orders of magnitude higher than that to the global storage. However, the local storage has much smaller capacity and is accessible only by one compute host.

Executing scientific workflows on HPC platforms, and thus on HPC clusters, is done via software infrastructures termed Workflow Management Systems (WMSs). Several WMSs have been developed [16, 15, 18, 36, 37, 3, 2] that allow scientists to construct and execute workflows on a broad range of software/hardware stacks. These systems, to date, when executing a workflow on an HPC cluster, only use global storage, i.e., all input/output files are read/written from/to global storage. Increasingly, workflow applications are being constructed that are data-intensive. In other words, when executed on a cluster and using global storage only, the execution time of these applications can be dominated by I/O time rather than compute time (in part due to the performance gap between I/O systems and processors).

In this thesis, we explore how local storage in HPC clusters could be used effectively to improve the performance of data-intensive scientific workflows. Even though local storage capacity is limited, due to its high bandwidth placing even only a few files in local storage judiciously could reduce I/O time significantly. More specifically, we wish to solve an off-line scheduling problem in which for each workflow task we must decide when and on which host to execute it, and for each file it produces whether this file should be written to global storage (which is always possible) or to local storage (which is not always possible). The objective is to minimize application execution time, or *makespan*.

Besides the challenge of limited capacity, use of local storage can actually increase makespan, in spite of higher I/O bandwidth. This is because using local storage can lead to *reduction in parallelism*. This occurs when independent ready tasks cannot execute concurrently on multiple available compute hosts because each task requires some input that is only available in the local storage of the same host. The execution of these tasks is then serialized, unless necessary files are copied from local storage to global storage, which not always practical in current HPC clusters. The way in which we mitigate this challenge, so that we can exploit high local storage bandwidth, is by using *task replication*. Executing replicas of a task onto multiple compute hosts allows multiple copies of its output files to be stored in local storage at those compute hosts. Consequently, this task's children tasks will be able to execute concurrently on those compute hosts.

Unsurprisingly, the above scheduling problem is NP-hard. We thus propose polynomial-time heuristics for scheduling tasks, deciding how to replicate tasks, and deciding where output files should be written, with the objective of minimizing overall application execution time, or *makespan*. We then evaluate these heuristics, and in particular compare them to the "use only global storage" approach implemented in current WMSs, for representative workflow applications both in simulation and on a real-world testbed. Based on experimental results, we draw conclusions about the potential performance benefits of using local storage, and about which heuristic ideas seem most promising.

This thesis is organized as follows. In Chapter 2 we formally define our problem, in terms of application, platform, and objective, and state and justify our assumptions. In Chapter 3, we discuss relevant related work. In Chapter 4, we present our proposed heuristics. In chapter 5, we discuss simulation and real-world experimental results. Finally, in Chapter 6 we summarize our findings and discuss relevant future work directions.

CHAPTER 2 PROBLEM DEFINITION

In this chapter we define our target scheduling problem. We describe and justify our platform model (Section 2.1), application model (Section 2.2), and execution model (Section 2.3). We then define our objective function (Section 2.4).

2.1 Platform Model

We consider a compute platform with h identical hosts: $H_1, \ldots, H_h$. Each host H_i has access to a *local* (i.e., private) storage device with I/O bandwidth b (in bytes/sec) and capacity C (in bytes). For simplicity we assume that the I/O bandwidth is the same for reading and writing. In practice these may differ, but it is straightforward to extend our work to account for different read and write bandwidths. Each host also has access to a *global* (i.e., shared) storage system with I/O bandwidth B and capacity ∞. Here again we assume identical read and write bandwidths. This model is representative of most HPC cluster platforms.

Because global storage is shared, hosts may contend for global storage bandwidth. Ideally, real-world clusters would provide dedicated network links from each host to the global storage. In the worst case, there could could be a single (bottleneck) such link. Some clusters in practice provide in-between solutions in which some number of hosts may read/write data from global storage without decreases in bandwidth (e.g., due to the use of dedicated I/O nodes, due to the use of storage area networks). Since the degree of global storage bandwidth sharing among the hosts can vary, to keep this work general we simply define a number of concurrent full-bandwidth connections that can be supported: *num_conns*. For instance, if *num_conns* $= 3$, then 3 hosts could each read/write data concurrently with transfer rate B. If a 4-th hosts reads/writes data, then each host would experience a 3 $B/4$ bandwidth (i.e., hosts share the bandwidth and at most the bandwidth is *num_conns* $* B$).

We actually ran simple "concurrent access to global storage" benchmark on two real-world clusters: the Catalyst cluster at Lawrence Livermore National Laboratory [1] and the UHHPC Cray cluster at the University of Hawaiʻi at Mānoa [2]. In both cases, benchmark results show that *num_conns* $= 1$ (i.e., the "worst-case scenario" mentioned above).

We assume that $B < b$. On the Catalyst cluster mentioned above, simple benchmarking show that b/B is around 20. To be specific, on the Catalyst cluster, simple benchmarking

[1] `https://computation.llnl.gov/computers/catalyst`

[2] `https://www.hawaii.edu/its/ci/hpc-resources/`

tests show that b is around 2GB/s and B is around 100MB/s. On the UHHPC cluster above, b/B is around a factor 4. To be specific, on the UHHPC cluster, b is around 400MBps and B is around 100MBps. In general, this factor could be very large, e.g., if local storage is on a Solid State Drive (SSD). Note that we ignore storage latencies, and simply compute the time to read or write s bytes of data as s divided by the bandwidth. Our target applications are data-intensive workflows that read/write large files and I/O time is thus bandwidth-bound.

A simple illustration of our platform model is shown in Figure 2.1.

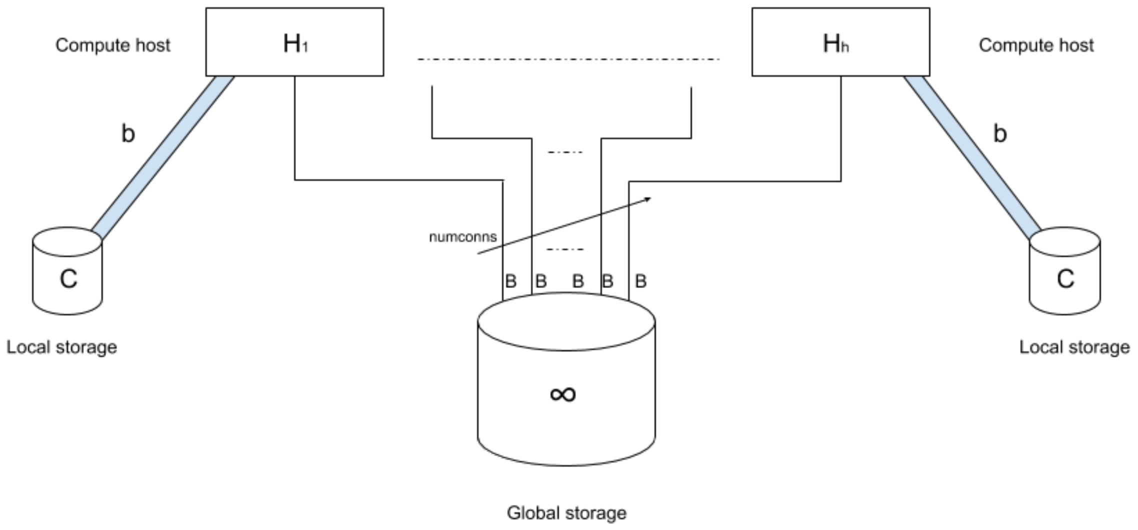

Figure 2.1: Platform Model

2.2 Application Model

We consider a workflow application with t tasks, $T_1, \ldots, T_t$, and f files, $F_1, \ldots, F_f$. Task T_i can only be executed on a single host on the platform (i.e., it is sequential task), and executes in w_i seconds (recall that hosts in our platform are homogeneous). Task T_i takes a set of input files $I_i \subset \{F_1, \ldots, F_f\}$ and produces a set of output files $O_i \subset \{F_1, \ldots, F_f\}$ ($I_i \cap O_i = \emptyset$). We assume that the input files needed by any non-entry task of the workflow are produced as output by other tasks in the workflow, and we assume no circular data dependencies. Input files required by entry-tasks are assumed available in global storage at the onset of the workflow execution. Note that in the literature workflow application are typically modeled as Directed Acyclic Graphs (DAGs), in which vertices are tasks and edges are files. However, this notion of task dependencies is vague as a task can generate more than one file that is used by another task (in which case one must consider two edges between both vertices), and a file can be used as input by multiple tasks (in which case a single file can lead to multiple edges). To avoid this confusion, in this work we simply consider a workflow as a DAG in which vertices are either tasks or files, and edges represent input/output file usage by tasks. Figure 2.2 shows an example workflow.

We assume that a task can only start once all its input files are available in stable storage, and that a task always writes its output files to stable storage. In other words, we do not consider "in-memory" data, even if a task produces a file and another task using this file happen to execute in sequence on the same host. This assumption corresponds to the reality of most scientific workflow applications in which tasks are "opaque" executables (i.e., legacy code) in which input and output is from and to files. Note that researchers are investigating "in-situ" workflow executions in which application data is held in RAM and file I/O can be avoided [39]. But in-situ executions require modifying the implementation of the tasks, which is often not feasible.

Finally, we assume that a task can only start when all its input files are available in stable storage, and that a task's output files are only available in stable storage once the tasks as completed. Again, this corresponds to the reality of real-world workflow applications in which tasks are opaque executables, and there is no feedback from these executables regarding which produced data files may have been finalized although the task is still executing.

2.3 Execution Model

We assume a workflow execution environment in which, when running a task, for each of the task's output file we can specify either to write it in local storage or to global storage. To the best of our knowledge, this capability is currently not available in current Workflow Management Systems (WMSs), and one of the objectives of this work is to make a case that it should be. We assume that I/O to/form the global storage is fully concurrent across hosts (but with bandwidth sharing).

We also assume that tasks running on different hosts can only "communicate" using files stored in global storage, i.e., as opposed to explicit network communications or explicit file copies from local storage to global storage. Recall that workflow tasks are typically opaque executables that do not perform network communication /or file copies. Therefore, allowing network communication and/or file copies would entail modifying the tasks' implementations and/or using some runtime system to orchestrate these communications/copies. While this

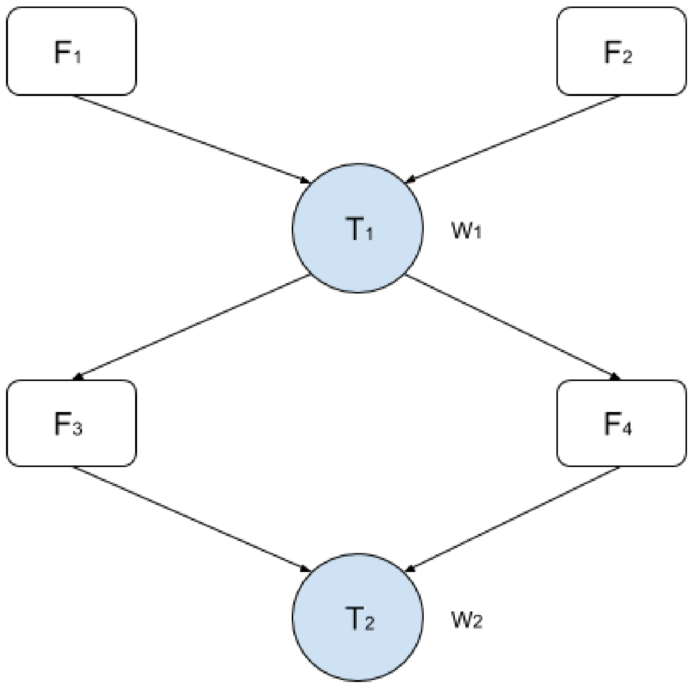

Figure 2.2: An workflow with two tasks T_1 and T_2 with $I_1 = \{F_1, F_2\}$, $I_2 = \{F_3, F_4\}$ and $O_1 = \{F_3, F_4\}$, $O_2 = \{\}$ and run in w_1, w_2 seconds respectively.

is conceivable, it is not necessarily practical on current HPC systems and with current WMS implementations. In this work, we simply assume the typical workflow application scenario in which the behavior of the task implementations cannot be modified, and we assume no particular capabilities beyond the ability to redirect files produced by tasks to global or local storage.

Finally, we also assume that a task can be replicated on multiple hosts. This implies that the WMS is capable of submitting multiple replicas of a task to different hosts. This capability is available in most WMS, e.g., for the purpose of fault-tolerance.

2.4 Objective

Given the model and constraints above, our scheduling objective is to minimize the overall workflow execution time, or *makespan* (i.e., the maximum completion time over all tasks in the workflow).

Unsurprisingly, this scheduling problem is NP-complete. A special case of it, DAG scheduling without considering data locality, is already NP-complete. Considering data locality and limited storage capacities makes the problem even more combinatorial. As a result, in this thesis we attack the problem by proposing (polynomial-time) heuristics.

CHAPTER 3
RELATED WORK

DAGs (Directed Acyclic Graphs) are a general model of computation, and therefore the issue of DAG scheduling has received an enormous amount of attention. DAG scheduling is known to be an NP-Complete problem even with restricted assumptions, and thus many polynomial-time scheduling heuristics have been proposed (see [26] for a survey of standard static DAG scheduling heuristics), typically with objective being to minimize makespan. Many variations of the DAG scheduling problem have been considered, and in particular variations in which there are communications between tasks (i.e., data dependencies in addition to control dependencies). Considering communication further complicates the scheduling problem. A typical assumption is that if a parent task generates data for a child task that is scheduled on the same host, then the communication cost is zero (note, however, that in case the child execution is not immediately after the completion of the parent, then the data would have to persist locally, in RAM or on disk). Otherwise, explicit inter-task communication has to be done on a network with some overhead. Although in a different setting, the work in this thesis is related to these communication-aware efforts because we consider tasks that communicate via files, and the global (high overhead) vs. local (low overhead) storage trade-off is akin to the "not on the same host" (high overhead) vs. "on the same host" (low overhead), albeit with a data locality component. Other works do consider file-based inter-task communication, and in this sense are more related to this work. We review relevant previous work that consider locality for DAG scheduling hereafter (Section 3.1), and then highlight how work differs and yet targets a setting directly relevant to practice (Section 3.2).

3.1 Locality Aware Scheduling

Many authors have considered minimizing workflow makespan assuming that each host has only local storage but data can be explicitly communicated (i.e., copied) between local storage at different hosts. In this case, the general idea is to promote data locality by attempting to schedule a task on a host that has in local storage as much of the input data needed by a task as possible, so as to reduce communication overhead. We describe three example of such works hereafter.

Horiuchi et al. [23] propose a straightforward list-scheduling method in which ready tasks are simply scheduled dynamically on idle hosts, always picking the host that has in its local storage the largest amount of input data needed by the task. Note that a bigger part of the contribution in [23] is the discovery of the workflow DAG based on profiling runs, while in this work we assume the workflow DAG is known (e.g., created explicitly by the end-user).

Bozdag et al. in their work [8] propose a greedy heuristic that attempts to schedule parent-child task pairs on the same host, prioritizing task pairs based on the amount of data that would be communicated were the two tasks executed on different hosts.

Vydyanathan et al. [34] target a version of the problem in which workflow tasks are parallel. They propose a heuristic called Locality Conscious Processor Allocation and Scheduling (LCPAS). LCPAS computes the critical path of the workflow and iteratively reduces both the communication and computation cost along the critical path. At each iteration, it reduces either the computation cost of a task or the communication cost between a pair of tasks. Computation costs are reduced by allocating more hosts to a task. Communication costs are reduced in two ways. First, more hosts are allocated to tasks so that they can benefit from parallel data transfer mechanisms, thus increasing communication bandwidth. Secondly, data locality is considered by striving to schedule tasks in free time slots on hosts that hold the largest amount of input data in their local storage. Note that in this work we do not consider workflow task that execute on multiple hosts.

Another idea for improving data locality is to partition a task graph into subgraphs. The subgraphs can be scheduled either on a single host or on a group of hosts that have fast inter-host communication bandwidth. The subgraphs are formed so that intra-subgraph edges have heavy weights (i.e., large data amounts) and inter-subgraph edges have light weights (i.e., low data amounts). Tanaka et al. [29] propose a multi-constraint graph partitioning method to obtain a balanced partitioning. Rather than simply partitioning the graph into same-size subgraphs based on edge weights (i.e., being purely data-conscious), their method produces subgraphs that have the same parallelism, as much as possible. This then makes it straightforward to schedule the subgraphs efficiently on groups of hosts. Ahmad et al. [1] also propose an algorithm to partition the graph such that the inter-subgraph data movement is minimal, but they assign tasks so different subgraphs based on edge weights. Once subgraphs have been computed, tasks are mapped to hosts so as to minimize inter-subgraph data movement. Note that they assume that the communication between tasks is done via direct data transfer and not via files stored in stable storage.

3.2 Our Approach

The previous works discussed in the previous section all attempt to maximize notions of data locality. In some cases locality means presence of files in stable storage (if tasks communicate via files), and in others it means network proximity (if tasks perform direct network communications). In this work, we only consider inter-task communication via files, and thus our notion of locality is whether a file is present in local storage at a host. This corresponds to current practice for scientific workflows, our target application domain.

One important factor is that relevant previous works in the workflow/DAG scheduling literature do not consider capacity limitations for local storage. Yet, in practice, local storage capacity is limited. For instance, in [32] the authors discuss the impact of limited local storage capacity, and thus motivate the use of "object stores" close to the hosts to help accommodate the necessary input files and intermediate data. Consequently, in this work we do consider limited local storage capacity. It is, thus, important to make careful and efficient storage decisions as there is both a notion of locality and of limited storage.

In this work, again motivated by the current practice of executing workflow applications on HPC platforms (see Chapter 2), we do not consider explicit data copies between local storage and global storage. This is a drastic difference with previous work and the implication is that if a file has been stored in local storage at a particular host, then it is only visible at that host. As a result, application parallelism can be reduced (i.e., all tasks needing that file can only run on that particular host). To address this issue we employ task replication. Task replication has been used in the context of DAG scheduling for reducing task start times (e.g., the 7 "Task-Duplication-Based" scheduling algorithms surveyed in [25] or to increasing reliability [35], but in this work we use it to mitigate parallelism loss due to local storage usage.

To the best of our knowledge, our target scheduling problem, which is relevant to current practice (see Chapter 2 for full details and further justifications), has not been previously studied.

CHAPTER 4
HEURISTICS

As explained in Chapter 2 our target problem is trivially NP-Complete. We thus approach the problem with heuristics. More specifically we design heuristics that use both local storage (because it is fast) and global storage (because it is of infinite capacity and the fact that it is accessible by all hosts) with the goal of minimizing workflow execution time.

