Homework for CR02

Deadline: October 17, 2016

The answer for each question should contain from a few lines up to a page. The total should have between 2 and 4 pages. Questions can be asked during class or at Anne.Benoit@ens-lyon.fr.

Course question: Prove that Greedy-offline is a $\left(\frac{4}{3} - \frac{1}{3p}\right)$-approximation algorithm for Indep(p).

The remaining questions focus on the article "*Interference Aware Scheduling*", by B. Kreaseck, L. Carter, H. Casanova, J. Ferrante and S. Nandy, enclosed.

Question 1: What is the problem mentioned in this article about the classical model for communications (as the one seen in class) ?

Question 2: Assume that a processor A has to send data to processors B and C. What is the methodology proposed by the authors to predict the computation performances during these transfers ?

Question 3: The "*bandwidth-centric principle*" is similar to some result(s) seen in class, which one(s) ?

Question 4: What is the meaning of $T(n)$? What is the objective pursued by the authors ?

Question 5: Motivate the choice of the parameters C^n, MR^n, C_r^n and $C_{sr}^n(i)$.

Question 6: In the solution proposed by the author in the *multi-port* setting, which processors do not participate to the computation ?

Question 7: Explain and discuss how the scheduling policies proposed in Section 3 are adapted as priority queues for the experiments of Section 4.

Interference-Aware Scheduling [*]

Barbara Kreaseck[3] Larry Carter[1] Henri Casanova [1,2]
Jeanne Ferrante[1] Sagnik Nandy[1]
[1] Department of Computer Science & Engineering [2] San Diego Supercomputer Center
University of California at San Diego
[3] La Sierra University
{kreaseck, carter, casanova, ferrante, snandy}@cs.ucsd.edu

Abstract

Overlapping communication with computation is a well-known technique to increase application performance. While it is commonly assumed that communication and computation can be overlapped at no cost, in reality they interfere with each other. In this paper we empirically evaluate the interference rate *of communication on computation via measurements on a single processor communicating on a heterogeneous collection of local and remote processors, in both Java and C. We then present a model of interference, which can be used for more effective application scheduling, as demonstrated by real-world experiments.*

1. Introduction

Overlapping communication with computation has long been recognized as a useful technique to increase the performance of parallel and distributed applications. Most published work assumes that this overlap can occur "for free" [5, 13, 4, 8, 16, 1, 6, 15, 3]. However, communication and computation at the same processor do contend for resources. In this paper we focus on the *interference* between these two activities, i.e., the decrease in computation rate of a host due to concurrent data transfers.

Consider a scenario where processor A is actively computing at the same time that it is sending to processor B. Due to possibly intermittent communication and bandwidth sharing with other network traffic, the transfer rate between processors A and B can vary. We can concurrently measure the computation rate and the transfer rate. Figure 1 shows a graph of these measurements for one of our configuration testbeds. The y-axis is the computation rate (normalized to the maximum achieved) for processor A. The x-axis is the send transfer rate in MB/sec. A data point (x, y) on the graph displays an instance of concurrent activity: when the send transfer rate was observed to be x MB/sec, the normalized computation rate was observed to be y. The graph of these points shows the observed interference: as the transfer rate increases, the computation rate decreases in a roughly linear fashion.

We define the *Interference Rate of Communication on Computation* (IR) as the negative slope of the linear least-squares fit of the data points in graphs such as the one shown in Figure 1. For Figure 1, the linear fit is $y \approx -0.037x + 0.96$, and thus IR $= 0.037$. While an IR value of 0 is typically assumed in the literature, our experiments show that in practice IR can be large.

This paper first presents an empirical study of the interference of communication on computation for a group of heterogeneous processors, both with Java and C. Our findings include:

- the computation rate can be reduced by over 50% due to concurrent communication,
- the reduction is largely independent of the number of communicating threads/processes,
- in general, the IR of receiving (IR$_r$) from a processor is larger than the IR of sending (IR$_s$) to the same processor,
- when sending and receiving concurrently, IR$_s$

[*]This work was supported by the National Science Foundation under Grant No. 0234233.

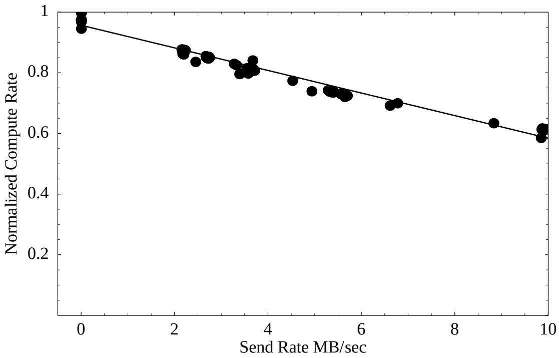


Figure 1. Observed Interference Rate of Communication on Computation when processor A computes and concurrently sends to processor B.

and IR_r are different from the IR's of separate communication, and there is a synergy in that the combined IR is smaller, and

- the value of IR depends upon the relative location, operating system, and architecture of the communicating processor. *We observed that, in most cases, more remote processors had higher IR's.*

The main objective of this paper is to investigate whether accounting for the interference rate of communication on computation can lead to more effective application scheduling on distributed platforms. To this end, we revisit our previous work on "bandwidth-centric allocation" [5], which focused on maximizing the steady-state throughput of an embarrassingly parallel application deployed on a tree overlay network using autonomous scheduling techniques [13]. We show that accounting for the interference of communication on computation leads to better schedules, both in theory and in practice.

In Section 2 we outline our experimental methodology and present our interference measurements. In Section 3 we describe an interference-aware scheduler, which we evaluate in the real world in Section 4. Section 5 discusses related work, and Section 6 concludes this paper.