We face a trade-off when using both types of storage. The use of higher bandwidth local storage reduces I/O time but, if we look on the other side of the coin, it may result in reduction of parallelism. This reduction in parallelism, assuming that a file is stored only in the local storage of a host, can occur for two different reasons. First, a task using that file will have to wait for that host to become idle before it can begin execution. Second, and related to the first reason, all the tasks using this file can only be executed on that particular host, and thus their executions will be serialized.

By contrast, storing a file in global storage is good because no constraint is imposed on where a task using this file can be executed. But, the use of global storage is also not good because of its lower bandwidth. Recall that state-of-the-art WMSs use solely global storage, and thus suffer from high I/O overhead for data-intensive workflows.

In this chapter we describe our approach for designing heuristics that attempt to address this trade-off. First we describe the overall skeleton of our heuristics (Section 4.1), which performs task replication, host selection, and storage selection. Heuristics for making these decisions are detailed in Sections 4.3-4.4, respectively.

4.1 Overall Approach

As described in chapter 2, our objective is to schedule all the tasks in a workflow onto hosts in a homogeneous cluster. This scheduling problem is *off-line*, in that we have full knowledge about each task (i.e., runtime, input/output files and their sizes). This knowledge is used by our scheduling heuristics, all of which perform "list scheduling" (i.e., whenever there is at least one idle host and there is at a least one ready task, then at least one instance of this ready task is started on one idle host). However, our heuristics do not compute a static schedule ahead of execution but instead maintain a list of ready tasks and schedule the tasks as they become ready and as hosts become idle. Simply put, we do *dynamic scheduling* rather than static scheduling.

As we dynamically schedule tasks, we perform task replication and make decisions about storage location of the output files produced by each task. We do task replication in order to alleviate the loss of parallelism due to the use of local storage. Replication of a task on multiple hosts allows its output files to be stored on local storage at each of these hosts. As a result the task's children can execute concurrently on these hosts.

Overall our approach proceeds in the following steps: (i) sort the list of ready tasks; (ii) pick the first task in the list; (iii) compute the maximum desired number of instances of this task to be scheduled; (iv) schedule some or all the instances on idle hosts that have the maximum number of bytes of the task's needed input data in their local storage; (v) decide which output files of the task (and each of its scheduled replicas) should be written to local storage and which should be written to global storage. These steps are implemented in the ExecuteWorkflow procedure shown in Algorithm 1, which takes as input a workflow and a set of hosts.

Algorithm 1 ExecuteWorkflow procedure

```
1: procedure ExecuteWorkflow($workflow$, $hosts$)
2:     $file\_locations \leftarrow \{\{global\}, \ldots, \{global\}\}$
3:     while $workflow$ has uncompleted tasks do
4:         $ready\_tasks \leftarrow$ ReorderReadyTasks($workflow.getReadyTasks()$)
5:         for each $task$ in $ready\_tasks$ do
6:             $num\_task\_instances \leftarrow$ PickNumTaskInstances($task, ready\_tasks, hosts$)
7:             if $num\_task\_instances = 0$ then
8:                 $continue$
9:             end if
10:            $candidate\_hosts \leftarrow$ pickHosts($task, num\_task\_instances, hosts$)
11:            for each $host$ in $candidate\_hosts$ do
12:                $file\_locations \leftarrow$ MakeStorageDecisions($file\_locations, host, task$)
13:                LaunchTaskInstance($task, host, file\_locations$)
14:            end for
15:        end for
16:        Wait for any $task$ completion
17:        Mark $task$ as completed, mark its children as ready
18:    end while
19: end procedure
```

First, Algorithm 1 initializes a map data structure, $file\_locations$, that specifies for each file in the workflow where it should be written and thus will be available for subsequent task executions (line 2). Values in this map are sets, since a file can be at multiple locations. Elements of these sets are either *global* or *h.local*, which denotes the local storage on host h, for all hosts h. Initially, all files are set to be written to global storage. Then the algorithm iterates until all workflow tasks are completed (line 3). At each iteration, ReorderReadyTasks is called (line 4). This procedure (pseudo-code not shown) simply reorders the tasks in the list of ready tasks by non-decreasing number of their children. The rationale is that prioritizing those tasks that have more children will increase the number of ready tasks, and thus potentially reduce workflow execution time. This is actually a well-known list-scheduling principle (e.g., see [28]) when computing a schedule dynamically at runtime. For each ready task, in this order, the algorithm then decides whether and how to launch it. First, in line 6, it calls procedure PickNumTaskInstances, which implements our task replication strategy and returns a number of task instances to be launched (see Section 4.2). If this procedure returns 0, then no instances of the task is launched (lines 7-9). Otherwise, procedure pickHosts is called (line 10). This procedure implements our host selection strategy (see Section 4.3) and returns a list of candidate hosts on which task instances should be launched. For each such candidate host, procedure MakeStorageDecisions is called (line 13). This procedure implements our storage selection strategy (see Section 4.4) and thus updates the $file\_locations$ map. Finally, procedure LaunchTaskInstance (pseudo-code not shown) is called to start an instance of a task on a given host, writing output files as specified in the $file\_locations$ map.

4.2 Task Replication Strategy

Given an ordered list of tasks, for each task in this order we must decide on the number of instances of this task to execute. This task replication strategy is implemented as procedure PickNumTaskInstances (see Algorithm 2, called in line 6 of Algorithms 1). Procedure PickNumTaskInstances determines the number of idle hosts (line 2), and computes the number of "extra" idle hosts, i.e., the number of hosts that would remain idle after one instance of each ready task is launched on one idle host (line 3). This number is then divided by the number of extra hosts by the number of ready tasks (line 4). The rationale is that each ready task should have at least one instance started if possible, and that "extra hosts" are given out as fairly as possible to tasks for the purpose of replication. Note, however, that a task can never have more instances than its number of children as more instances would not be useful (line 5).

Algorithm 2 Algorithm to decide the number of instances of a task

```
1: procedure PickNumTaskInstances($task, ready\_tasks, hosts$)
2:     num_idle_hosts ← number of hosts currently idle
3:     extra_hosts ← (num_idle_hosts − $ready\_tasks.size() - 1$)
4:     num_instances ← $max(1, \lceil$extra_hosts/$ready\_tasks.size()\rceil)$
5:     num_children ← $max(1, task.getNumChildren())$
6:     return $min$(num_children, num_instances)
7: end procedure
```

4.3 Hosts Selection Strategy

Given a decided number of instances of a task, procedure pickHosts (called in line 9 of Algorithms 1) is used to pick a set of hosts on which to execute these instances. pickHosts is shown in Algorithm 3. The rationale in pickHosts is to pick those hosts that have the maximum number of bytes of input data for the task in their local storage. The goal is to take advantage off the high I/O bandwidth of local storage. Procedure pickHosts creates a list of idle hosts and for each such host computes how many bytes of input data is stored in local storage (lines 2-8). This list is sorted by non-decreasing order of number of bytes stored (line 9). It then computes how many hosts should be returned, *num*, as the minimum of the number of hosts in the list and the number of task instances desired. Note that it may thus return fewer hosts than the decided number of instances, in which case fewer instances will be started (line 10). The first *num* hosts in the list are returned (line 11).

Algorithm 3 Algorithm to pick a set of hosts on which to execute instances of a task

```
1: procedure pickHosts($task, num\_task\_instances, hosts$)
2:     picked_hosts ← an empty list
3:     for each host in hosts do
4:         if host is idle then
5:             picked_hosts ← picked_hosts ∪ {host}
6:             numBytes[host] ←#bytes of input data for task in local storage at host
7:         end if
8:     end for
9:     sort picked_hosts by non-decreasing numBytes value
10:    num ← $min$(picked_hosts.$size()$, $num\_task\_instances$)
11:    return first num hosts in picked_hosts
12: end procedure
```

4.4 Storage Selection Strategy

Given a decided number of task instances and the set of hosts on which to launch these instances, for each task instance we need to decide where it should write its output files: global storage or local storage at the host on which it is launched. For this purpose, we formulated three related heuristics, as described hereafter, to implement procedure MAKESTORAGEDECISIONS (called in Algorithm 1 at line 12).

4.4.1 S_W_RATIO

The main intuition behind this heuristic is that a file should be stored in local storage if it is large (as local storage has higher bandwidth than global storage) and/or if it is used by tasks with relatively short execution times (as these tasks could then become I/O-bounded if the file were to be stored in global storage). Algorithm 4 shows the pseudo-code for this heuristic. It first initializes a list called *file_list* with all the output files of the task to be scheduled (line 2), and then sorts this list so as to determine which of these files should have priority for being written to local storage (lines 2-3).

Formally, consider a task T_i, an instance of which is to be started on a host *host*. Let $\{F_1, \ldots, F_{n_i}\}$ be the files produced by T_i. For each file F_j, let us define the set of indices of tasks that uses F_j as input:

$$\mathcal{I}(F_j) = \{k | T_k \quad \text{uses } F_j \quad \text{as input}\} \ .$$

For each file F_j we then compute its "S/W ratio" SW_j as:

$$SW_j = \max_{k \in \mathcal{I}(F_j)} \frac{s_j}{w_k}$$

where s_j is the size in bytes of file F_j and w_k is the execution time in seconds of task T_k. Intuitively, this ratio corresponds to the worst I/O-boundness among the tasks that use file F_j as input. We then sort all output files of T_i, $\{F_1, \ldots, F_{n_i}\}$, by non-decreasing SW_j values.

For each file thus sorted (line 4), if the task has children (line 5), and if the file can fit on the local storage at *host* (line 6), then we attempt to place the file on that local storage (line 7). Note that if a task has no children then it is an exit task of the workflow and its output files should always be written to global storage.

Placing the file in local storage may actually increase the overall execution time. This is because of the loss of parallelism caused by the use of local storage. For instance, consider a

Algorithm 4 MAKESTORAGEDECISIONS, using the S_W_RATIO heuristic

```
1:  procedure MAKESTORAGEDECISIONS($file\_locations, task, host$)
2:      $file\_list \leftarrow task.getListOfOutputFiles()$
3:      sort $file\_list$ by S/W ratio                                  ▷ See Section 4.4.1
4:      for each $file$ in $file\_list$ do
5:          if $task.getNumChildren() > 0$ then
6:              if $file$ can fit in $host.local$ storage then
7:                  $file\_locations[file].add(host.local)$         ▷ Tentatively use local storage
8:                  for each $task'$ that has $file$ as input do   ▷ Check if local storage is useful
9:                      if $task'$ would complete earlier using global storage then
10:                         $file\_locations[file] = \{global\}$
11:                     end if
12:                     for each $file' \neq file$ in $file\_list$ do    ▷ Check if it is safe to use local
13:                         if $file\_locations[file'] \cap \{global, host.local\} = \emptyset$ then
14:                             $file\_locations[file] = \{global\}$
15:                         end if
16:                     end for
17:                 end for
18:             end if
19:         end if
20:     end for
21:     return $file\_locations$
22: end procedure
```

task with n children, each with an execution time of t seconds, and all using one output file from their parent stored in local storage. These n children are then necessarily serialized. Therefore, some of these children may complete later than if the file had been stored in global storage instead (because children could execute in parallel). Consequently, we check if any child would complete later using local storage, due to serialization, than using global storage, in spite of lower I/O bandwidth (line 9). If so, we decide to write the file to global storage (line 10). Note that this is merely an estimation that ignores resource contention due to the execution of other tasks in the workflow.

Writing the file to local storage, even if it reduces the overall execution time, may lead to a situation where a task would have to read input files from local storage at two or more different hosts. Given our model and assumptions in Chapter 2 the execution of this task is then not feasible. We find such situations by checking if each $file'$ of $file\_list$ (different from the file under consideration) is present in global or in $host.local$ storage (line 13). If not, i.e., $file'$ is stored on local storage at another host, we then decide to store the file to global storage (line 14). The procedure then terminates and returns an updated file location map.

4.4.2 INV_S_W_RATIO

This heuristic is just the inverse of the S_W_RATIO heuristic. Instead of computing the S/W ratio, we compute the W/S ratio for all the output files of each task. The file having the highest W/S ratio is prioritized and thus stored in local storage if possible. The basic intuition behind this heuristic is to quickly execute a task that has a higher runtime and uses relatively small files. This could be a sensible choice especially when this task lies on the DAGs' critical path.

4.4.3 THREE_PASS

The basic goal of this heuristic is to load balance the execution of the ready tasks in addition to using local storage sensibly. The motivation for this heuristic stems for our observation that the two previous heuristics sometimes unnecessarily use local storage for tasks that are not makespan "bottlenecks." More specifically, if a ready task would finish much earlier than other ready tasks, it may be a good idea to let that task read input files from global storage, thus slowing it down.

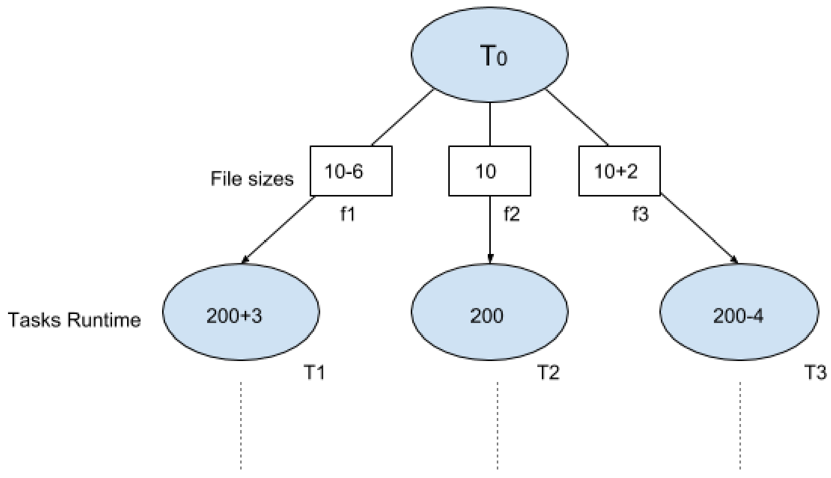

Figure 4.1: A simple example for which S_W_RATIO and INV_S_W_RATIO make the "wrong" choice.

In Figure 4.1, we show a case where S_W_RATIO and INV_S_W_RATIO do not necessarily choose the best task that should use local storage. In the figure, a task T_0 has three children tasks, T_1, T_2, T_3 that have runtimes of 203, 200, 196 seconds and task as input files f_1, f_2, f_3, respectively. The size of file f_1, f_2, f_3 is 4, 10, and 12 GiBs, respectively. Also assume that local storage can hold only one file (e.g., its capacity is 12 GiBs). Therefore, these files have S/W ratio of $(4 \cdot 1024 \cdot 1024)/203, (10 \cdot 1024 \cdot 1024)/200$, and $(12 \cdot 1024 \cdot 1024)/196$. So, S_W_RATIO will prioritize task T_3 because its input file has the highest S/W ratio of $(12 \cdot 1024 \cdot 1024)/196$. Conversely, INV_S_W_RATIO will prioritize task T_1 because its input file has the highest W/S ratio of $203/(4 \cdot 1024 \cdot 1024)$. Assuming I/O bandwidth to global storage is 1 GiB per second, then the total estimated completion time of T_1, T_2, T_3 will be 207, 210 and 208 seconds, respectively. So, the task that finishes the latest, when not using local storage, would be T_2. And, neither S_W_RATIO nor INV_S_W_RATIO will place T_2's input file in local storage.

In this section we propose the THREE_PASS heuristic, which would make the right choice for the case in Figure 4.1, by recognizing that the task that would complete the latest if using global storage should most likely be made to use local storage. More generally, THREE_PASS attempts to make storage decisions that will slow down fast tasks and thus potentially speed up slow tasks, thus achieving better load balancing. This is done by pre-processing the DAG in three different passes before execution:

1. Compute task top-levels;
2. Based on top-levels, make putative storage decisions for good load-balancing;
3. Fix storage decisions as necessary to make execution feasible.

We detail each pass hereafter.

First Pass – In the first pass, we traverse the DAG from top to bottom and for each task we compute its estimated completion time (ECT). This is done by computing the task's top-level, i.e., the length (in seconds) of the longest path from the DAG's entry task to this task, excluding this task's execution time. In other words, the top-level is the sum of all the computational runtime of the tasks from the entry task to this task plus the sum of the I/O times between those tasks. Since files may be stored in global or local storage, we do not know exact I/O times beforehand. However, we know that I/O time between a parent task t_1 and a child task t_2 is directly proportional to the number of bytes (i.e., files sizes) produced by t_1 and consumed by t_2. Thus, while calculating the ECT of a task t, we assume that the I/O time is directly proportional to the sum of all the input file sizes of task t. Also, while computing the top-level of each task, we assume an infinite number of hosts, and that all hosts are connected to a hypothetical global storage with a bandwidth equal to that of our hosts' local storage. Note that similar assumptions are commonly used in DAG scheduling heuristics [26].

Second Pass – In the second pass, we traverse the DAG again from top to bottom, and process tasks level by level. A level is defined as the set of tasks that are at the same distance (in maximum number of edges) from the DAG's entry task. For each level we use a simple heuristic. We assume that all files are available in local storage, and based on this assumption we compute the execution time (including I/O) of each task. We then select the task with the slowest execution time. For each task that has a faster execution time, then we "slow down" its execution by forcing it to use global storage instead of local storage. This achieves some approximate load-balancing of the execution. More sophisticated decisions are possible (to truly load-balance the execution via some iterative process, or by selecting only subsets of input files to be stored in global storage). But it is important to note that while making such decisions we do not know what the actual schedule will be and/or what the available local storage capacity will be on the hosts that end up executing particular tasks. Therefore, it is unclear whether more sophisticated load-balancing heuristics would be effective. Our results in Chapter 5 showcase instances in which the approximate load-balancing strategy above is effective.

Third Pass – In the third pass, we repair infeasible executions. This is because, for the same reasons as for the S_W_RATIO heuristic, the storages decisions may create a situation where a task would have to read input files from local storage on two distinct hosts, which is prohibited by our execution model (see justification in Chapter 2). In such cases, as in S_W_RATIO, we force some of the parent tasks to write their output files to the global storage. This is done by traversing the DAG from top to bottom one last time.

These three passes are executed instead of the initialization at line 2 in the EXECUTE-WORKFLOW procedure (Algorithm 1). The execution of this procedure is unchanged, but for the MAKESTORAGEDECISIONS procedure. This procedure, simply implements the storage decisions made by the three passes, but may fail to use local storage due to capacity constraints encountered at runtime. In this case, global storage is used instead.

CHAPTER 5
RESULTS

In this chapter we evaluate and compare the effectiveness of the heuristics described in Chapter 4. Our broader goal is to quantify the workflow execution time reduction due to the judicious use of local storage. We obtain results both in simulation and on a real-world HPC cluster. We use simulation because it allows us to run large numbers of hypothetical application and platform scenarios, which would be time-intensive, labor-intensive, and/or infeasible for the real-world applications on real-world platforms at our disposal. Nevertheless, we also obtain real-world results for selected experimental scenarios, in part to verify that our simulation results are representative of real-world settings.