2. Measuring the Interference of Communication on Computations

We implemented a synthetic distributed application that can be configured to run over an arbitrary collection of processors, and for which each processor can perform three activities (compute, send, or receive) in different threads concurrently. We used different configurations of this application to experiment with different scenarios in which communication may interfere with computation. We implemented this application first in Java. (Our results for C are reported in Section2.5). Most processors were running Java (j2sdk1.3.1 or later) with natural threads, and processors sent and received data through Java sockets.

We created a simple benchmark to stress both the compute rate and the transfer rate at a node. The atomic compute task repeatedly calculates k diagonals of the square of a matrix. Unless otherwise noted, the matrix was 1024 × 1024 integers (i.e., 4MB), and $k = 1$. We measure the time it takes to compute each atomic compute task, and record its inverse, the compute rate in tasks/sec. The send and receive activities involved repeatedly sending or receiving 1MB of random data. The transfer rate is recorded in MB/sec.

2.1. Testbed

The testbed used to run our application spans resources over a wide area. Table 1 describes the nodes in our testbed. Most nodes are Intel-based desktop workstations running some flavor of Linux. A few are Sun workstations running Solaris. The distance between nodes ranges from a few meters (in the same lab) to thousands of kilometers (across continents). As far as possible, we used quiescent systems and averaged results over at least 5 duplicate experiments.

2.2. Experimental Scenario

All experiments in this section measure the Interference Rate (IR) of communication on computation for one node in the testbed. For the sake of this description we assume that measurements are conducted from the perspective of machine Lab0. For a portion of each experiment (usually the beginning), Lab0 computes tasks in isolation. The rest of the experiment introduces activities involving other nodes, including communication with Lab0. By correlating the compute and transfer rates, we are able to determine the IR of communication (send and/or receive) on Lab0's computation. To obtain a range of data points, we introduce various loads into the system in the form of additional communications involving Lab0.

Label	Arch	OS	MHz	RAM MB	Location
Lab0	P4	Linux RH K2.4	1700	512	HPC Lab (SD) oh
Lab1	P2	Linux RH K2.4	300	512	HPC Lab (SD) tandem
Lab2	P4	Linux RH K2.4	1700	512	HPC Lab (SD) ct
Lab3	P4	Linux RH K2.4	2000	1024	HPC Lab (SD) pa
Lab4	U.SPARC	SunOS 5.8	440	256	HPC Lab (SD) kalmar
Lab5	U.SPARC	SunOS 5.8	333	128	HPC Lab (SD) picard
Lab6	P2	Linux RH K2.4	451	384	HPC Lab (SD) boltzmann
Campus0-8	dual P3	FreeBSD 4.6.2	800x2	1024	AW Cluster (SD) broach, et al.
Campus9	U.SPARC	SunOS 5.8	333	128	APE Lab (SD) ursus
SB0-4	P3 Xeon	Linux Deb. K2.4	2200	512	UCSB Mayhem (SB) ash, et al.
SB5	P4	Linux Deb. K2.4	1800	512	UCSB Mayhem (SB) charcoal
Tenn	P3 Cu.mine	Linux Deb. K2.4	700	320	SAU (TN) cpp
Brazil	P3 Cu.mine	Linux RH K2.4	865	640	UFCG (BR) lula
France	quad Xeon	Linux RH K2.4	2400x4	1024	ENS-Lyon (FR) graal

Table 1. Processor architecture, operating system, and physical location of nodes in our testbed. We measured computation speed on Lab0 while it communicated within our Lab and across Campus, California (SB), the U.S. (TN), the equator (BR), and the Atlantic (FR).

Next, we convert the raw observations into related pairs of rates. To eliminate anomalies caused by the granularity of our measurements, we average the compute rates and transfer rates over regular intervals. After normalizing the compute rate to the maximum achieved compute rate, we perform a linear least-squares fit and compute IR as the negative slope of this fit. Thus, the IR is the percentage decrease in the computation rate per every MB/sec of transfer rate. We also performed and report on experiments in which Lab0 receives data from other nodes, and in which Lab0 both sends and receives data.

2.3. Impact of receiving on computing

As seen in Section 1, there is a steady degradation in the computation rate as the data transfer rate increases. We conducted experiments with a local node in our lab, Lab0, computing and receiving from one other node in Table 1. A single experiment generates an average of 95 data points relating observed transfer rates to normalized computation rates. For each experiment, we performed a least-squares fit. Over all 117 experiments, a *linear* fit errs by at most 9.1%; the average max error is 2.2% and the standard deviation is around 0.75%. A *quadratic* least-squares fit gives only marginal improvement (max error of 7.4%, average max error of 1.8%, a standard deviation of 0.52%), and we use the simpler linear fit hereafter.

The left half of Table 2 shows the Interference Rates due to receiving (IR_r) when Lab0 computes and receives from one other node. The average IR_r for Lab0 is approximately 0.052. This means that on average when Lab0 is receiving at 10 MB/sec, it degrades to $1 - 10 \times 0.052 = 48\%$ of its maximum compute rate. Depending upon the processor type, operating system, network connection and distance to Lab0, the IR_r varies between 0.0452 and 0.0891. The dominant factor is distance. For the closest nodes (Lab$n$) the IR_r is roughly 0.0463, and the IR_r increases with distance to above 0.074. For the node in France, the receive transfer rates were so low that the data looked more like a cluster than a line; nevertheless, the computed IR_r was consistent with other locations.

We also experimented with Lab0 receiving data from several nodes simultaneously. We observed a maximum total receiving transfer rate of just over 11 MB/sec on Lab0. When receiving from a single quiescent near node, the observed data transfer rate achieves 85%-95% of that maximum. When receiving from multiple quiescent near nodes, the observed data transfer rate achieves the maximum. In general, once a node is receiving at its maximum transfer rate, the *impact on computation* is maximized, regardless of the number of receiving threads.

It turns out that it is possible to estimate the resulting compute rate at Lab0 using the sum of the individual IR_r's weighted by their respective transfer rates.

	Lab0 Receives from Node				Lab0 Sends to Node			
Node	avg IR_r	avg const	avg StDev	avg MaxE	avg IR_s	avg const	avg StDev	avg MaxE
Lab1	**0.0458**	0.9714	0.0125	0.0312	**0.0314**	0.9909	0.0085	0.0226
Lab2	**0.0467**	0.9770	0.0146	0.0396	**0.0299**	0.9740	0.0119	0.0247
Lab3	**0.0468**	0.9781	0.0136	0.0371	**0.0299**	0.9752	0.0124	0.0301
Lab4	**0.0452**	0.9674	0.0150	0.0371	**0.0199**	0.9922	0.0048	0.0155
Lab5	**0.0467**	0.9723	0.0212	0.0434	**0.0211**	0.9704	0.0249	0.0631
Lab6	**0.0465**	0.9754	0.0140	0.0376	**0.0305**	0.9738	0.0112	0.0276
Campus0	**0.0474**	0.9777	0.0053	0.0192	**0.0388**	1.0296	0.0184	0.0567
Campus1	**0.0477**	0.9823	0.0052	0.0163	**0.0373**	1.0223	0.0209	0.0547
Campus2	**0.0473**	0.9772	0.0048	0.0147	**0.0383**	1.0245	0.0186	0.0549
Campus3	**0.0471**	0.9757	0.0056	0.0255	**0.0381**	1.0219	0.0151	0.0509
Campus4	**0.0474**	0.9790	0.0054	0.0214	**0.0367**	1.0122	0.0185	0.0540
Campus5	**0.0475**	0.9777	0.0046	0.0155	**0.0377**	1.0218	0.0205	0.0554
Campus6	**0.0474**	0.9779	0.0050	0.0163	**0.0372**	1.0136	0.0157	0.0507
Campus7	**0.0472**	0.9765	0.0052	0.0167	**0.0377**	1.0177	0.0141	0.0410
Campus8	**0.0477**	0.9797	0.0048	0.0145	**0.0366**	1.0073	0.0148	0.0456
Campus9	**0.0494**	0.9737	0.0233	0.0905	**0.0237**	0.9745	0.0175	0.0404
SB0	**0.0510**	0.9844	0.0053	0.0171	**0.0310**	0.9771	0.0026	0.0098
SB1	**0.0515**	0.9840	0.0054	0.0150	**0.0312**	0.9781	0.0026	0.0124
SB2	**0.0513**	0.9844	0.0053	0.0153	**0.0313**	0.9762	0.0022	0.0071
SB3	**0.0511**	0.9841	0.0044	0.0144	**0.0310**	0.9758	0.0027	0.0127
SB4	**0.0511**	0.9758	0.0058	0.0147	**0.0307**	0.9762	0.0034	0.0158
SB5	**0.0501**	0.9762	0.0053	0.0163	**0.0307**	0.9784	0.0032	0.0110
Tenn	**0.0743**	0.9979	0.0019	0.0090	**0.0638**	0.9911	0.0016	0.0074
France	**0.0848**	0.9989	0.0024	0.0166	**0.0372**	0.9966	0.0015	0.0079
Brazil	**0.0891**	0.9992	0.0013	0.0053	**0.0431**	0.9955	0.0008	0.0037

Table 2. IR_r when Lab0 receives from various nodes. IR_s when Lab0 sends to various nodes.

We calculate the *expected* normalized compute rate to be $[1 - \sum_{i=1}^{k}(IR_r(i)\ TR(i))]$ when receiving from k nodes, where $IR_r(i)$ is the interference rate from node i, and $TR(i)$ is the transfer rate from node i.

When computing on Lab0 and concurrently receiving from four nodes with similar IR_r values within the Lab, the expected compute rate varies from the actual compute rate by 2.58% on average. For dissimilar IR_r values, (e.g., when computing on Lab0 and concurrently receiving from Lab1, Campus4, SB1, and Tenn), the expected compute rate varied from the actual compute rate by 1.21% on average. Over all our experiments with multiple receives on Lab0, using the aggregate interference rates to predict compute rates yields a *standard deviation* as low as 0.22%, and 0.82% on average. This demonstrates the utility of collecting the individual interference rates (IR_r's) between nodes. These rates can be combined arbitrarily to understand the aggregate impact of receiving multiple communications on computation.

2.4. Impact of sending on computing

When a node is computing and concurrently sending to other nodes, we also see a steady degradation in the compute rate per MB/sec of sending. We conducted experiments with Lab0 computing and concurrently sending to a receiving node chosen from those listed in Table 1. Over all 118 IR_s experiments we performed, a linear least-squares fit errs by at most 9.6% (average max error is 3.1%) with a standard deviation around 1.05%. A quadratic least-squares fit errs by at most 10.1% (average max error is 2.6%) with a standard deviation around 0.74%. Both fits are looser than those we saw with receiving, and again we use the linear fit for simplicity.

The right half of Table 2 shows IR_s values. The average IR_s for Lab0 is around 0.034: on average when Lab0 is sending at 10 MB/sec, its computation rate degrades to 66% of its maximum compute rate. Thus, *sending has less of an impact on computation than receiving does*. In our experiments, IR_s varies between

0.0199 and 0.0638, and generally increases with node distance, although not as much as IR_r.

The observed maximum transfer rate for sending is slightly less than that for receiving. When simultaneously sending to six nodes, Lab0 achieved a rate of around 10.7 MB/sec. As with receiving, once a node is sending at its maximum transfer rate, the impact on computation is constant, regardless of the number of sending threads. Again, we can estimate the resulting compute rate using the formula $[1 - \sum_{i=1}^{k}(IR_s(i)\ TR(i))]$, where $IR_s(i)$ is the interference rate of sending to node i, and $TR(i)$ is the transfer rate to node i. When computing on Lab0 and concurrently sending to four Linux nodes with similar IR_s values within the Lab, the calculated compute rate varies from the actual compute rate by 0.88% on average. For dissimilar IR_s values due to distance, (e.g., when computing on Lab0 and concurrently sending to Lab1, Campus4, SB1, and Tenn), the calculated compute rate varied from the actual compute rate by 2.04% on average. Over all our experiments with multiple sends on Lab0, using the aggregate interference rates to predict compute rates yields a standard deviation as low as 0.43%, and 1.21% on average.