This chapter is organized as follows. Section 5.1 details our simulation experimental methodology and Section 5.2 discusses simulation results. Section 5.3 details our real-world experimental methodology and Section 5.4 discusses real-world results. Finally Section 5.5 provides a synthesis of our findings.

5.1 Simulation Methodology

5.1.1 Simulation Software

We use the WRENCH [38] simulation tool to implement in C++ a simulator of workflow executions using our heuristics. WRENCH is a scientific instrument designed as a software framework for simulating workflow executions on arbitrary (simulated) platforms, and a large part of its intended use is the investigation of scheduling strategies. WRENCH is built on top of SimGrid [13], which provides core simulation capabilities and is used to study distributed systems and applications that occur in the Grid, Cloud, HPC or P2P computing domains. The simulation models implemented in SimGrid have been developed for over a decade and have been thoroughly validated. Furthermore, SimGrid makes it possible to execute simulations on a single computer (i.e., it has low execution time and low memory footprint). In the end, the WRENCH framework provides us simple abstractions that allow us to focus mostly on implementing our heuristics and, because WRENCH builds on SimGrid, makes it possible to evaluate our heuristics accurately and scalably for arbitrary application and platform configurations of interest.

5.1.2 Simulated Workflows

Workflow Structures

We wish to simulate the execution of workflow configurations that are representative of actual scientific workflow applications. To this end, we consider seven types of of workflow structures so as to test our heuristics on a wide variety of application scenarios. Three of these structures are based on real-world applications: Genome (from the bioinformatics domain) [17], Cybershake (from the earthquake engineering domain) [22], and Montage (from the astronomy domain) [27]. Please refer to Appendix A for example visual representations of these workflows. We also consider four synthetic workflow structures: Outtree, Intree, ForkJoinSeq1, and ForkJoinSeq2. Outtree and Intree are simple out-tree and in-tree graphs. ForkJoinSeq1 corresponds to a sequence of fork-join graphs in which the sink of a fork-join graph is the source of the next fork-join graph in the sequence. ForkJoinSeq2 also corresponds to a sequence of fork-join graphs, but in this case the sink of a fork-join graph is the parent of the source of the next fork-join graph in the sequence. These synthetic workflow structures often occur as part of larger workflow structures encountered in real-world scientific workflow applications.

We keep the numbers of tasks fixed (but we do vary the scale of the platform on which the workflow executions are simulated). Specifically, we generate workflows for each above structures with 1,000 tasks. To generate Genome, Cybershake, and Montage workflows we rely on the workflow generated provided by the Pegasus project [15]. This generator outputs workflow structures that are based on actual workflow instances for these applications. It turns out that this generator cannot generate structures for arbitrary numbers of tasks for all applications. In particular, we are only able to generate Genome workflows with 997. For our synthetic workflows, because of the 1,000-task constraint we vary arities accordingly. For instance, we generate Intree, resp. Outtree, workflows with branching factor 4, but the first, resp. last, level of the workflow contains tasks with fewer than 4 children, resp. parents.

For each workflow structure, we generate 10 workflow instances with task execution times sampled from a uniform random distribution with range [0, 3600] (in seconds), and file sizes are sampled from a uniform random distribution with range [10KiB, 2GiB]. These ranges are somewhat arbitrary (even though they correspond to some real-world applications), but hereafter we explain how we vary the data-intensiveness of our generated workflow instances.

Data Intensiveness

A key characteristic of scientific workflows, in particular for the purpose of studying scheduling strategies, is *data-intensiveness*. The data-intensiveness of a workflow is defined as the ratio of the total time spent computing to the total time spent doing I/O, or CCR (Computation to Communication Ratio). The time spent doing I/O, in our case, depends on whether the I/O is to/from local or global storage, which will vary between executions depending on the actual schedule, which is computed at runtime. Therefore, we simply compute the time spent doing I/O as the time to read all files in the workflow from global storage (see Section 2.1 for how we pick a realistic global storage bandwidth value).

For each workflow instance generated as described above, we scale the files sizes by a single factor so as to generate instances with given CCR values. We wish to experiment with low CCR values because many real-world workflow application are data-intensive, which provides motivation for this work in the first place. We consider CCR values between 1 (roughly equal time-consuming to perform I/O and to compute) and 20 (roughly 20x less time-consuming to perform I/O than to compute).

5.1.3 Simulated Platform Parameters

Number of Hosts

While we keep the number of tasks in our workflows at 1,000, we vary the number of compute hosts, h. The number of hosts is an important parameter because as it increases there is more opportunity for task replication. We, thus, vary the number of hosts from 2 to 200.

Storage system

Recall from Chapter 2 that, based on measurements on real-world clusters, we set the local storage bandwidth to 2 GBps and the global storage bandwidth to 100 MBps. Recall also that an important parameter, *num_conns*, is the degree of sharing of the global storage bandwidth among the compute hosts. Conceptually *num_conns* is the number of individual links that connect the compute hosts to the global storage. In the real-world clusters at our disposal, we have found that $num\_conns = 1$. However, in some other storage systems there are dedicated "I/O nodes" that provide larger aggregated bandwidth to the storage, because they allow concurrent accesses. Therefore, for completeness, in our experiments we consider *num_conns* ranging from 1 to 50. A larger *num_conns* value would in general lessen the performance penalty of using global storage relative to using local storage.

5.2 Simulation Results

We first obtain a set of "base" results using particular values of the CCR, num_conns, and the number of hosts h. Namely, we use $CCR = 1$ (perfect balance between I/O and computation times, which implies a data-intensive workflow), $num\_conns = 1$ (a single link to the global storage, as seen in our real-world HPC clusters), and $h = 10$ (a small typical allocation obtained on an HPC cluster, but sufficient for task replication to have some benefits).

To evaluate our heuristics we also consider two baseline heuristics:

ALL_IN_GLOBAL: Since, state-of-the-art WMSs only use global storage, we consider a simple ALL_IN_GLOBAL heuristic, which simply never uses local storage. A broad practical objective of this work is to quantify potential workflow makespan reductions due to using local storage.

RANDOM: We also consider a heuristic that randomly decides whether a file should be written to local or to global storage. However, this heuristic also checks whether storing the file in local storage is "worth it" in terms of loss in parallelism as done in the S_W_RATIO and INV_S_W_RATIO (see lines 9-11 of Algorithm 4). A reason for including this heuristic is that it should allow to evaluate whether our proposed heuristics make good decisions when using local storage. Furthermore, for complex combinatorial scheduling problems such as ours, purely random heuristics are known to produce, sometimes unexpectedly, good results. Note that, like our proposed heuristics, RANDOM may make decisions that make task executions infeasible. So, RANDOM prevents infeasible task executions while making storage decisions.

Through this chapter we use ALL_IN_GLOBAL as our reference and discuss improvement achieved by other heuristics, including RANDOM, relative to ALL_IN_GLOBAL.

5.2.1 Base Results

For each workflow structure, we simulate the execution of each of 10 instances with each of our 5 heuristics, with our base parameter values: $num\_conns = 1$, $CCR = 1$, $h = 10$. Each simulation outputs a makespan, for Table 5.1 shows makespan differences relative to ALL_IN_GLOBAL, averaged over the 10 instances. The average, however, is not biased by any outliers in the data which is supported by the fact that maximum coefficient of variation ranges to 7 in our individual executions of each workflow.

DAX	ALL_IN_ GLOBAL	S_W_RATIO	INV_S_W_ RATIO	THREE_PASS	RANDOM
Cybershake	0%	17.149%	17.174%	14.763%	17.191%
Montage	0%	2.080%	2.080%	-0.439%	2.079%
Genome	0%	-51.463%	-29.736%	-48.209%	-41.598%
Intree	0%	-48.243%	-46.487%	-43.986%	-47.379%
Outtree	0%	-9.351%	-19.109%	-14.903%	-13.010%
ForkJoin Seq1	0%	-9.322%	-10.169%	-9.322%	-10.169%
ForkJoin Seq2	0%	-0.704%	-0.169%	-4.890%	-0.340%

Table 5.1: Average simulated application makespan differences relative to ALL_IN_GLOBAL for all heuristics (Base Results: $num\_conns = 1$, $CCR = 1$, $h = 10$).

The main observation from the results in Table 5.1 is that many values are negative, meaning that ALL_IN_GLOBAL is typically outperformed by heuristics that use local storage (i.e., these heuristics lead to lower makespans). This is not surprising overall, since in general using local storage should improve performance (which motivates this work), but hereafter we discuss these results for each workflow structure.

Cybershake – This workflow structure is the only one for which no heuristic outperforms ALL_IN_GLOBAL. The reason why any of our heuristic could be outperformed by ALL_IN_GLOBAL is loss of parallelism due to the use of local storage. Our solution to remedy this loss of parallelism is to use task replication. However, depending on the structure of the workflow itself, task replication may be only rarely possible, especially when the number of hosts is small. In the case of Cybershake, after a few entry tasks have completed, there is then an extremely large number (about 500) of ready tasks. Given that only 10 hosts are available, there is almost never any opportunity for task replication given that our task replication strategy of using only "extra hosts" for task replication. It follows that, due to no task replication, subsequent levels in the workflow suffer from loss of parallelism. This explanation is confirmed by experiments with Cybershake workflows with only 200 tasks executed on 10 hosts and workflows with 1,000 tasks but executed on 100 hosts. In both cases, all our heuristics outperform ALL_IN_GLOBAL. In Section 5.2.4 we show results for various numbers of hosts in the platform. We conclude that for very shallow workflows like Cybershake, and unless the number of hosts is sufficiently large, our task replication approach is not sufficiently aggressive to warrant not using the standard ALL_IN_GLOBAL strategy.

Montage – For this workflow structure only the THREE_PASS heuristic outperforms ALL_IN_GLOBAL, and it does not outperform it by a large margin (under .5% improvement). The reason why THREE_PASS is the best among the heuristics is due to the presence in Montage workflows of a commonly found sub-structure: a parent task with a single child that has many independent children. See the discussion of the results for the ForkJoin-Seq2 workflow structure hereafter. The reason why the improvement of THREE_PASS over ALL_IN_GLOBAL is only marginal is the same as for the Cybershake workflow structure, as discussed above. Therefore, similarly, reducing the number of tasks and/or increasing the number of hosts increase the improvement of THREE_PASS relative to ALL_IN_GLOBAL.

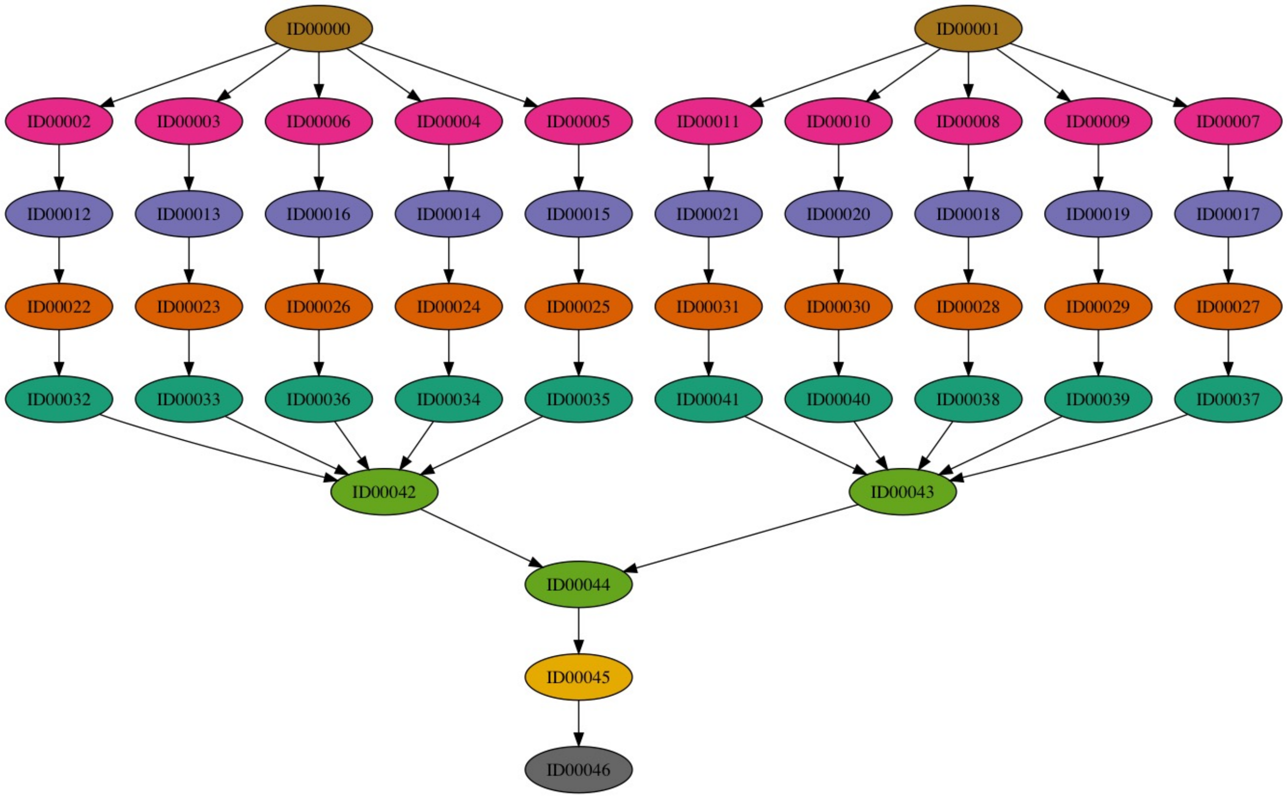

Figure 5.1: An example instance of the Genome workflow.

Genome – For this workflow structure we see that all heuristics do significantly better than ALL_IN_GLOBAL, with S_W_RATIO leading to the best results, and THREE_PASS being a close second. The reason for these large improvements is the particular structure of the Genome workflow. As seen in Figure 5.1, Genome workflows contain many single-parent-single-child structures. This allows in significant use of local storage while not negatively impacting parallelism.

Intree – Here also we see significant improvement in average makespan achieved by heuristics relative to ALL_IN_GLOBAL. Specifically, S_W_RATIO is the best heuristic for this workflow type, although it leads to results very similar to RANDOM and S_W_RATIO. For these heuristics, due to the in-three structure, storage decisions do not hurt parallelism (because for task executions to be feasible, many parents are forced to write the global storage). Yet, a significant number of files are written to local storage, leading to potentially large makespan reductions when compared to ALL_IN_GLOBAL. Because the workflow has a large number of entry tasks, THREE_PASS attempts to load-balance the first level of the workflow, thus causing many entry tasks to write to global storage ("slowing them down"). Local storage is thus not fully used for these entry tasks, which explains why THREE_PASS does not do as well.

Outtree – Here again all heuristics are better than ALL_IN_GLOBAL, with INV_S_W_RATIO leading to the best results. Overall improvement magnitudes are lower than for Intree workflows. This is due to loss of parallelism due to using local storage (which, for Intree never occurs, as explained above).

ForkJoinSeq1 – For this workflow structures all heuristics perform similarly, leading to about 10% improvement over ALL_IN_GLOBAL. This seems to indicate that for this structure using local storage helps reduce I/O overhead, but selecting which files should be written to local storage is not important.

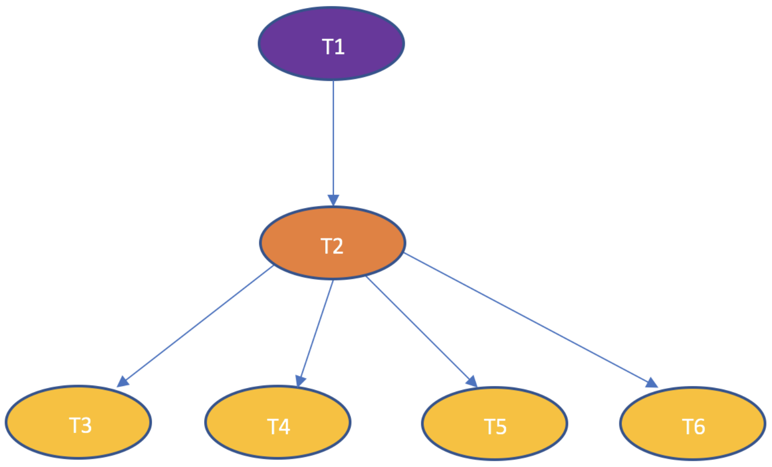

Figure 5.2: A scenario where S/W and W/S heuristics do not perform well.

ForkJoinSeq2 – For this workflow structure THREE_PASS leads to the best results, out-performing ALL_IN_GLOBAL by almost 5%. Rest of the heuristics are not significant enough when compared to ALL_IN_GLOBAL. The reason behind this is the structure of the ForkJoin Seq2. In this workflow type, we have frequent repetitions of the pattern as shown in Figure 5.2. This pattern in Figure 5.2 corresponds to a structure in ForkJoinSeq2 workflow type where every fork-join sequence is followed by a single task. In Figure 5.2, the first task T_1, having only one child, is decided to perform its I/O from local storage by S/W and W/S heuristics. Consequently, the task T_2, having many children, cannot be replicated as its input files are present in only one host where its parent task T_1 was executed. This decision greatly reduces parallelism as all the children tasks of task T_2 will have to execute on the same host where T_2 was executed. In this particular situation, to prevent this parallelism loss, instead of one step look-ahead to count the number of children, we can do a two step look-ahead to count the number of grand children of task T_1 and decide to have multiple instances of task T_1. However, this is just a particular case. We can have situations where to make correct decisions we may have to look until the very end of the graph and this hugely complicates our heuristics. And so both S_W_RATIO and INV_S_W_RATIO simply do a single step look-ahead that assigns task T_1 to use local storage and thus suffers from loss in parallelism. However, THREE_PASS, in an attempt to load-balance the tasks at every level assigns the first task T_1 to perform its I/O from global storage. As as result, the THREE_PASS heuristic outperforms all other heuristics for this workflow type. Generally, such patterns are frequent in many workflow types and the THREE_PASS heuristic can help improve the makespan for those workflows.

All of these above results provide a measure of the performance of all our heuristics for our base scenario, i.e., $num\_conns = 1$, $CCR = 1$, $h = 10$. We now turn to exploring how these parameters impact the performance of our heuristics. The following three sections explore the impact of num_conns, CCR and h), respectively. We select representative results for three particular workflows: Genome, Cybershake, and Outtree. See Figures 5.1, 5.3, and 5.4 for example instances of these workflows.

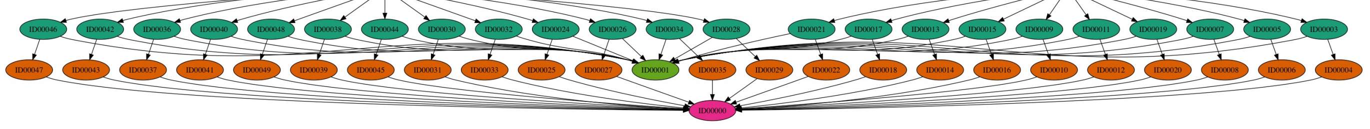

Figure 5.3: An example instance of the Cybershake workflow.

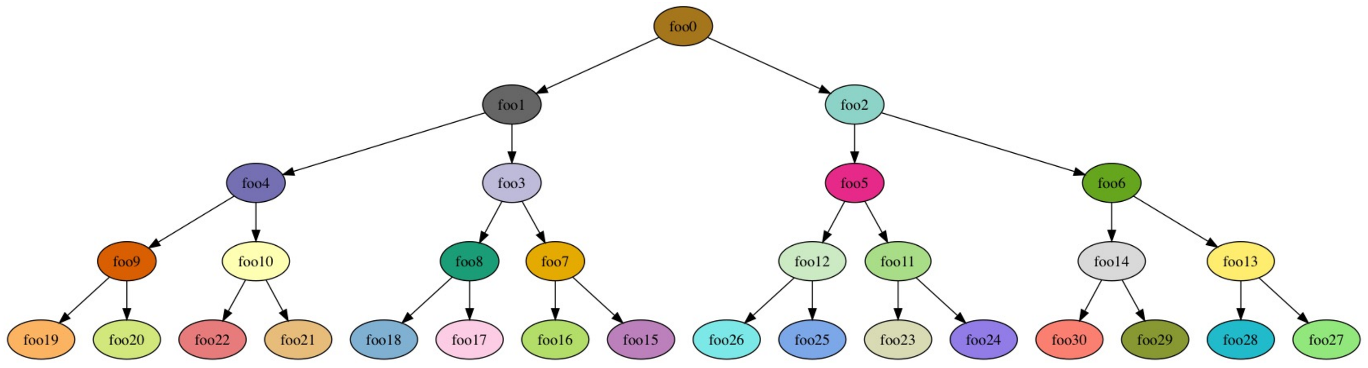

Figure 5.4: An example instance of the Outtree workflow.

5.2.2 Impact of *num_conns*

In this section, we discuss the impact of varying *num_conns*. All the plots shown in this section show average simulated application makespans (in seconds) on the vertical axis, and *num_conns* on the horizontal axis, which ranges from 1 to 50. In all these results, CCR is equal to 1, h is equal to 10, and the number of tasks is 997 for Genome and 1000 for the other workflows as described in Section 5.1.

Genome Results

Figure 5.5 shows results for the Genome. We can see from the figure that at lower values of *num_conns*, i.e., when there is more bandwidth contention to global storage, ALL_IN_GLOBAL is significantly worse than all the other heuristics. However, as *num_conns* increases, ALL_IN_GLOBAL eventually outperforms all the other heuristics. This is an expected result as increasing *num_conns* allows hosts to have a bigger share of global storage bandwidth. As a result, ALL_IN_GLOBAL does better when *num_conns* is high, i.e., when I/O to/from global storage is not as expensive and there is no loss of parallelism due to simply not using local storage. All other heuristics see their makespans decrease as *num_conns* increases. The reason behind this is that other heuristics do not only use local storage at the hosts but also make use of global storage whenever necessary.

Cybershake Results

Figure 5.6 shows results for the Cybershake workflow. Recall from Section 5.2.1 that in Cybershake workflows, using our base simulation parameter configuration, ALL_IN_GLOBAL outperforms all other heuristics. This is because with only 10 hosts, other heuristics cannot

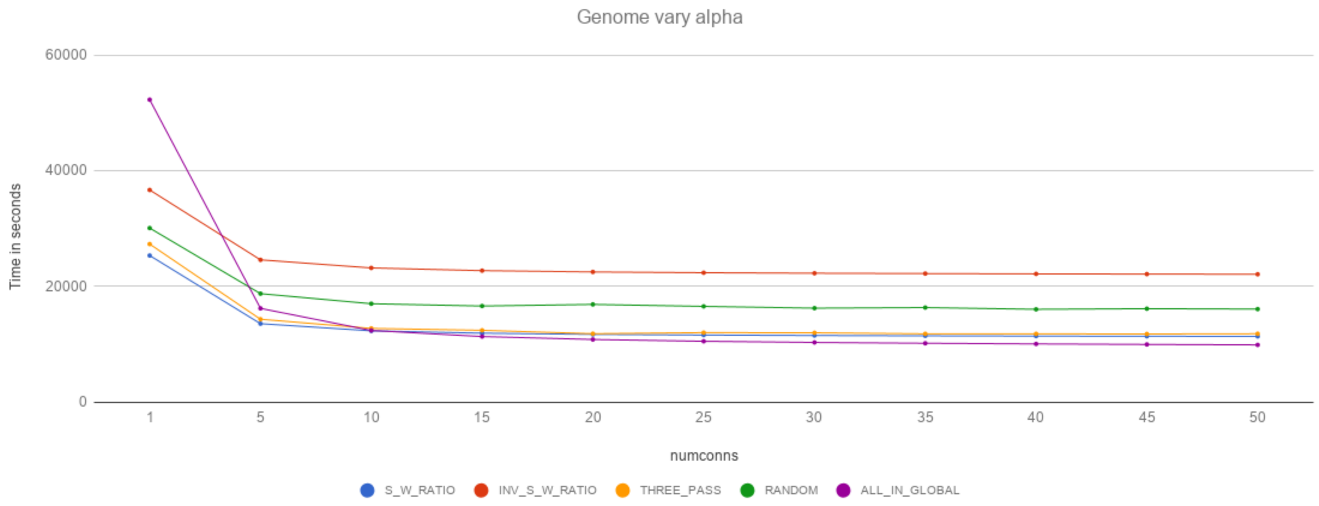

Figure 5.5: Average simulated application makespan vs. *num_conns* for Genome workflows

perform effective task replication. Expectedly, as *num_conns* increases, ALL_IN_GLOBAL further improves. Also, recall from the base results that S_W_RATIO, INV_S_W_RATIO and RANDOM, all had the same percentage runtime difference when compared to ALL_IN_GLOBAL. So, expectedly these three heuristics coincide on the same line even for higher values of *num_conns*. This is because of a particular feature of Cybershake workflows. These three heuristics are only different from one another in choosing the order of files to store in local storage. However, in Cybershake workflows, many tasks produce a single file. As a result, the order of choosing the files does not impact the schedule and thus in overall makespan. By contrast, THREE_PASS heuristic performs well compared to other heuristics though it is still outperformed by ALL_IN_GLOBAL at higher values of *num_conns*.

Outtree Results

Figure 5.7 shows results for Outtree workflows. Similar to Genome results, we can see that for lower values of *num_conns*, the application makespan achieved by ALL_IN_GLOBAL is significantly larger than that for other heuristics. As seen for Genome, as *num_conns* increases, ALL_IN_GLOBAL do significantly better compared to all other heuristics. One important difference that we see in this Outtree workflow compared to Genome is that the increase in *num_conns* does not help the other heuristics perform better. The reason behind this is that the other heuristics make significant use of local storages and use global storage sparingly. So, increasing *num_conns* to improve the network connection to the global storage

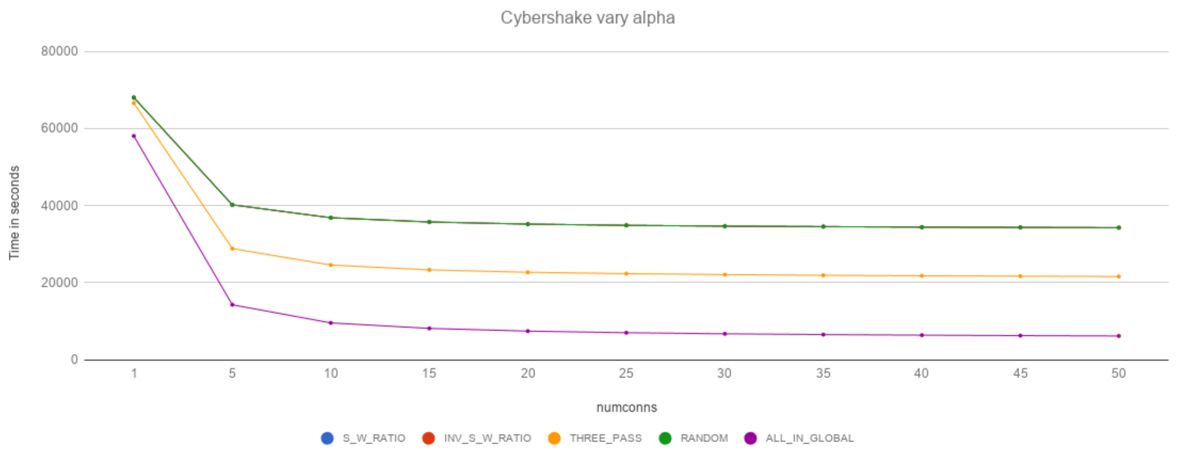

Figure 5.6: Average simulated application makespan vs. *num_conns* for Cybershake workflows

helps ALL_IN_GLOBAL significantly, but helps the other heuristics only marginally.

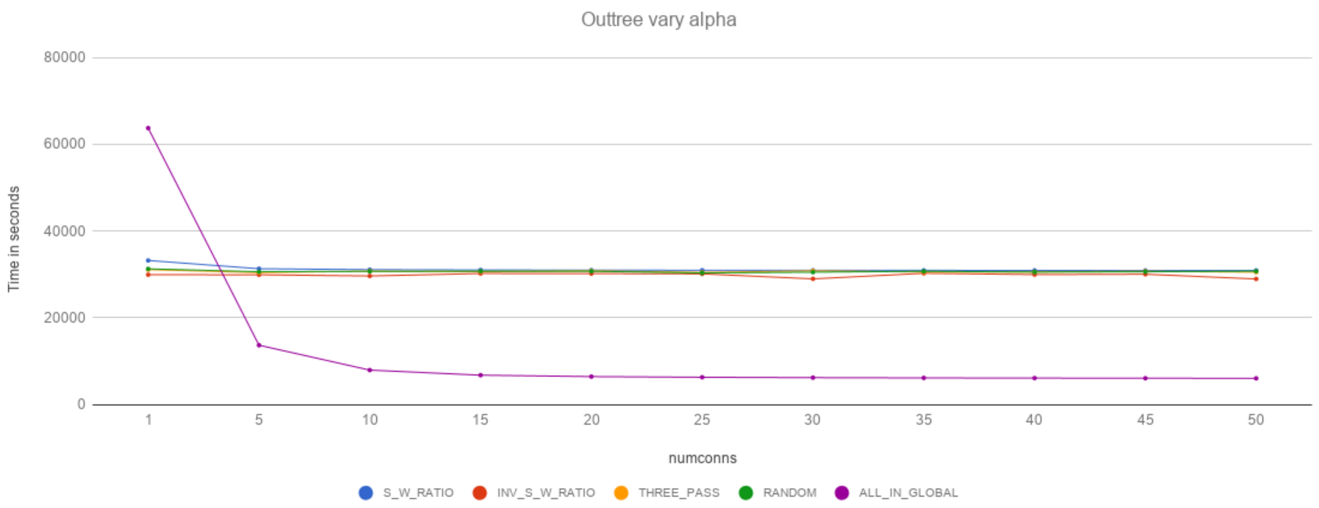

Figure 5.7: Average simulated application makespan vs. *num_conns* for Outtree workflows

5.2.3 Impact of CCR

In this section, we describe the impact of varying CCR. All the plots shown in this section show average simulated application makespans (in seconds) on the vertical axis, and CCR on the horizontal axis, which ranges from 1 to 20. In all these results, *num_conns* is equals to 1, h is equal to 10, and the number of tasks is 997 for Genome and 1000 for the other workflows as described in Section 5.1.

Genome Results

Figure 5.8 shows results for Genome workflows. We can see from the figure that at lower values of CCR, i.e., when communication is very costly compared to computation, ALL_IN_GLOBAL leads to worse results than all other heuristics. This is because with low CCR values the use of global storage is a bottleneck. However, as CCR increases, i.e., as I/O becomes relatively cheaper, ALL_IN_GLOBAL eventually outperforms RANDOM and INV_S_W_RATIO. If the value of CCR were to be increased further, ALL_IN_GLOBAL would eventually outperform all other heuristics.

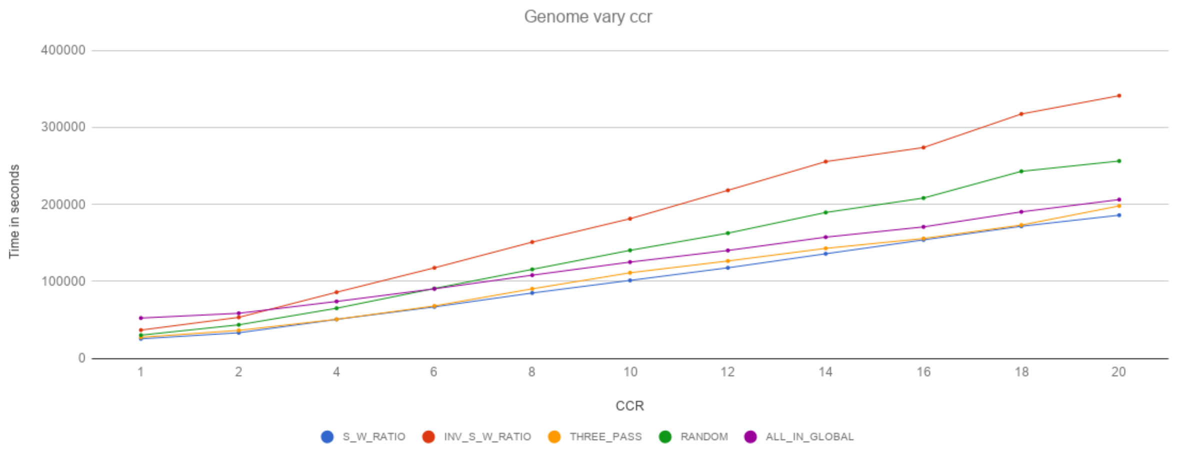

Figure 5.8: Average simulated application makespan vs. CCR for Genome workflows

Cybershake Results

Figure 5.9 shows results for Cybershake workflows. At lower values of CCR, when I/O is relatively costly, all the heuristics lead to similar results. However, as CCR increases, ALL_IN_GLOBAL outperforms all other heuristics, even more so that for the Genome results. Again, since all other heuristics attempt to make use of local storage more than global storage, making I/O cheaper helps ALL_IN_GLOBAL more than the other heuristics. Note that, RANDOM, S_W_RATIO and INV_S_W_RATIO all coincide on the same line, which is reasonable as described in Section 5.2.2.

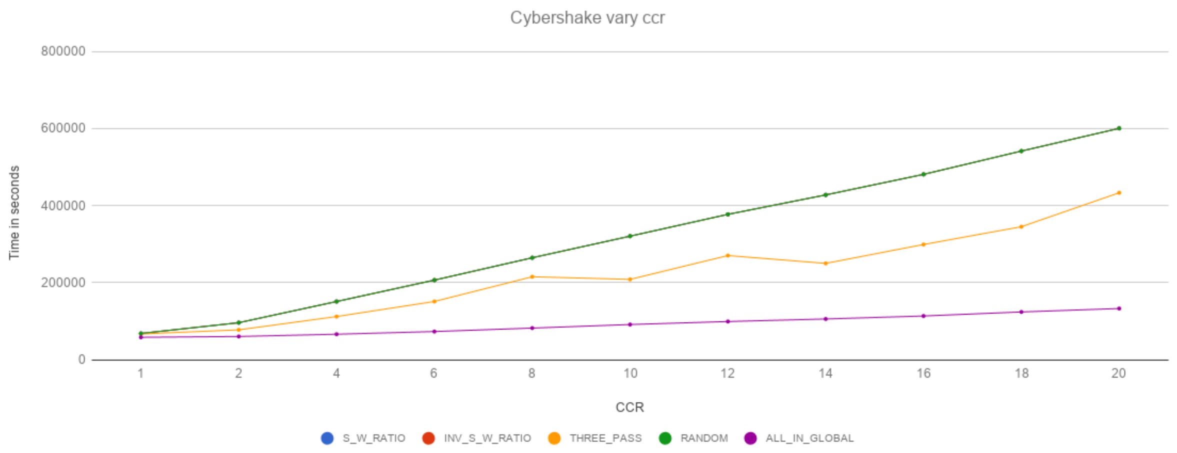

Figure 5.9: Average simulated application makespan vs. CCR for Cybershake workflows

Outtree Results

Figure 5.10 shows results for Outtree workflows. The trend is similar to that of Cybershake and Genome workflows, and the explanations for these results are also similar.

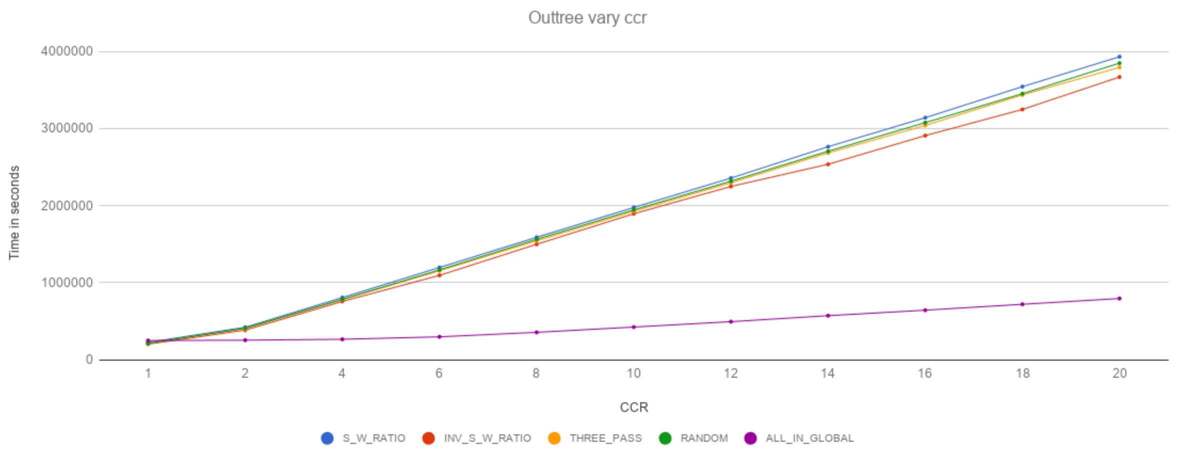

Figure 5.10: Average simulated application makespan vs. CCR for Outtree workflows

5.2.4 Impact of h

In this section, we describe the impacts of varying h. All the plots shown in this section show average simulated application makespans (in seconds) on the vertical axis, and h on the horizontal axis, which ranges from 2 to 200. In all these results, num_conns is equals to

1, CCR is equal to 1, and the number of tasks is 997 for Genome and 1000 for the other workflows as described in Section 5.1.