2.5. Further experiments

We summarize results from other experiments (see [12, 14] for details).

Sending and receiving concurrently. One might hope that when a processor node is receiving at rate $TR(A)$ from node A and sending at $TR(B)$ to node B, the resulting normalized compute rate would be the weighted sum $1 - IR_r(A)\,TR(A) - IR_s(B)\,TR(B)$. Interestingly, we found that this was not the case. We observed a synergistic effect, where the actual compute rates were higher than this formula predicts.

In a typical experiment, measuring the receive and send interference rates separately gave the formula $1 - .0502\,TR(A) - .0327\,TR(B)$. We collected 234 measurements of the compute rate at a variety of receive and send rates. The error of the measured values from the predicted value was large, 6.7%, and the maximum error was 25%. If we made a planar fit to the 234 data points, the resulting formula was $.9542 - .0427\,TR(A) - .0265\,TR(B)$. This formula had an average error of only 2.2%, and the maximum error was 12.4% Note in particular that the more accurate predictor uses lower interference rates, showing that receiving and sending concurrently is more efficient than doing them separately.

We conducted 456 experiments with Lab0 computing while sending to one node and receiving from another. On average, the best planar fit to all the points had a maximum error of 19.1% and a mean error of 1.5% with a standard deviation of 1.2%. When using the IR_r and IR_s values from receiving or sending alone, then on average there was a maximum error of 37.3%, and a mean error of 5.3% with a standard deviation of 3.1%.

Another interesting observation is that when concurrently sending and receiving to nearby nodes without contention, the processor node is able to send at almost the maximum send transfer rate (90%-95%), but it is only able to receive at a 45% to 70% of the maximum receive transfer rate. When sending at a lower transfer rate (due to sending contention), the processor node is able to receive at a higher transfer rate.

Data footprint. As the amount of memory required by the computation decreases (by using matrix sizes from 1024×1024 down to 8×8) both IR_r and IR_s decrease to about half their original values.

Message size. As the message size decreased from 8 MB to 128 KB, there was no significant difference observed in interference rates.

Distance. Even though near nodes in general had lower interference rates than remote nodes, near nodes achieved much higher transfer rates, and thus had a much greater total interference on computation.

Software heterogeneity. Both the operating system (Linux versus Solaris) and the Java implementation (native versus green thread) impacted the interference rates. For example, Solaris had about a third lower send interference rate than Linux. We also found that Solaris received a smaller portion of data than a Linux node, when both were receiving concurrently.

Programming language. We duplicated a subset of our experiments using C processes instead of Java threads, in order to determine whether interference was simply an artifact of Java and threads. We used four of the nodes previously used (Lab0, Lab1, Campus0 and Campus2), and also an additional campus node and several remote PlanetLab nodes. The computation and application setup were almost identical to

	Lab0 Receives from Node				Lab0 Sends to Node			
Node	avg IR_r	avg const	avg StDev	avg MaxE	avg IR_s	avg const	avg StDev	avg MaxE
Lab1	**0.0421**	0.9792	0.0110	0.0258	**0.0228**	0.9792	0.0088	0.0208
Lab2	**0.0412**	0.9890	0.0077	0.0321	**0.0229**	0.9628	0.0195	0.0372
Campus0	**0.0389**	0.9528	0.0120	0.0472	**0.0272**	0.9471	0.0278	0.0611
Campus2	**0.0409**	0.9908	0.0089	0.0365	**0.0290**	0.9954	0.0217	0.0501
Campus10	**0.0421**	0.9952	0.0087	0.0409	**0.0299**	0.9963	0.0345	0.0749
chicago	**0.0499**	0.9378	0.0109	0.0622	**0.0330**	0.9375	0.0125	0.0600
new york	**0.0476**	0.9513	0.0086	0.0487	**0.0335**	0.9502	0.0107	0.0425
india	**0.1086**	0.9438	0.0107	0.0562	**0.0687**	0.9755	0.0115	0.0601

Table 3. C experiments: IR_r when Lab0 receives from and IR_s when Lab0 sends to various nodes.

the Java experiments, differing only in message size of 1KB instead of 1MB.

Our results confirm that, just as with Java, the interference on computation is linearly proportional to the send and receive bandwidths, that the interference rate of sending is less than that of receiving, and so on. Table 3 shows the results of the C experiments. The mean error of the measured data from a linear fit was 2.51%. It is also interesting, but not surprising, that on the four machines used in both experiments, the interference rates using C were marginally lower than the rates when using Java.

3. Interference-Aware Scheduling Model

Our goal in precisely evaluating the interference of communication on computation is to obtain a more realistic model that can be used as a basis for practical scheduling. Scheduling algorithms should account for interference when assigning computation and data to resources. In what follows, we show how this can be done for one specific scheduling scenario.

We model a collection of heterogeneous resources and the communication links between them as the nodes and edges of a rooted tree. Each node is a computing resource (a processor, or a cluster, or whatever) capable of computing and/or communicating with its neighbors at (possibly different) rates. Given n independent, identical tasks, we assume that their input data is initially located on or generated by the root of the tree. The root processor decides which tasks to execute itself, and which to forward to its children. Each child faces in turn the same decision: for each task, it decides whether to execute it or delegate it. Since we assume a heterogeneous system, nodes process tasks at different rates. To prevent overloading a node, a parent only sends work to a child when the child requests it. When the parent has requests from multiple children, the parent's scheduling policy determines which requests to fulfill and in what order.