Genome Results

Figure 5.11 shows results for Genome workflows. We can see from the figure that ALL_IN_GLOBAL is outperformed by all other heuristics across the board. Initially, as h increases, all heuristics lead to lower makespans. But, after some point ($h \geq 20$), the makespan achieved by ALL_IN_GLOBAL dramatically increases. The reason is that the increase in the number of hosts helps to enhance the parallelism but at the same time, contention for the global storage bandwidth (recall that in these results num_conns is 1). Global storage bandwidth becomes a bottleneck, and makespans increase. This behavior is not as pronounced for the other heuristics because they strive to not use global storage.

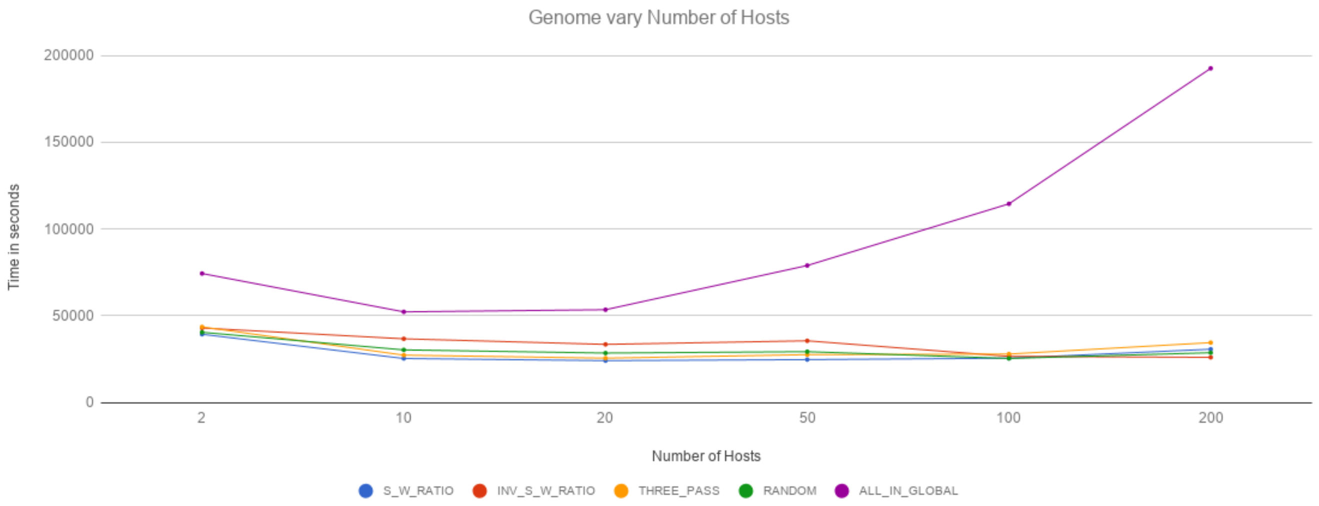

Figure 5.11: Average simulated application makespan vs. h for Genome workflows

Cybershake Results

Figure 5.12 shows results for Cybershake workflows. The general trend for ALL_IN_GLOBAL is similar as that seen in the Genome results, for the same results. The other heuristics, while they eventually outperform ALL_IN_GLOBAL when h becomes large enough, do not lead to as good relative performance. In fact, when h increases beyond some threshold, all heuristics see increases in makespan. This is because these other heuristics also mostly used global storage. The reason is the dense data-dependencies in and width of Cybershake workflows (see the example workflow Figure 5.3). More specifically, many of the single-level tasks (green in the figure) are forced to read their input from global storage because, with only h =10 hosts, the replicas of the entry tasks must write many of their output files to global storage.

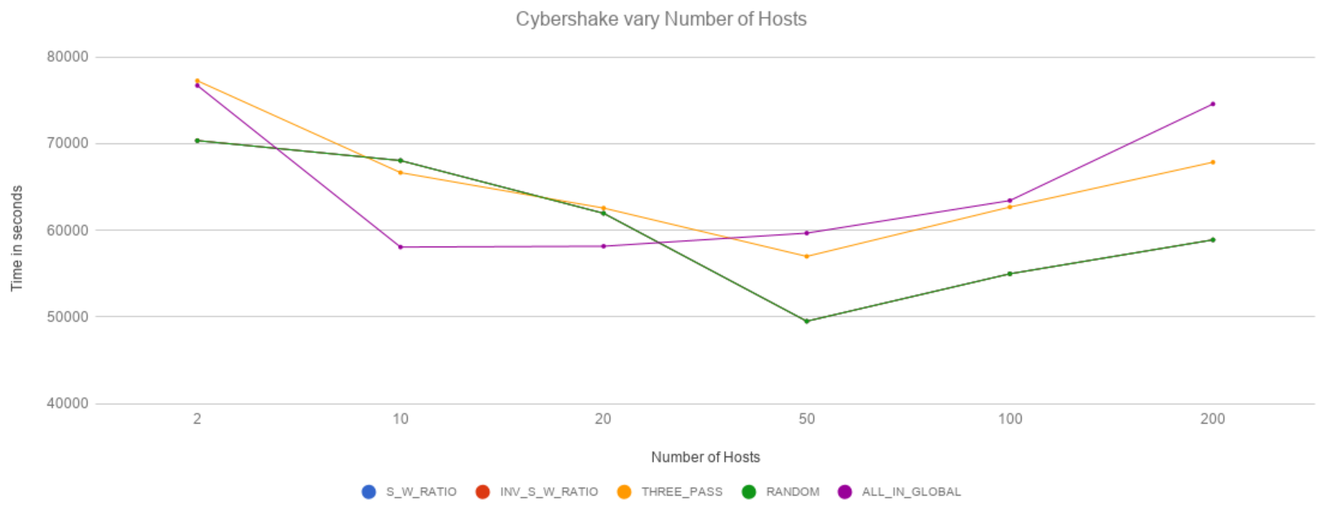

Figure 5.12: Average simulated application makespan vs. h for Cybershake workflows

Outtree Results

Figure 5.13 shows results for Outtree workflows. These results are similar to those for Genome workflows, with all heuristics benefiting from more hosts by ALL_IN_GLOBAL suffering from the global storage bandwidth bottleneck.

5.3 Real-World Methodology

We implemented a software prototype that drives the execution of workflow applications on a real-world cluster using our proposed heuristics. This prototype re-uses the "in-simulation" implementation of the heuristics and translates simulated activities into real-world activities (to execute computation on the cluster's compute nodes, and to read/write files to/from local/global storage, which on our target cluster consists of local disks and of a Lustre distributed file system).

We use the UHHPC Cray cluster at the University of Hawaiʻi at Mānoa as a testbed to test our heuristics on a real cluster. We attempt to repeat (a subset of) our simulation

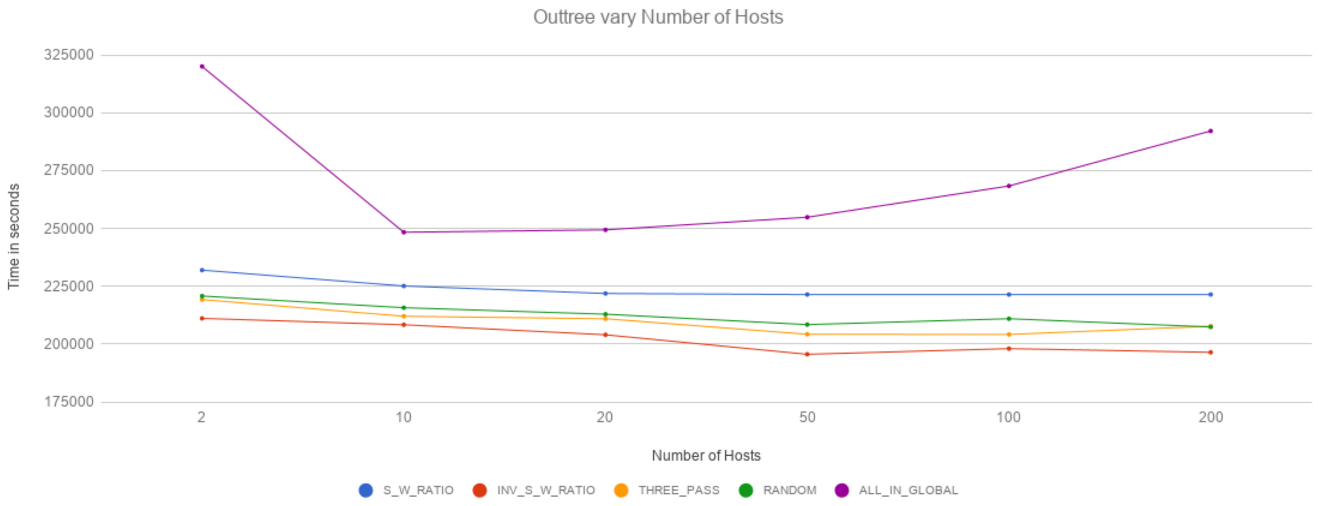

Figure 5.13: Average simulated application makespan vs. h for Outtree workflows

experiments but in real life, for the same seven types of workflows. Note that the goal here is not necessarily to compare simulated to real makespans, but rather to see whether trends seen in simulation seem to translate to the real world. Such comparison should be possible, but unfortunately, we encountered difficulties with the LLNL cluster that we initially used to perform benchmark experiments. Therefore, although our simulations are instantiated based on the hardware characteristics of the LLNL cluster, our account on that machine was abruptly suspended in the middle of us obtaining our real-world results. As an emergency measure, we have ported our real-world experiments to the UHHPC cluster. Note that the preliminary, but incomplete, results we obtained on the LLNL cluster, showed the same trends as our results on the UHHPC cluster. In both of these clusters, benchmark experiments show that $num\_conns = 1$.

In our real-world experiments, so as to feasibly obtain results in a timely fashion, we set the number of tasks in each workflow types to be 100 and the number of instances of each workflow types to be 2 with task execution times sampled from a uniform random distribution with range [0, 3600]. Note that, as a result, there are more opportunities for task replication than in our simulation results (i.e., the tasks / hosts ratio is smaller). This modifies some our results, as explained in upcoming sections. Also to make sure that we do not overload the storage system on compute hosts of UHHPC Cray cluster, we sample file sizes from a uniform random distribution with range [10KiB, 5MiB]. This means that we are not executing actual workflow applications, which would be too time consuming and would require us to install deep and complex scientific software stacks, but instead execute "mock" such applications in which data is randomly generated and computation (on space-shared hosts) is simulated by sleeps.

5.4 Real-World Results

5.4.1 Base Results

DAX	ALL_IN_GLOBAL	S_W_RATIO	INV_S_W_RATIO	THREE_PASS	RANDOM
Cybershake	0%	-18.531%	-19.801%	-11.177%	-19.732%
Montage	0%	-6.801%	-4.274%	-3.638%	-4.246%
Genome	0%	-40.621%	-41.291%	-39.105%	-39.566%
Intree	0%	-17.457%	-17.340%	-1.358%	-18.066%
Outtree	0%	-36.688%	-36.454%	-35.349%	-34.589%
ForkJoin Seq1	0%	-9.484%	-8.593%	-6.044%	-8.932%
ForkJoin Seq2	0%	-39.588%	-40.053%	-35.396%	-41.310%

Table 5.2: Average real-world application makespan differences relative to ALL_IN_GLOBAL for all heuristics (Base Results: $num\_conns = 1$, $CCR = 1$, $h = 10$).

As in Section 5.2, we first obtain a set of "base", in this case with $CCR = 1$, $num\_conns = 1$ (which is due to our hardware platform), $h = 10$. Table 5.2 shows average makespan differences relative to ALL_IN_GLOBAL, averaged over 2 instances for each workflow types.
Cybershake – Unlike with the simulation results, on our cluster all heuristics perform better than ALL_IN_GLOBAL. Recall that in our base simulation settings, our workflows had 1,000 tasks but the number of hosts was only 10. As a result, and as explained in Section 5.2, our heuristics could not do enough task replication to alleviate loss of parallelism due to using local storage. However, in this experiment on our real cluster, executing 100 tasks on 10 hosts allows enough task replication which consequently mitigates this loss in parallelism.
Montage – Again, unlike for simulation results, we see that all the heuristics outperform ALL_IN_GLOBAL. The explanation is the same as for Cybershake workflows, i.e., with fewer tasks task replication can be used more effectively to mitigate the loss of parallelism due to using local storage.

Genome – Recall that Genome workflows have many single-parent-single-child structures, and is thus very amenable to effective use of local storage. For this reason, all heuristics performed well in our simulation results, and thus with fewer tasks we also see significant performance improvements for all the heuristics compared to ALL_IN_GLOBAL on our cluster.

Intree – For these workflows all the heuristics except THREE_PASS do significantly well compared to ALL_IN_GLOBAL. The reason that THREE_PASS does not perform as well is because of the trade-off THREE_PASS encounters while deciding to use global storage for load-balancing instead of using local storage. Even if all the ready tasks at hand can use local storages for their I/O, THREE_PASS may decide not to do so to load balance the tasks. This in turn hurts makespan in the case of Intree workflows, which is because these workflows have first levels that consist of large numbers of independent tasks, and many of these tasks have a single child. Yet, for load-balancing reasons, THREE_PASS may decide to "slow down" these tasks by forcing them to use global storage. Note that when we construct Intree workflows for a given number of tasks, the number of single-parent-single-child structures varies. This is why this effect wasn't seen as sharply in our simulation results due to the workflow containing 1,000 tasks (but note that in those simulation results also THREE_PASS did not do as well as the other heuristics).

Outtree and ForkJoinSeq1 – Results for these workflows are in line with simulation results, i.e., all the heuristics perform significantly well compared to ALL_IN_GLOBAL.

ForkJoinSeq2 – Unlike in simulation, we see that all the heuristics performing relatively well compared to ALL_IN_GLOBAL. In simulation, only THREE_PASS had a non-marginal improvement. The reason for this is again the lower number of tasks in the workflows, which makes the use task of replication more feasible to mitigate loss of parallelism.

5.4.2 Simulation vs. Real-World Results

One interesting question is that of how accurately our simulation results match up with our real-world results. We have thus re-run simulations using the hardware characteristics of the UHHPC cluster ($b = 400$ MB/s, $B = 100$ MB/s) and with the same workflow parameters for the base results presented in the previous section (100 tasks, 10 hosts, CCR =1). Results are shown in Table 5.3. Ideally, these results would perfectly match up with the real-world results in Table 5.2. What we observe instead is that the average makespan differences for all heuristics relative to ALL_IN_GLOBAL, while sometimes different in magnitude, show the same trends as the real-world results, and thus lead to the same conclusions. Differences in

DAX	ALL_IN_GLOBAL	S_W_RATIO	INV_S_W_RATIO	THREE_PASS	RANDOM
Cybershake	0%	-12.702%	-12.647%	-0.611%	-12.645%
Montage	0%	-0.0293%	-0.0293%	-3.520%	-0.0293%
Genome	0%	-72.215%	-73.623%	-69.695%	-73.037%
Intree	0%	-20.336%	-18.925%	-5.420%	-19.556%
Outtree	0%	-1.545%	-7.659%	-5.255%	-4.910%
ForkJoin Seq1	0%	-12.609%	-13.727%	-12.691%	-13.106%
ForkJoin Seq2	0%	-5.092%	-5.670%	-8.975%	-5.350%

Table 5.3: Average makespan differences relative to ALL_IN_GLOBAL for all heuristics run with UHHPC cluster parameters.

magnitude are due to the usual simulation bias problem, i.e., instantiating simulation models that match particular hardware/software stacks is not straightforward. Such simulation "callibration" is known to be challenging, and is typically done in ad-hoc manners using labor-intensive trial-and-error approaches based on extensive benchmarking of the target platform. Automating such callibration is actually an open question. Regardless, we leave such callibration for future work, especially given that our simulation results corroborate conclusions drawn our from real-world results.

5.4.3 Impact of CCR

To have a comparative analysis of results from simulation and a real cluster, in this section, we present the impact of varying CCR for the same three workflow types– Genome, Cybershake and Outtree. All the plots shown in this section show average real-world application makespans (in seconds) on the vertical axis, and CCR on the horizontal axis, which ranges from 1 to 48. In all these results, *num_conns* is equals to 1, h is equal to 10, and the number of tasks is 100 for all workflow types.

Genome Results

Figure 5.14 shows results for Genome workflows from real-world executions on a real cluster. We can see that even for higher value of CCR, all other heuristics perform relatively well compared to ALL_IN_GLOBAL. Recall that Genome having significant amount of single-parent-single-child substructures is a suitable workflow that favors the use of local storages.

Also, recall that in these real-world executions, we are using 100 tasks and 10 hosts and thus we have enough replication. As a result, all the heuristics perform relatively well compared to ALL_IN_GLOBAL even for higher values of CCR. CCR should be further increased significantly for ALL_IN_GLOBAL to outperform all other heuristics.

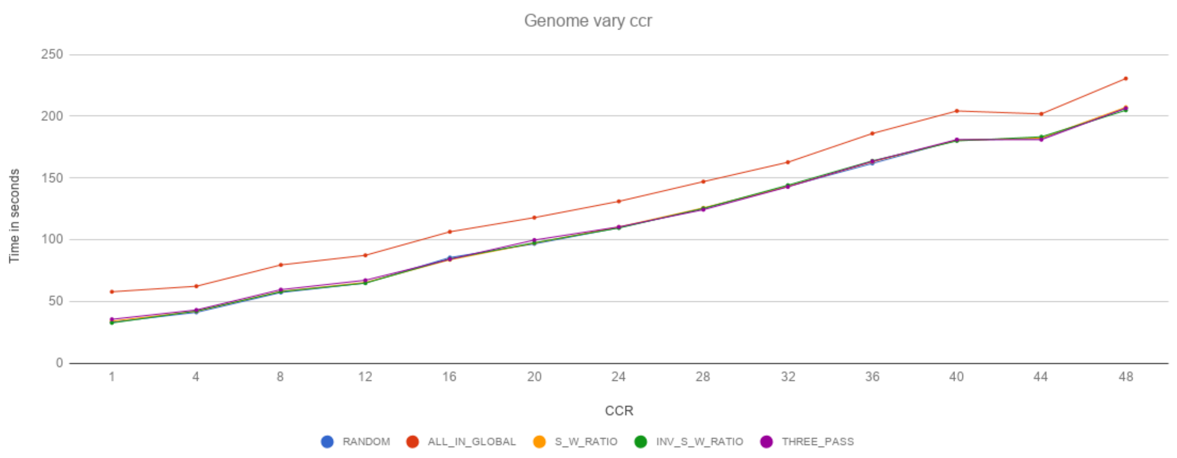

Figure 5.14: Average real-world application makespan vs. CCR for Genome workflows

Cybershake Results

Figure 5.15 shows results for Cybershake workflows from real-world executions on the real cluster. At lower values of CCR, when I/O is relatively costly, because of enough task replication, all other heuristics outperform ALL_IN_GLOBAL. However, as CCR increases ALL_IN_GLOBAL is in line with all other heuristics.