In our previous work [5], we assigned priorities based on the communication times between a node and its children. We considered two types of models: one in which communication could proceed independently of computation (i.e., each child has an interference rate of 0), and one in which computation and communication were mutually exclusive activities (i.e., each child has an interference rate of $1/Z$, where Z is the task size). Given these assumptions, we showed that the scheduling problem is not NP-complete, but is solved by the *bandwidth-centric* principle: if enough bandwidth is available, then keep all children busy; if bandwidth is limited, then tasks should be allocated to the children that have sufficiently fast communication times,[1] and priority is given to the children with the fastest communication time. Perhaps counterintuitively, the scheduling priorities are not based on the processing speed of the children — however, the processing speed does affect how frequently the child requests work.

The experiments of the previous section show that a real system is different from the model used in our bandwidth-centric work in three important respects: (1) different children have different interference rates, and these rates are not necessarily 0 (fully overlappable communication) or $1/Z$ (mutually exclusive), (2) if a parent communicates with sufficiently many

[1] In the mutually exclusive model, a parent never sends tasks to a child if doing so would take more time than computing the task itself.

children concurrently, the total bandwidth available for delegating tasks is largely independent of *which* children it delegates them to, and (3) there is synergy that reduces the interference rates when sending and receiving concurrently.

In this section, we develop a simplified model of a computer's computation and computation power that is appropriate for the independent-task scheduling problem, and then modify the bandwidth-centric approach to conform to the new model.

3.1 A Simple Model for a Scheduler

Our experiments established that if node n is receiving at rate $TR(n)$ from its parent (node p) and sending to its k children at rates $TR(1), TR(2), \ldots, TR(k)$, then its normalized compute rate is approximated by the linear formula:

$1 - IR_r(p)\, TR(n) - \sum_{i=1}^{k}(IR_s(i)\, TR(i))$,

Unfortunately, as described in Section 2.5, getting a good fit required sampling the compute rate at many combinations of send and receive rates, which in turn required an extensive experimental setup. We would like a simpler methodology.

Modeling interference at the root node is simple, since the root only sends data to other nodes, and our experiments show that the interference due to multiple sends is additive. Any other node will receive at least as much data as it sends, and presumably more since it consumes some of the data locally.[2] Thus, the most likely activities are (1) it is not communicating at all, (2) it is receiving but not sending, or (3) it is both receiving and sending. It may not be important to correctly model the case that a node is sending but not receiving. In fact, if the scheduler has enough control, it should strive to make sends overlap with receives to take advantage of their synergy.

Thus we propose a simple model based on measuring the computation rates at the three types of points listed above and interpolate linearly between them. Making this explicit, for each non-root node n in the system, we first measure C^n, the number of tasks per unit time that can be executed by the node when it is not communicating. Next we measure the maximum receive bandwidth MR^n and corresponding compute rate C^n_r when the node is receiving from its parent but not sending. Finally, for each child i of n, we make a measurement of n's compute rate $C^n_{sr}(i)$ while it is sending to child i at bandwidth $SR(i)$ and receiving from its parent at bandwidth $RR(i)$. Now define:

$IR_r = (\frac{C^n - C^n_r}{C^n})/MR^n$

and

$IR_s(i) = (1 - \frac{RR(i)}{MR^n}\frac{C^n - C^n_r}{C^n} - \frac{C^n_{sr}(i)}{C^n})/SR(i)$

Our model for the normalized compute rate when receiving at bandwidth $TR(n)$ and sending to child i at rate $TR(i)$ is:

$1 - IR_r\, TR(n) - \sum_{i=1}^{k}(IR_s(i)\, TR(i))$,

This formula was chosen to exactly match the (k+2) measured values.

Going back to the example of Section 2.5, recall that the best linear fit to all 234 data points was the formula $.9542 - .0427\, TR(A) - .0265\, TR(B)$, which had an average error of 2.2%. If we just used the three data points as suggested above, we would derive the formula $.9440 - .0438\, TR(A) - .0235\, TR(B)$. Although this formula has a worse error rate (2.6%) when averaged over all data points, it does substantially better for the 86 points that lie within the convex hull of the three sampled points, which should correspond to the points that are relevant to our scheduling problem. The average error on the subset of points was reduced from 1.2% to 0.67%, and the maximum error went from 5.5% to 3.9%. Thus, by tailoring the model to the scheduling application, we both simplify the measurements needed and get a more accurate model.

3.2 Interference-aware scheduling

In this section, we develop a scheduling strategy that maximizes the throughput of independent tasks on a tree-overlay network, assuming the model given above. A summary of notation:

- Let Z be the size of each task, in megabytes.
- Let B^n_r be the maximum bandwidth, in tasks/second, that node n can receive from its parent. Thus, $B^n_r = MR^n/Z$.
- Given a schedule, we will denote by $S(n)$ the average or "steady state" number of tasks per second that are computed in node n.

[2] Since the receive bandwidth is reduced when a node is sending, it is even less likely the node will need to be sending but not receiving.

- We use the model given above: if node n is receiving at rate $TR(n)$ megabytes/second from its parent and sending at a rate of $TR(i)$ to child i, then $S(n)$ is at most $[1 - IR_r^n \ TR(n) - \sum_i (IR_s^n(i) \ TR(i))]C^n$ tasks per second, where the sum is over all children of node n.
- Let $T(n)$ be the total number of tasks per second executed in the subtree rooted at node n. Since these tasks must all be communicated from n's parent to n, $T(n) = TR(n)/Z$. Also, $T(n) = S(n) + \sum_i T(i)$. Note that we have the constraint $T(n) \leq B_r^n$.

We consider two different types of schedules. The first allows multi-port sends, that is a node can send tasks to multiple children concurrently. This is known to be advantageous to take best advantage of available network bandwidth. The second is single-port sends, where a node can send a task to at most one child at a time.