Outtree Results

Figure 5.16 shows results for Outtree workflows from real-world executions on the real cluster. The trend is similar to that of Cybershake workflows and the explanations for these results are also similar.

5.5 Discussion

From our experimental results in both simulation and on a real cluster, we can see that use of local storage reduces I/O time, which consequently lowers overall application makespan. However, this can only be achieved if sufficient task replication can be performed. Recall

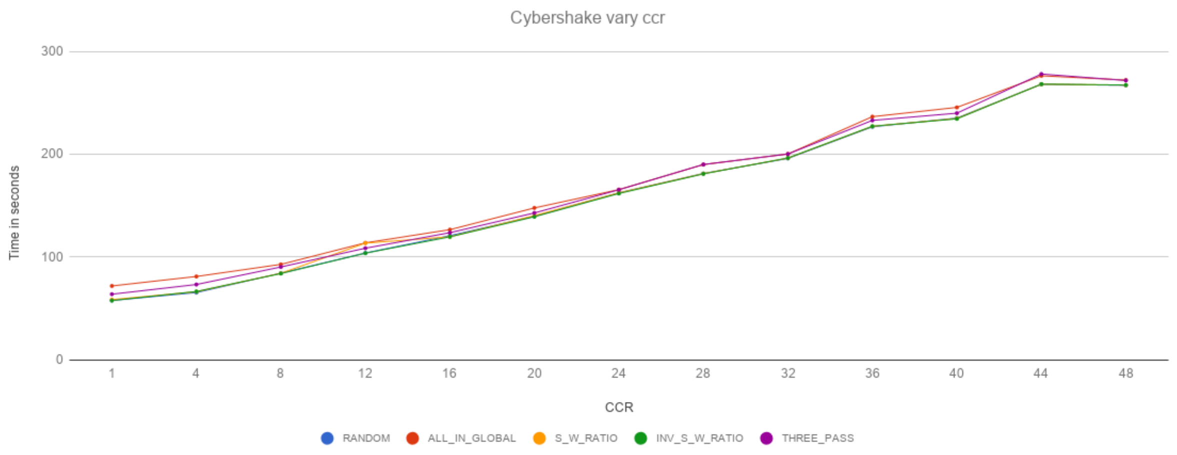

Figure 5.15: Average real-world application makespan vs. CCR for Cybershake workflows

the discussion of the base results for Cybershake workflows discussed in Section 5.2.1: if we do not have enough hosts to perform task replication then using local storage actually hurts performance. A corollary of this finding is that if a workflow structure is deep , i.e., many edges on the paths from the entry tasks to the exit tasks, relative to the total number of tasks, then local storage can be used more effectively. This is because for a deeper graph, the number of ready tasks is lower at each level, and thus task replication can be applied more often, which limits loss of parallelism. By contrast, for shallower workflows larger numbers of hosts are required to allow for a significant amount of replication. This is again supported by the base results for Cybershake workflows discussed in Section 5.2.1, as Cybershake workflows are shallow and wide.

Perhaps unsurprisingly, we find that the (relative) performance of our heuristics depend significantly on the structure of the workflows. For instance, the base results presented in Section 5.2.1 show clearly that results vary enormously between workflows (i.e., between rows of Table 5.1). A contributing factor to these variations is the existence and number of single-parent-single-child structures, as these structures make the use of local storage desirable in most of the cases. This if, for instance, the reason while all our heuristics perform well for Genome workflows as they contain many such structures (see Figure 5.1).

In many cases, both in simulation and with a real cluster, we found that the RANDOM heuristic performs well when compared to ALL_IN_GLOBAL. This signifies that use of local storage along with enough task replication is good enough to achieve better makespans even without proper prioritization of which files for local storage. This is particularly true for relatively low numbers of tasks and files, in which cases we find that RANDOM does very

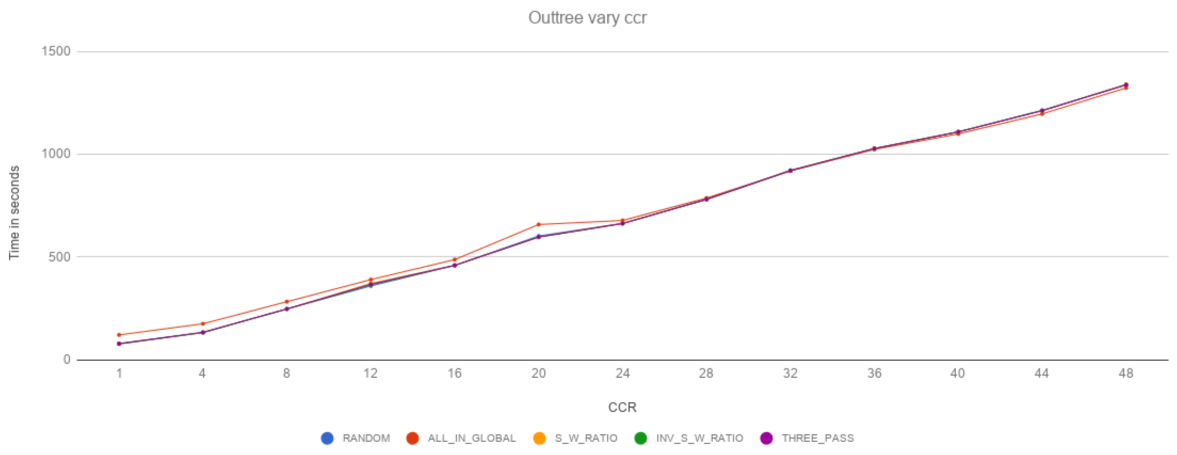

Figure 5.16: Average real-world application makespan vs. CCR for Outtree workflows

well. This is supported by comparative analysis of the base results between simulation and the real cluster, i.e., RANDOM seem to perform better in base case experiments on the real cluster (with 100 tasks) compared to the base case experiments in simulation (with 1,000 tasks).

Our results show expected trends regarding the sharing of global storage bandwidth. All heuristics benefit from a higher num_conns value, and the more a heuristic uses global storage the more it benefits. As a result, for high num_conns values, the ALL_IN_GLOBAL approach is the best. In practice, on two distinct HPC clusters, we found via benchmarking that $num\_conns = 1$, showing that ALL_IN_GLOBAL is likely not effective in practice. Furthermore, we found that all our proposed heuristics, across all workflow types, perform well when the workflows are more data-intensive. This is supported by the results in Section 5.2.3, which show that for low CCR values our heuristics vastly outperform ALL_IN_GLOBAL because they are able to use local storage effectively. Recall that ALL_IN_GLOBAL is the approach used in current state-of-the-art WMSs. Therefore, claim that these WMSs poorly support data-intensive workflows. However, our results show that minor modifications to their implementations, combined with a heuristic as simple as our RANDOM heuristics, could lead to substantially better performance for these workflows.

CHAPTER 6
CONCLUSION

Scientific workflows nowadays are, increasingly, being used to represent crucial applications in most fields of science and engineering. These applications, due to the increase in capacity of storage systems and of the computing capabilities of computers, tend to make use of large data files, relative to the amount of computation they perform, thus making them data-intensive. It has thus become necessary to reduce I/O overhead while executing such workflows on HPC clusters. HPC clusters, however, often use commodity interconnects with relatively low I/O bandwidth to connect compute hosts with some global storage system. As a result, the global storage system bandwidth can be a performance bottleneck for data-intensive workflows. In spite of this bottleneck, state-of-the-art Workflow Management Systems (WMSs) always use global storage while executing workflows on HPC clusters. In this thesis we have explored an alternative approach in a view to reducing I/O overhead.

Our approach consists of using, whenever possible, the local storage system available at each compute hosts in most HPC clusters. The connection bandwidth from each compute host to its local storage can be orders of magnitude higher than that to the global storage system. Therefore, storing data files in local storage can significantly reduce I/O overhead. Such local storage, however, is of limited capacity and is also private to a single compute host. As a result, not only is there an issue of which files should be selected to be stored in local storage due to capacity constraints, but also an issue of loss in parallelism because local storage can only be accessed by a single lost.

6.1 Summary of Contribution

In this thesis, we have formalized an off-line workflow scheduling problem that accounts to both local and global storage in homogeneous clusters, with the objective of minimizing workflow makespan. This scheduling problem, like most scheduling problems, is NP-hard, and thus we approached it with heuristics. These heuristics compute the schedule dynamically at runtime, i.e., they schedule tasks as they become ready on hosts as they become idle. More specifically, our propose heuristics make:

- **Storage decisions** – Given the capacity constraint of local storages and since each local storage is only accessible to one host, it is typically not possible to use local storage for holding all data files. Therefore, our heuristics decide to use local storage whenever feasible (in terms of capacity and in terms of feasible workflow executions) and deemed worthwhile given the data-intensiveness of the workflow.

- **Task replication decisions** – Using local storage may lead to worse workflow performance due to loss of parallelism. So, in addition to making storage decisions, our heuristics perform task replication. Task replication allows copies of input/output files of tasks to be available on different compute hosts, thus making it possible for independent tasks to run on different compute hosts concurrently.

We have implemented a re-usable simulation framework to evaluate the execution of workflow applications in cluster settings with local and global storage. This framework implements our proposed heuristics, as well as the state-of-the-art "use only global storage" approach. This framework is re-usable for further research.

To the best of our knowledge, current WMSs do not have the ability to make decisions about where the output files of workflow tasks should be written in a cluster setting as they do not account for local storage at compute hosts. We have, thus, implemented a software prototype with this capability so as to evaluate our heuristics in the real-world. We have obtained experimental results both in simulation and on a real cluster. The broad finding is that using local storage can provide significant performance benefits in practice. More specifically, it is a good idea to use local storage if there is a balance between the number of hosts and the number of ready tasks (which depends on the workflow's parallelism) so that tasks can be sufficiently replicated to mitigate parallelism loss.

We have shown that although some of our proposed heuristic do well for particular workflow configurations, most likely a single simple greedy heuristic can be effective across the board of workflow configurations. In fact, a random heuristic can do surprisingly well in some scenarios.

6.2 Future Work Directions

We have proposed several heuristics that employ local storage while satisfying the storage limitations, attempting to use local storage only when it is "worth it", and avoiding situations that would make application execution infeasible. Our heuristics consider several factors while making such decisions (e.g., files sizes, runtime of workflow tasks, CCR of the workflow). However, our heuristics do not explicitly look for particular structural patterns in the workflow. Yet, our results show that particular patterns have a high impact of the (relative) effectiveness of our heuristics. Therefore, a promising research direction is to design heuristics that discover and account for particular structural patterns. While there is conceivable a large number of possible patterns, real-world workflow applications tend to reuse a limited number of them (i.e., chains, fork-joins, etc). It is thus possible that heuristics designed to discover and exploit only a few patterns could yield good results across a broad range of applications.

Regardless of the heuristic chosen for making storage decision, another promising direction is to "tech-transfer" our approach, as pioneered in the software prototype used to obtain the results in Section 5.4 into an actual WMS system. This should be done with only minor modification of the WMS implementation, namely, including the ability to specify for a task execution which files should be written to global storage and which files should be written to local storage. Typically, tasks in production workflows use file paths as command-line arguments, and thus adding this capability to a WMS could be completely straightforward. A bit more involved is adding the capability of tracking which file is stored where, including the fact that a file can be available at multiple locations (e.g., in global storage and in local storage at several hosts). Note that WMS typically has this capability for tracking the presence of files and file replicas in distributed storage infrastructures. It should thus be feasible to add the capability of tracking file replicas in global and local storage systems within a cluster. An obvious first target for this tech-transfer would be the popular Pegasus [15] WMS.