3.2.1 Multi-port sends

In the multi-port send case, our experimental data suggest there is a fixed limit B_s^n on the total number of tasks per second going out from node n, that is, $\sum_i T(i) \leq B_s^n$. In addition, if n is not the root, we have the constraint:

$$\begin{aligned} T(n) &= S(n) + \textstyle\sum_i T(i) \\ &\leq (1 - IR_r^n \, T(n) \, Z - \textstyle\sum_i (IR_s^n(i) \, T(i) \, Z))C^n \\ &\quad + \textstyle\sum_i T(i). \end{aligned}$$

Solving for $T(n)$, we obtain the constraint:

$$T(n) \leq (C^n + \textstyle\sum_i T(i) \, (1 - IR_s^n(i) \, Z \, C^n))/(1 + IR_r^n \, Z \, C^n).$$

For the root, there is no interference due to received tasks, so the corresponding constraint is simply:

$$T(n) \leq (C^n + \textstyle\sum_i T(i) \, (1 - IR_s^n(i) \, Z \, C^n)).$$

We first note that if we are given a feasible schedule that has node n's processor idle some of the time but sends some tasks to a child k of n, then reassigning work from k's to n's processor will be "even more" feasible — the interference at n's processor will be decreased, and reduced data traffic from n and k might even increase the available bandwidth into n. Thus, we can assume without loss of generality that either there are *no* tasks computed by n's children, or that there is no idle time on n's processor. In the first case, we have $T(n) = \min(B_r^n, \ C^n/(1 + IR_r^n \ Z \ C^n)$ (or simply $T(n) = C^n$ for the root.) In the latter case, the $\leq$ in the above constraint becomes an equality.

Next note that if $1 - IR_s^n(i) \, Z \, C^n$ is negative, $T(n)$ becomes smaller when $T(i)$ is increased. In this case, the maximum throughput entails setting $T(i) = 0$. This corresponds to the intuitive observation that one should never off-load work to a child if doing so would cost more processing time (due to increased interference) than would be needed to compute the task locally.

Now suppose for some i and j with $IR_s^n(i) < IR_s^n(j)$, it would be consistent with the children's bandwidth constraints to reassign a task from child j to child i. This would not change either $T(n)$ or $\sum_i T(i)$. Thus, it would not change the bandwidth constraint on the edge coming into node n from its parent.[3] However, the total interference on n's processor would be decreased, possibly allowing more tasks to be executed.

These observations lead to the following principle. *When a schedule uses multi-port sends, throughput at each node will be maximized by prioritizing the children by their interference rates* $IR_s^n(i)$ (lower interference rate children get higher priority), and never sending tasks to child i if $IR_s^n(i) \, Z \, C^n \geq 1$.

We can now determine the steady-state throughput for a tree overlay network that uses this principle. This is done using the same method as developed in bandwidth-centric scheduling [5]. Starting from the leaf nodes and working toward the root, we will compute an upper bound $\hat{T}(n)$ on the number of tasks per second that can be communicated to and computed in the subtree rooted at root n. Assume: the upper bound $\hat{T}(i)$ for each child i of n; the children of node n are sorted so that for $i < j$, $IR_s^n(i) \leq IR_s^n(j)$; and we ignore all children for which $IR_s^n(i) \ Z \ C^n \geq 1$. Then $T(n)$ is maximized by sending the first p children (for some p) as much as they can handle (i.e. setting $T(i) = \hat{T}(i)$ for $1 \leq i \leq p$), sending the $(p{+}1)^{th}$ child as much "leftover" work as possible, and sending the remaining children no work. Thus, we first maximize p then maximize $T(p + 1)$ subject to three constraints:

[3] We assume that even though the total amount of data that node n can receive from its parent might be a (monotone decreasing) function of the total outgoing data, it is not affected by *which* nodes the data is being sent to.

1. $\sum_{i=1}^{p+1} T(i) \leq B_s^n$,
2. $\hat{T}(n) = (C^n + \sum_i T(i)\,(1 - IR_s^n(i)\,Z\,C^n))/(1 + IR_r^n\,Z\,C^n)$ is within the bandwidth limit into node n when one is sending $\sum_i T(i)$ out from n. (If n is the root, there is no constraint; we simply set $\hat{T}(n) = (C^n + \sum_i T(i)\ (1 - IR_s^n(i)\ Z\ C^n))$.)
3. The number of tasks computed at node n is non-negative, i.e. $\hat{T}(n) - \sum_i T(i) \geq 0$.

The above formula for $\hat{T}(n)$ is an upper bound. As with bandwidth-centric scheduling, this bound can be achieved after a fixed-length startup period, provided that the scheduler can precisely control how much of the total bandwidth is allocated to each child and there is sufficient buffer space on all nodes to accommodate the transmission latencies.

3.2.2 Single-port sends

In the second case, a node can only communicate with one child at a time. Now, in place of the constraint that $\sum_i T(i) \leq B_s^n$, we have that $\sum_i T(i)/B_r^i \leq 1$

Let us assume that the receive bandwidth limit on incoming edge is a constant, independent of the current send speed.[4]

Suppose we are given a feasible schedule. Consider the effect of switching *one second* of communication to child j to one second of communication to child i. In this case, the amount of work offloaded from node n to its children will be increased by $B_r^i - B_r^j$ tasks. However, the number of tasks that can be computed locally at node n will be changed by $IR_s^n(j)B_r^j ZC^n - IR_s^n(i)B_r^i ZC^n$. Thus, the total number of tasks that can be computed at the subtree rooted at n will increase if and only if $B_r^i(1 - IR_s^n(i)\,Z C^n) > B_r^j(1 - IR_s^n(j)\ Z\ C^n)$.