APPENDIX A
WORKFLOW TYPES

A.1 Genome

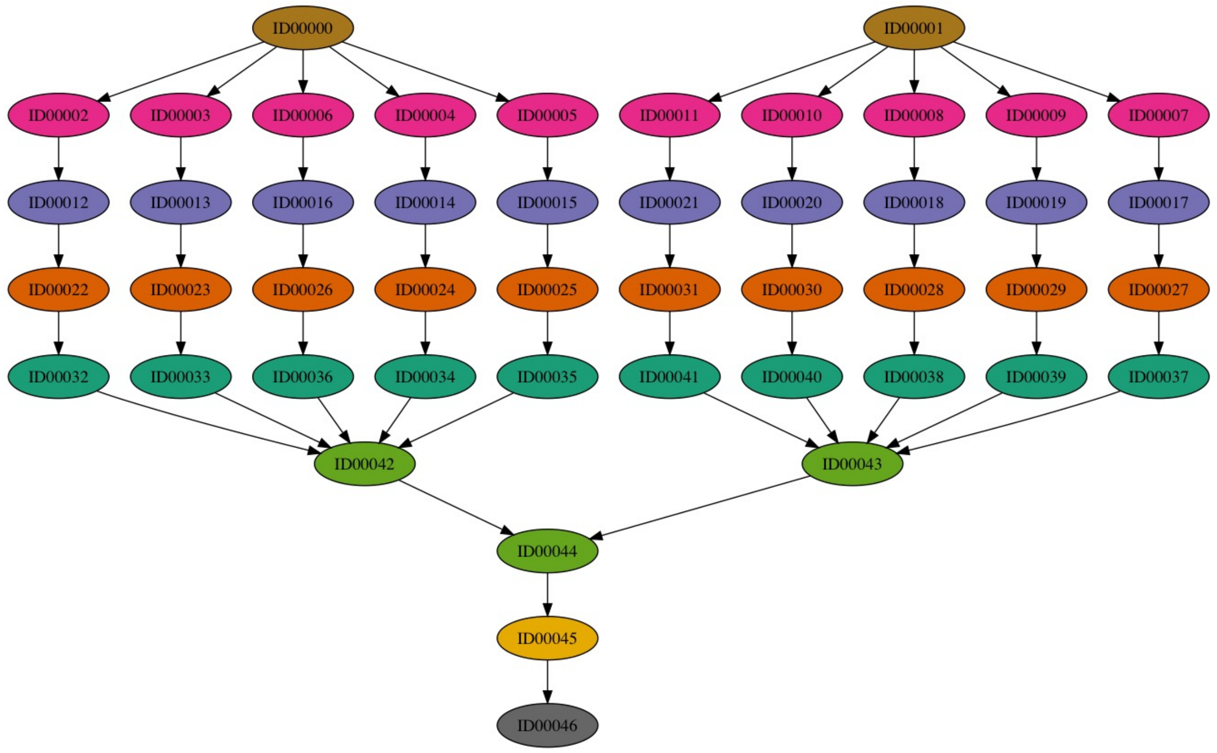

Figure A.1: An example instance of the Genome Workflow

A.2 Cybershake

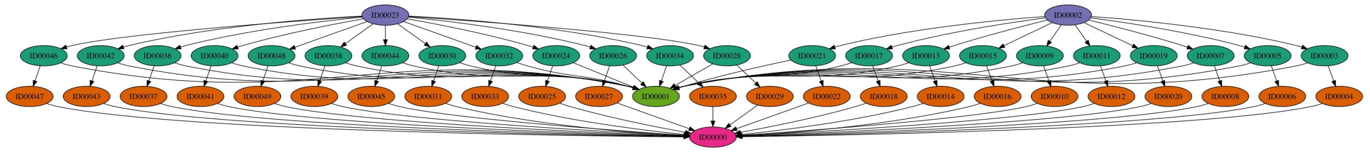

Figure A.2: An example instance of the Cybershake Workflow

A.3 Outtree

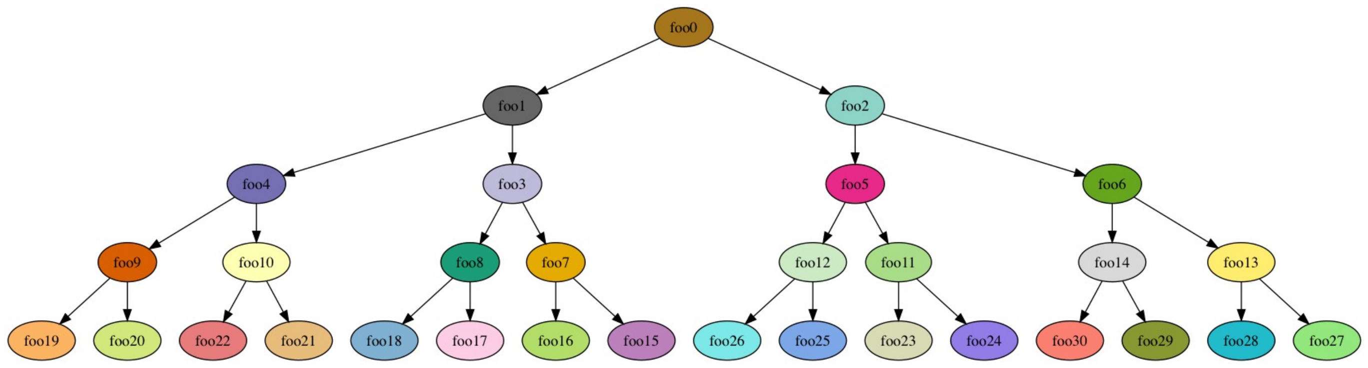

Figure A.3: An example instance of the Outtree Workflow

A.4 Intree

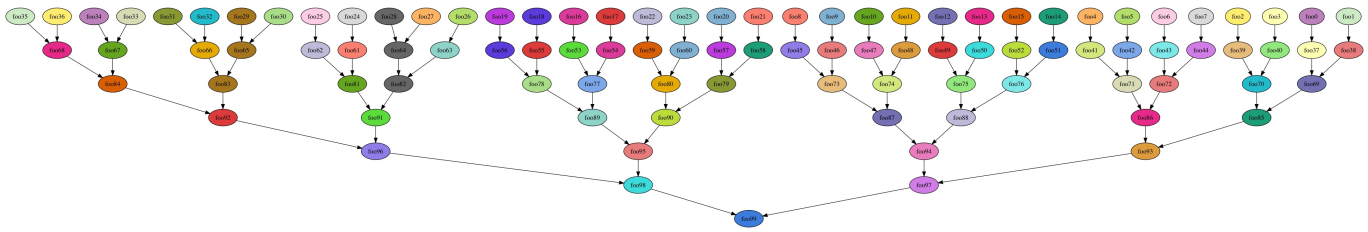

Figure A.4: An example instance of the Intree Workflow

A.5 Montage

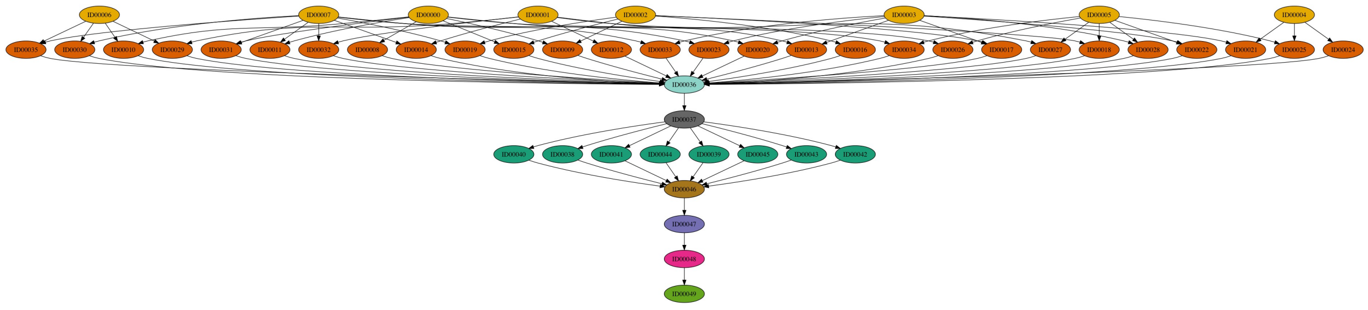

Figure A.5: An example instance of the Montage Workflow

A.6 ForkJoinSeq1

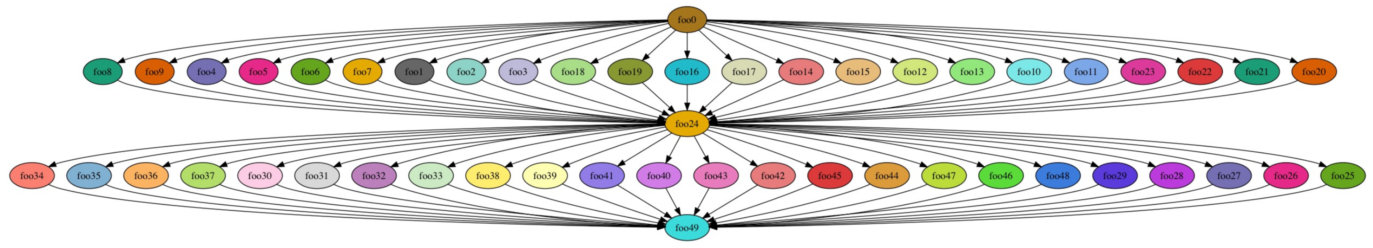

Figure A.6: An example instance of the ForkJoinSeq1 Workflow

A.7 ForkJoinSeq2

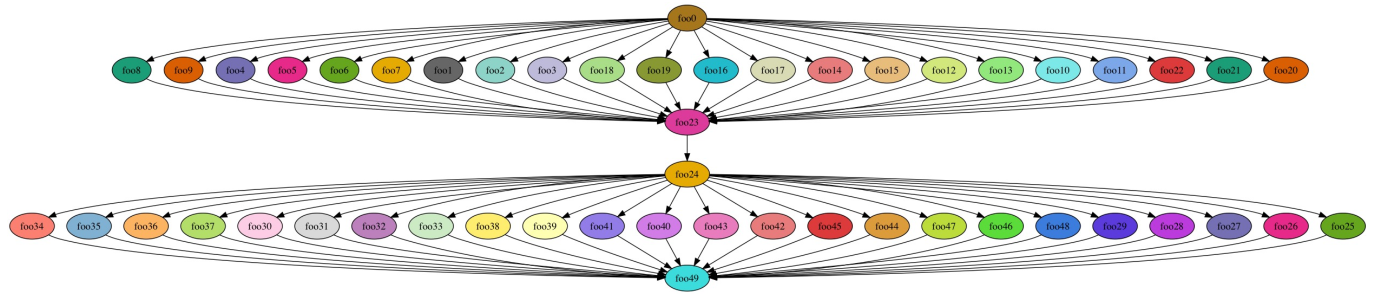

Figure A.7: An example instance of the ForkJoinSeq2 Workflow

APPENDIX B
SIMULATION RESULTS

B.1 Impact of *num_conns*

B.1.1 Intree Results

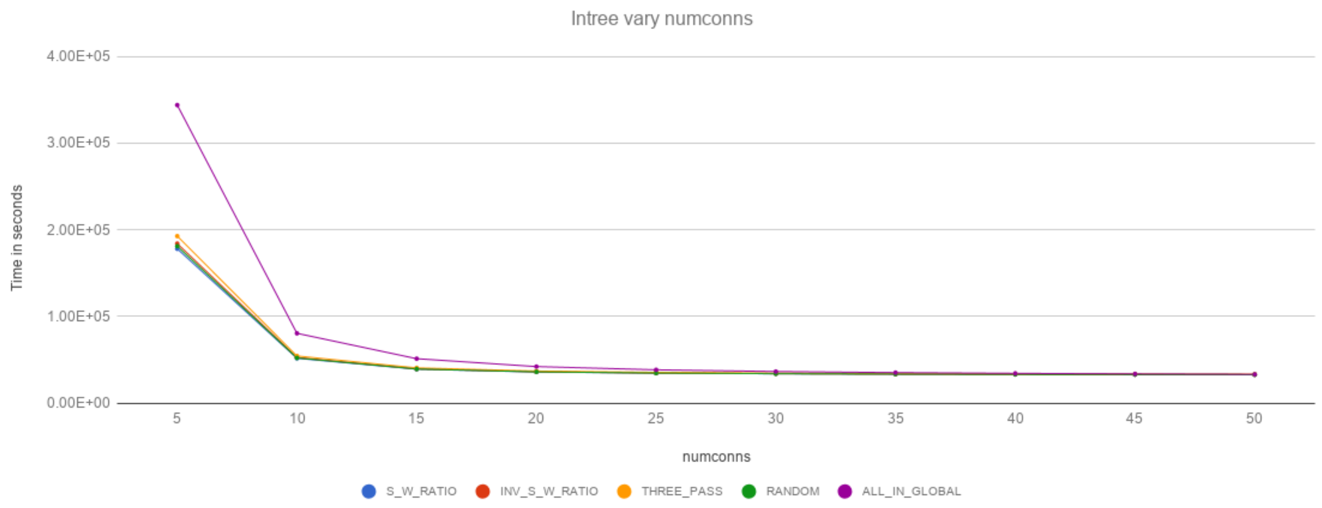

Figure B.1: Average simulated application makespan vs. *num_conns* for Intree workflows

B.1.2 Montage Results

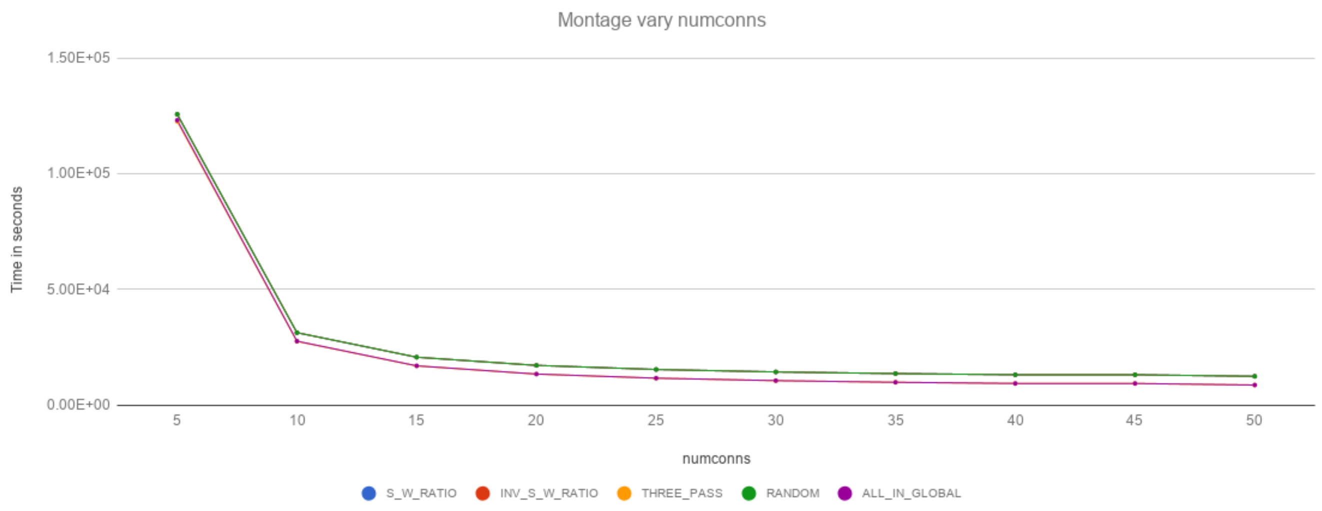

Figure B.2: Average simulated application makespan vs. *num_conns* for Montage workflows

B.1.3 ForkJoinSeq1 Results

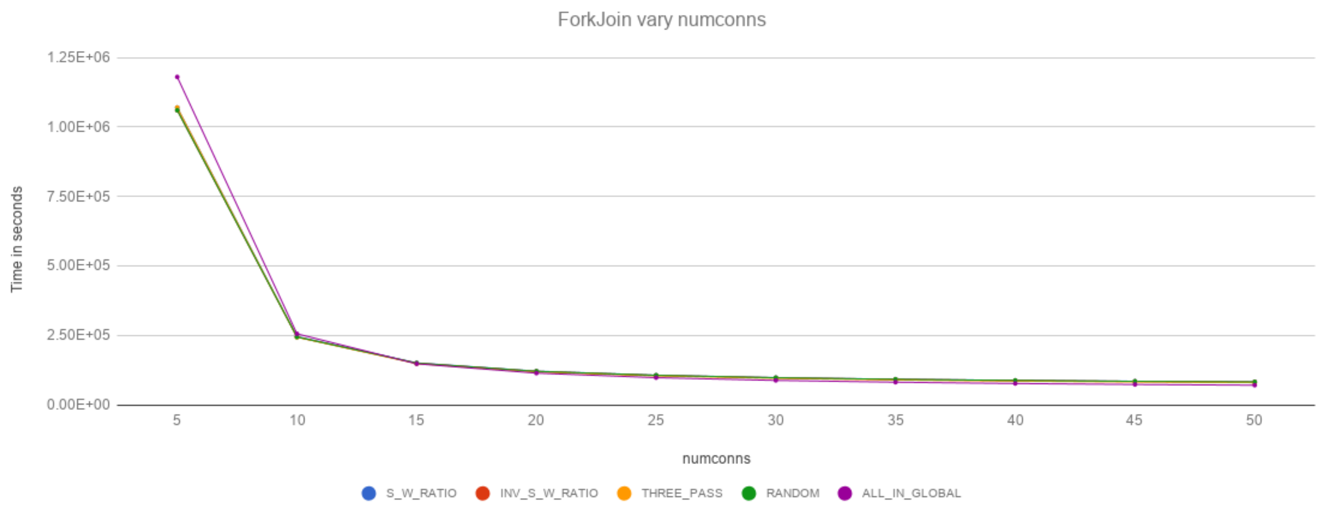

Figure B.3: Average simulated application makespan vs. *num_conns* for ForkJoinSeq1 workflows

B.1.4 ForkJoinSeq2 Results

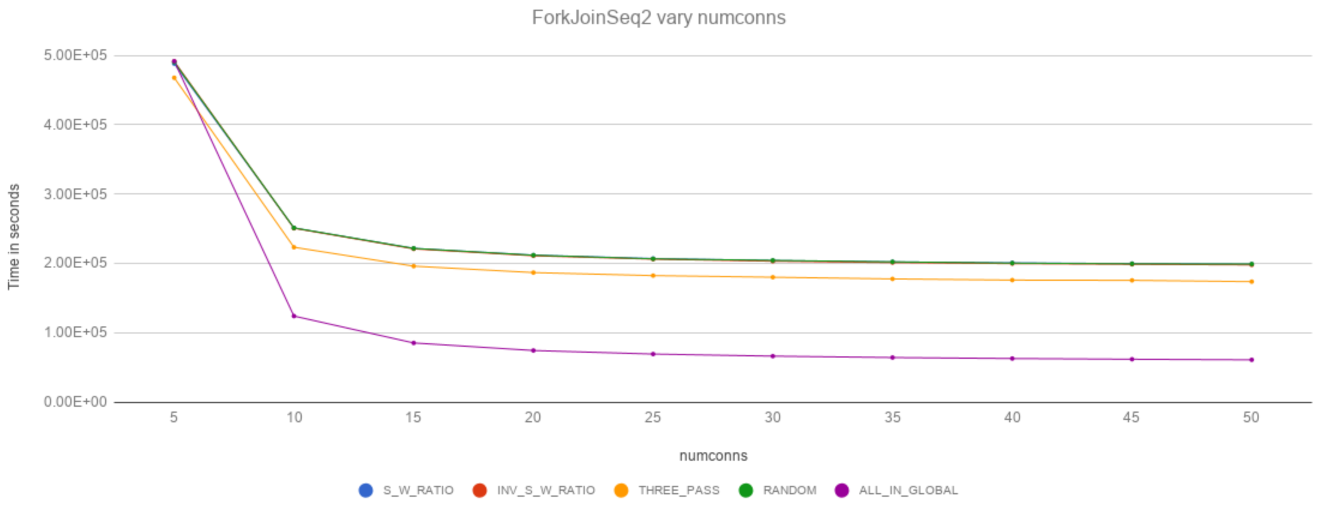

Figure B.4: Average simulated application makespan vs. *num_conns* for ForkJoinSeq2 workflows

B.2 Impact of CCR

B.2.1 Intree Results

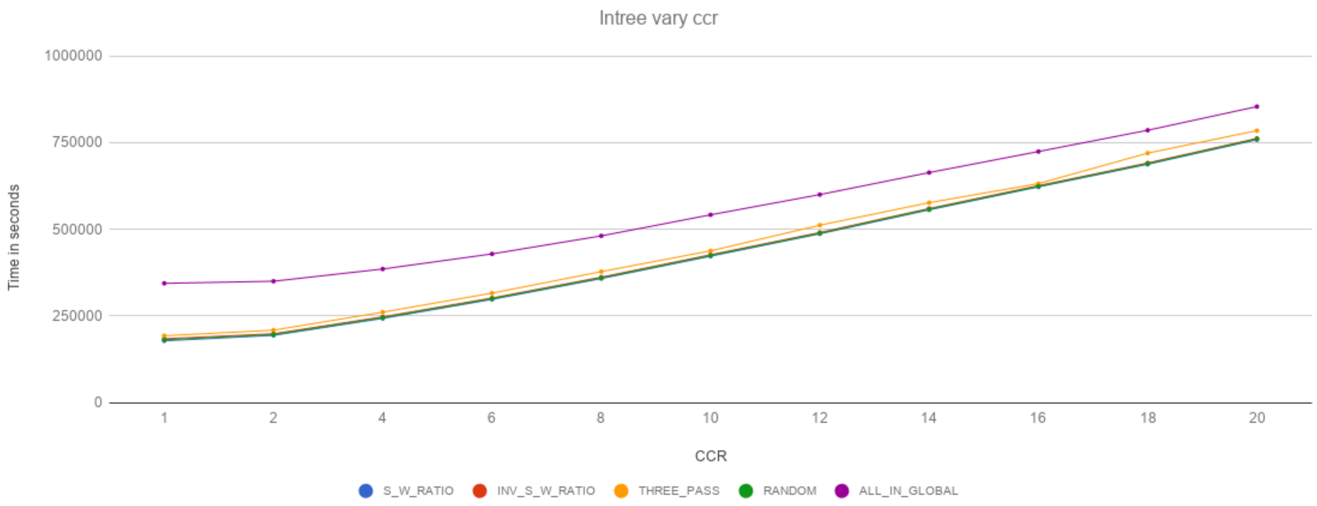

Figure B.5: Average simulated application makespan vs. CCR for Intree workflows

B.2.2 Montage Results

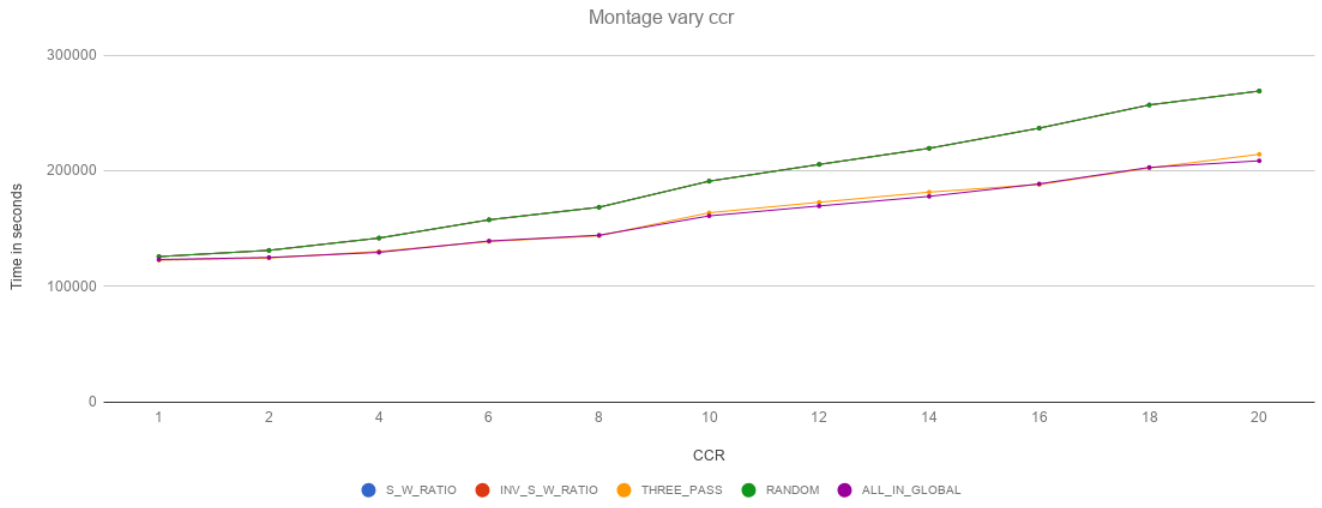

Figure B.6: Average simulated application makespan vs. CCR for Montage workflows

B.2.3 ForkJoinSeq1 Results

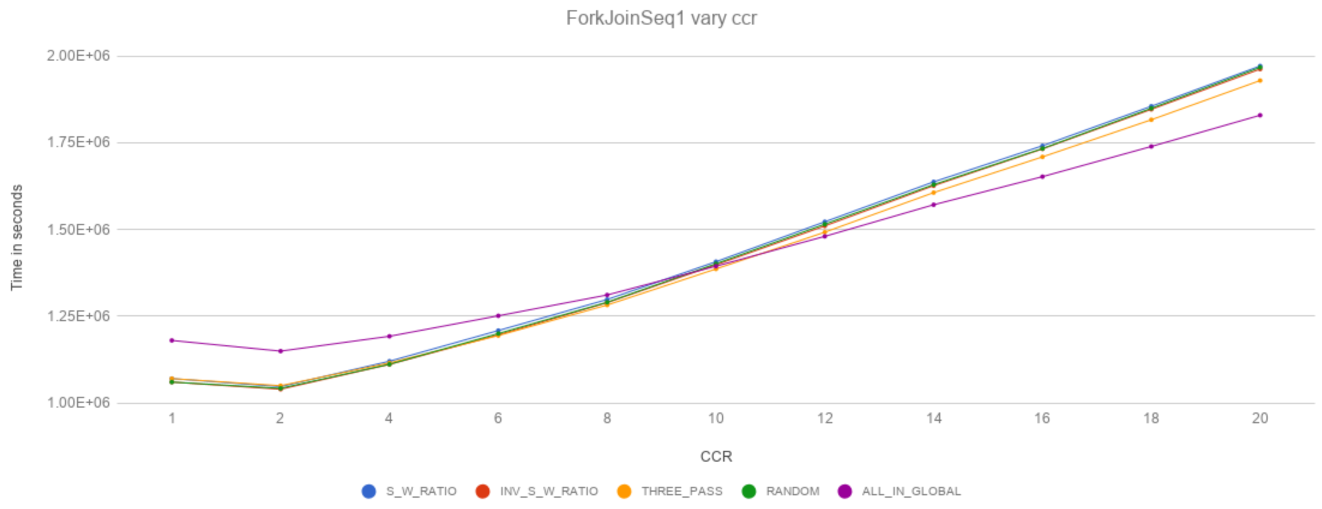

Figure B.7: Average simulated application makespan vs. CCR for ForkJoinSeq1 workflows

B.2.4 ForkJoinSeq2 Results

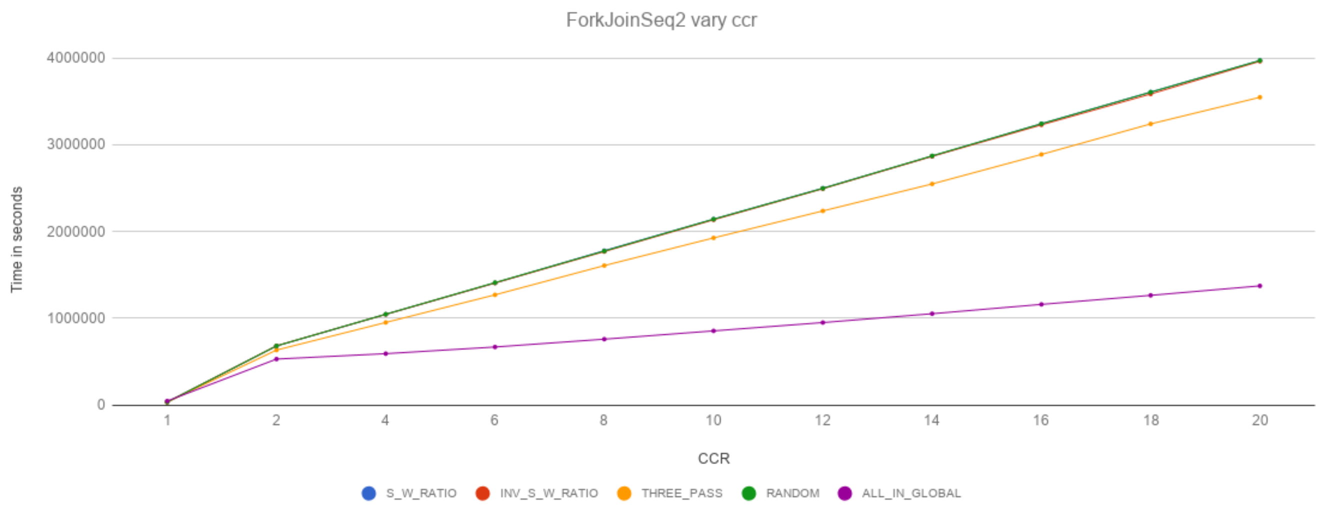

Figure B.8: Average simulated application makespan vs. CCR for ForkJoinSeq2 workflows

B.3 Impact of h

B.3.1 Intree Results

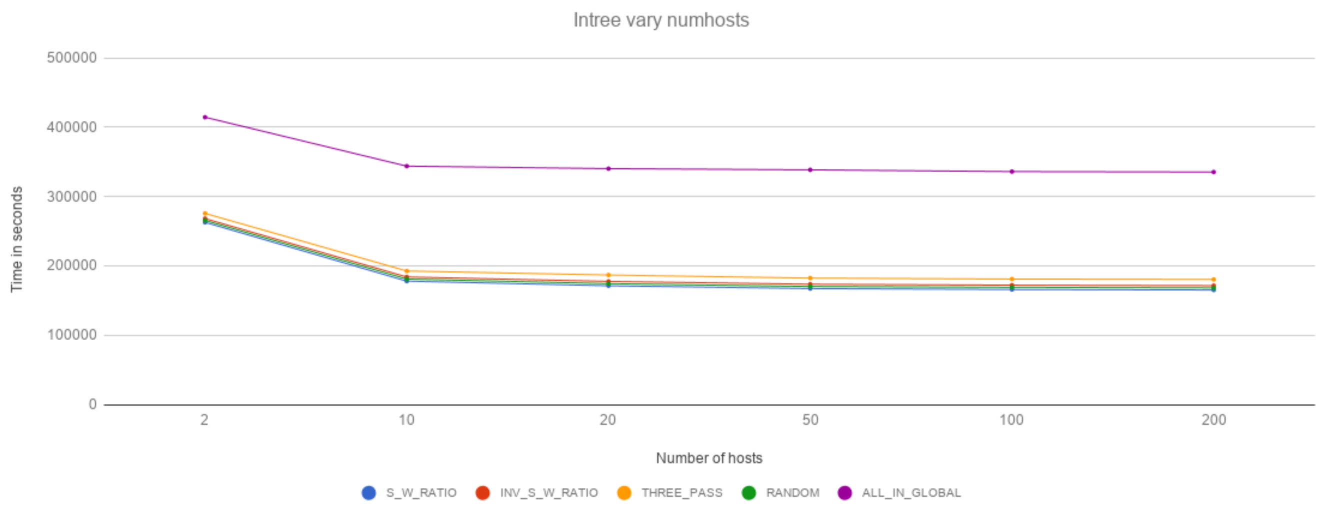

Figure B.9: Average simulated application makespan vs. h for Intree workflows

B.3.2 Montage Results

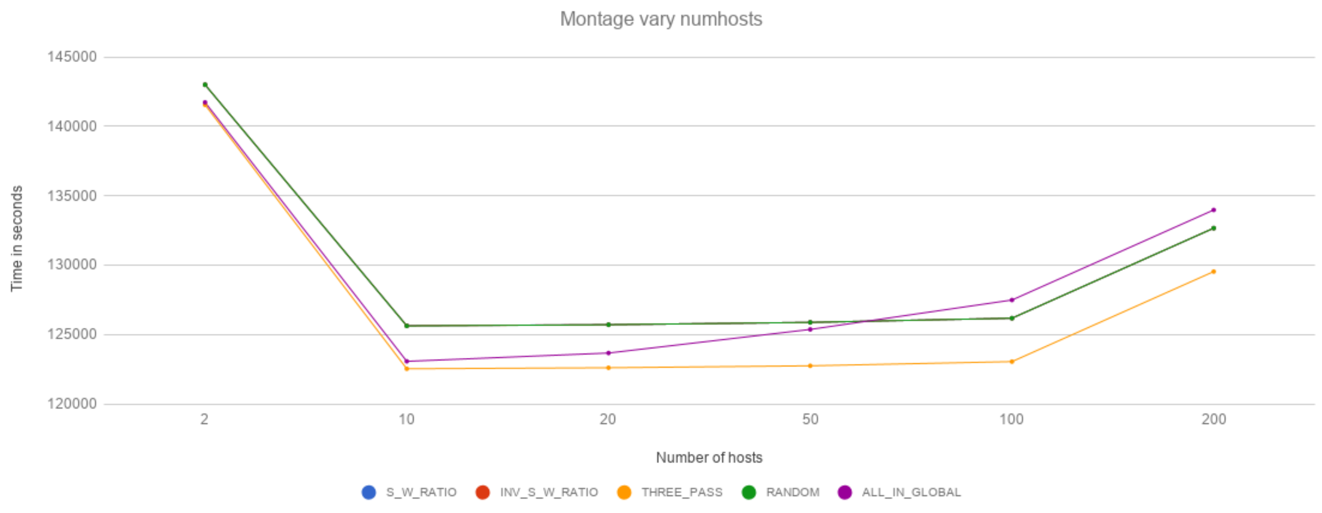

Figure B.10: Average simulated application makespan vs. h for Montage workflows

B.3.3 ForkJoinSeq1 Results

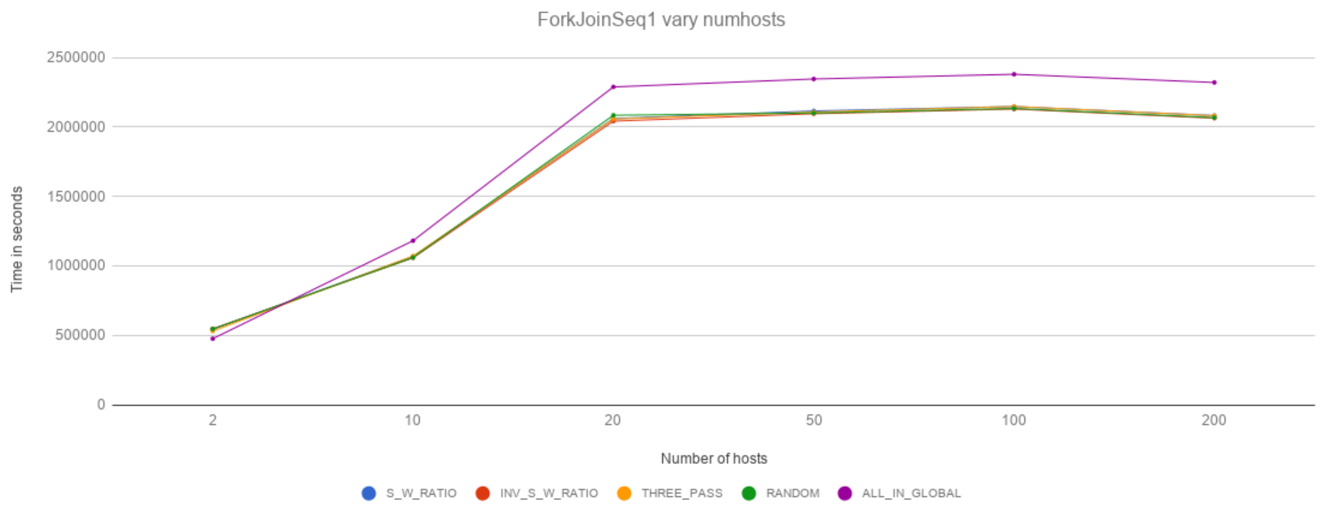

Figure B.11: Average simulated application makespan vs. h for ForkJoinSeq1 workflows

B.3.4 ForkJoinSeq2 Results

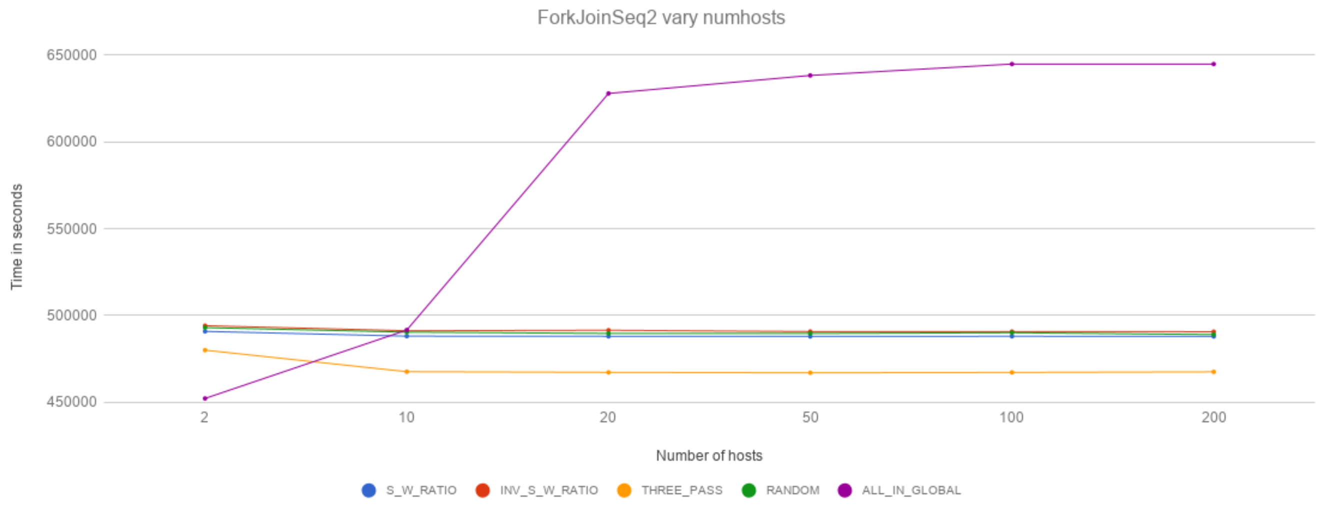

Figure B.12: Average simulated application makespan vs. h for ForkJoinSeq2 workflows

BIBLIOGRAPHY

[1] S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir, and S. U. Khan. Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 129–136, Dec 2014.

[2] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.

[3] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[4] G Bruce Berriman, Ewa Deelman, John C Good, Joseph C Jacob, Daniel S Katz, Carl Kesselman, Anastasia C Laity, Thomas A Prince, Gurmeet Singh, and Mei-Hu Su. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *Astronomical Telescopes and Instrumentation*, pages 221–232. International Society for Optics and Photonics, 2004.

[5] G Bruce Berriman, Gideon Juve, Ewa Deelman, Moira Regelson, and Peter Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *e-Science Workshops, 2010 Sixth IEEE International Conference on*, pages 1–7. IEEE, 2010.

[6] GB Berriman, JC Good, AC Laity, A Bergou, J Jacob, DS Katz, E Deelman, C Kesselman, G Singh, M-H Su, et al. Montage: a grid enabled image mosaic service for the national virtual observatory. In *Astronomical Data Analysis Software and Systems (ADASS) XIII*, volume 314, page 593, 2004.

[7] Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. Galaxy: a web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, pages 19–10, 2010.

[8] D. Bozdag, F. Ozguner, E. Ekici, and U. Catalyurek. A task duplication based scheduling algorithm using partial schedules. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 630–637, June 2005.

[9] Duncan A Brown, Patrick R Brady, Alexander Dietz, Junwei Cao, Ben Johnson, and John McNabb. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. In *Workflows for e-Science*, pages 39–59. Springer, 2007.

[10] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, et al. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, 76(6):428–446, 2010.

[11] Scott Callaghan, Philip Maechling, Ewa Deelman, Karan Vahi, Gaurang Mehta, Gideon Juve, Kevin Milner, Robert Graves, Edward Field, David Okaya, et al. Reducing time-to-solution using distributed high-throughput mega-workflows-experiences from scec cybershake. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 151–158. IEEE, 2008.

[12] Scott Callaghan, Philip Maechling, Patrick Small, Kevin Milner, Gideon Juve, Thomas H Jordan, Ewa Deelman, Gaurang Mehta, Karan Vahi, Dan Gunter, et al. Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *International Journal of High Performance Computing Applications*, pages 274–285, 2011.

[13] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[14] Tracy Craddock, Phillip Lord, Colin Harwood, and Anil Wipat. E-science tools for the genomic scale characterisation of bacterial secreted proteins. In *All hands meeting*, pages 788–795, 2006.

[15] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity,

Joseph C. Jacob, and Daniel S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[16] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Phil J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46(0):17–35, 2015.

[17] Epigenomics. `http://epigenome.usc.edu`.

[18] Thomas Fahringer, Radu Prodan, Rubing Duan, Jüurgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, et al. Askalon: A development and grid computing environment for scientific workflows. In *Workflows for e-Science*, pages 450–471. Springer, 2007.

[19] Paul Fisher, Harry Noyes, Stephen Kemp, Robert Stevens, and Andrew Brass. A systematic strategy for the discovery of candidate genes responsible for phenotypic variation. In *Cardiovascular Genomics*, pages 329–345. Springer, 2009.

[20] Rob Gaizauskas, Neil Davis, George Demetriou, Yikun Guod, and Ian Roberts. Integrating biomedical text mining services into a distributed workflow environment. In *Proceedings of UK e-Science All Hands Meeting*, 2004.

[21] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

[22] Robert Graves, Thomas H Jordan, Scott Callaghan, Ewa Deelman, Edward Field, Gideon Juve, Carl Kesselman, Philip Maechling, Gaurang Mehta, Kevin Milner, et al. Cybershake: a physics-based seismic hazard model for southern california. *Pure and Applied Geophysics*, 168(3-4):367–381, 2011.

[23] M. Horiuchi and K. Taura. Acceleration of data-intensive workflow applications by using file access history. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 157–165, Nov 2012.

[24] Kepler/clotho integration. `http://sourceforge.net/projects/keplerclotho`.

[25] Y. K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 531–537, Mar 1998.

[26] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[27] Montage. `http://montage.ipac.caltech.edu`.

[28] A. L. Rosenberg and M. Yurkewych. Guidelines for scheduling some common computation-dags for internet-based computing. *IEEE Transactions on Computers*, 54(4):428–438, April 2005.

[29] M. Tanaka and O. Tatebe. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 65–72, May 2012.

[30] I.J. Taylor, E. Deelman, D.B. Gannon, and M. Shields. *Workflows for e-Science*. Springer, 2007.

[31] Top500. `https://www.top500.org`.

[32] K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman. Rethinking data management for big data scientific workflows. In *2013 IEEE International Conference on Big Data*, pages 27–35, Oct 2013.

[33] Jens-Sönke Vöckler, Gideon Juve, Ewa Deelman, Mats Rynge, and Bruce Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 15–24. ACM, 2011.

[34] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, U. V. Catalyurek, T. Kurc, P. Sadayappan, and J. H. Saltz. An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1158–1172, Aug 2009.

[35] S. Wang, K. Li, J. Mei, K. Li, and Y. Wang. A task scheduling algorithm based on replication for maximizing reliability on heterogeneous computing systems. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1562–1571, May 2014.

[36] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[37] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.

[38] Wrench. `http://wrench-project.org/`.

[39] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1352–1363, May 2012.