Thus, *if a schedule sends to only one child at a time, throughput is maximized if it prioritizes the children by* $B_r^i(1 - IR_s^n(i)\ Z\ C^n)$ (larger values get higher priority). As before, one should never send to child i if $IR_s^n(i)\ Z\ C \geq 1$.

Once again, we can determine an upper bound on the steady-state throughput by giving the first p children as much work as they request and then giving the $(p+1)^{th}$ child as much as possible, subject to the constraint $\sum_i T(i)/B_r^i \leq 1$ along with (2.) and (3.) above.

4. Real-world Scheduling Experiments

4.1 Experimental Distributed Application System

Our experimental distributed application system contains a Controller process that sets up experiments on Node processes on the various distributed heterogeneous processors available. An experiment consists of a description of the application, a description of the tree overlay network upon which the application should be run, and a choice of scheduling algorithm. We use a synthetic *independent task application* where the results of computation are not collected.

Each algorithm considered specifies the *child node priority*: all child nodes may have equal priority, or information is used to prioritize the sending of tasks to its child nodes. The information might be the compute rate of the child, the transfer rate to the child, or the interference rate at the parent of sending to the child. Algorithms also differ in their ability to overlap send operations to a child, that is whether sending is *single-ported* or *multi-ported*.

We now describe six autonomous scheduling algorithms. The first two are interference-aware, and use the interference rate of sending (IR_s) to prioritize the children. Three other autonomous algorithms were implemented, and for comparison the non-distributed RootComputes.

- **IA:Single** (interference aware: single port). When a parent finishes sending a task to a child, it initiates sending a new task to the requesting child with the lowest IR_s value.[5]
- **IA:Multi**: a multi-ported algorithm that will send to multiple children concurrently. When the cumulative bandwidth used by the current tasks being sent is less than the sending bandwidth limit of the parent, the parent will initiate sending a new task to the requesting child with the lowest IR_s value that is not currently receiving a

[4] Linear relationships between send and receive bandwidths could be handled by linear programming techniques.

[5] This is not necessarily the optimal priorities as derived in section 3. Unfortunately, we conducted the experiments before deriving the optimal algorithms.

task. The advantage here is that when there is bandwidth available and there are requesting children, it is likely that the parent will be sending at its maximum send bandwidth. This communication differs from the others in being *interruptible*. We implement interruptible communication by breaking the send task into chunks of 512 KBytes, and having the sending thread check for new requests after each chunk (rather than after the entire task) is sent.

- **FCFS** (first-come, first-serve): a single-ported algorithm that prioritizes the requests from child nodes based upon the arrival time of the request. When the parent node finishes sending a task, it initiates sending a new task to the requesting child with the oldest outstanding request. This is the only strategy tested that is "fair" in the sense that all children will be given tasks (unless the parent can compute all tasks sent to it) .

- **CompRate**: a single-ported algorithm that prioritizes requests from child nodes based upon the computation rate of each child node. When the parent node finishes sending a task, it initiates sending a new task to the requesting child with the highest computation rate.

- **BWC** (bandwidth centric): a single-ported algorithm that prioritizes requests from child nodes based upon the maximum transfer rate from the parent node to the child node. When the parent node finishes sending a task, it starts sending a new task to the requesting child with the largest maximum transfer rate.

- **RootComputes**. The root node computes all the tasks. There is no communication to its children.

Note that all of the algorithms considered are single-ported, except for IA:Multi. With single-ported scheduling, the percentage of transfer time that is allocated to a child node can be manipulated by the number of tasks sent to that child node in relation to all other tasks sent from the parent. With multi-ported scheduling, using Java threads and concurrent TCP/IP connections, the exact portions of the root's total transfer rate consumed by each concurrent task is handled by the underlying hardware and software. Although one might be able to dictate the exact level of sharing of the transfer rate by packetizing and multiplexing by hand, we did not do so. Thus our multi-ported scheduling algorithm, IA:Multi, may be sub-optimal.

4.2 Synthetic Application and Example Testbed

Rather than use an existing application, we chose to design an application that would enable us to exercise a wide range of computation-to-communication ratios. Our synthetic application is based upon the computation and communication tasks that we used in Section 2. The computation task involves computing the diagonals of a matrix. In our synthetic application, we only send tasks from parents to children; the children do not return results.

An experiment is described by the following:

- the scheduling algorithm choice,
- the description of a task: send size in kilobytes, and compute size in number of diagonals.
- the size of the experiment in number of tasks (which was always 1000 tasks for the experiments reported here),
- the number of nodes in the tree overlay network,
- for each node: node name, computation rate, number of child nodes, and for each child node: computation rate, maximum transfer rate, and the interference of communication to the child node on computation at the parent.

From the nodes that we studied in Section 2 (see Table 1), we selected a subset to form the testbed for our distributed system. The selected nodes are described in Table 4. The computation rates are measured in diagonals/sec (on a 1024×1024 matrix) and the transfer rates are in MB/sec. Lab0 is the root node for all fork graphs in our experiments. But unlike our past experiments, not only will there be computation on the root node, but also at the leaf nodes (P_i). For each (root, leaf) pair, we measured the maximum computation rate (no sending or receiving) shown in Table 4, on the first line for the root node and on the second line for the leaf node. On the third and fourth lines we record the computation rate for the root node and leaf node respectively when the root node is concurrently sending a task to the leaf node. On the fifth and sixth line we record the transfer rates when sending from the root node to the leaf node with no computation

		Leaf Nodes (P_i)					
		Lab3	Lab4	Lab5	Lab6	SB0	Tenn
1	Compute Rate Root	9.057	9.055	9.055	8.98	9.18	9.06
2	Compute Rate P_i	23.86	3.47	3.02	8.27	22.55	13.23
3	Compute Rate Root (w/ sending)	5.8	7.02	7.1	5.8	6.75	8.92
4	Compute Rate P_i (w/ receiving)	12.45	1.77	1.2	7.12	15.15	12.7
5	Transfer Rate (no Compute)	11.1	11.12	11.05	10.85	7.7	0.3
6	Transfer Rate (both Compute)	10.8	10.81	9.62	10.7	7.73	0.2
7	$IR_s(i)$ at Root (Root Computes)	0.0299	0.0199	0.0211	0.0305	0.0310	0.0638
8	$IR_s(i)$) at Root (both Compute)	0.0333	0.0208	0.0226	0.0331	0.0342	0.0331
9	$IR_r(i)$) at P_i (both Compute)	0.0443	0.0454	0.0621	0.0130	0.0425	0.1300

Table 4. Various measurements on our example testbed fork graph where $Lab0$ is the Root. We record various computation rates (in diagonals/sec), transfer rates (in MB/sec) and interference rates (in percent degradation per MB/sec) from a variety of communication/computation scenarios on the fork graph.

and with both nodes computing respectively. The seventh line re-records the Interference Rate of sending on computation at the root node when only the root is computing (taken from Table 2). When the root sends a task to a leaf node and both nodes are computing, we measure the Interference Rate of sending to P_i on computation at the root ($IR_s(i)$) and the Interference Rate of receiving from the root node on computation at P_i ($IR_r(i)$) These measurements are shown in the eighth and ninth lines respectively.

4.3 Results

To compare the performance of the various autonomous scheduling algorithms, we designed and executed a number of experiments on the simple seven-node fork graph of Table 4. We varied the computation-to-communication ratio by using three task send sizes (2MB, 5MB, and 10MB), and six task compute sizes ranging from 1 to 11 diagonals, resulting in 18 application types. On Lab0, a 2MB, 11 diagonal experiment represents a computation-to-communication ratio of around 5:1, and a 10MB, 1 diagonal experiment represents a ratio of around 1:10.

The two Interference Aware scheduling algorithms use the Interference Rate of sending to a child node to prioritize the order in which tasks are sent to child nodes. Figure 2 shows our example testbed with the leaf nodes arranged in non-decreasing order of this rate. In contrast, Figure 3 shows our example testbed with computation rate priority, and Figure 4 shows maximum transfer rate order used by the Bandwidth-centric scheduling algorithm.

Figure 5 shows the simulation results, normalized to the time of RootComputes. In general, the IA:Multi scheduling algorithm outperforms all of the other algorithms, with one exception. In Figure 5(c) (task send size 10 MB), notice that for the smallest compute task size, 1 diagonal, the best scheduling algorithm is Root Computes. This was the only interference-aware experiment in which $IR_s(i)\ Z\ C^n > 1$. Hence, as discussed in section 3, the children should not have been sent any tasks.

We also observed that the overhead of sending a task varies between and across experiment runs. For instance, when sending a 2 MB send task from Lab0 to Lab3, the send time ranges from 200 ms to over 2000 ms with non-interruptible communication. This effect is most strongly seen in Figure 5(a) where the compute task size is 9 or 11 diagonals. These are the only applications where the BWC algorithms perform worse than CompRate. For these algorithms, the second-highest priority child was Lab3. During the CompRate runs, the send time to Lab3 was almost always around 200 ms. During the BWC runs, the send time to Lab3 was almost always around 2000 ms. This accounted for the apparent disparity in their execution times on these two applications as opposed to all other application types.

In summary, our multi-ported Interference Aware scheduling algorithm outperformed all others for all

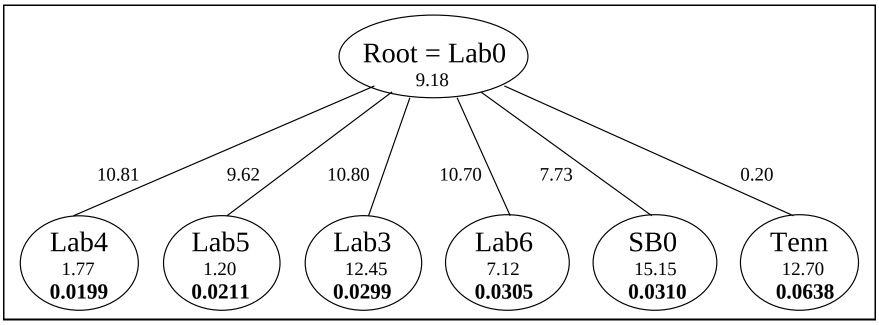


Figure 2. Testbed Fork Graph: Interference Aware Order. Within each node, we list the computation rate in diagonals/sec, and (for child i) its interference rate $I_s^{root}(i)$ on the root. Each edge shows the transfer rate in MB/sec. Here the leaf nodes are arranged in non-decreasing order of the interference rates. This is the priority of child nodes used by the Interference Aware scheduling algorithms.

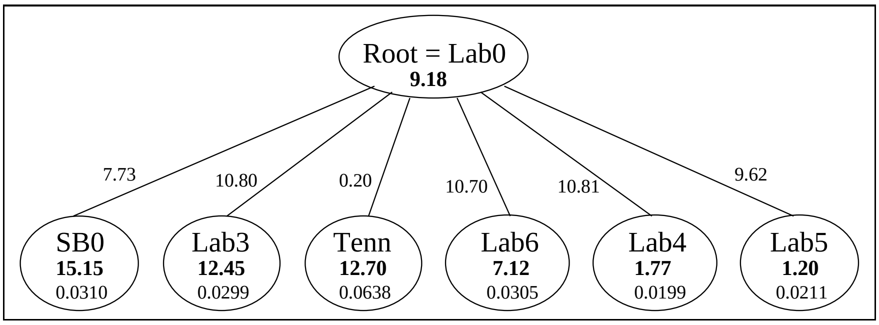


Figure 3. Testbed Fork Graph: Computation Rate Order. Here the leaf nodes are arranged in non-increasing order of computation rate, as used by the CompRate scheduling algorithm.

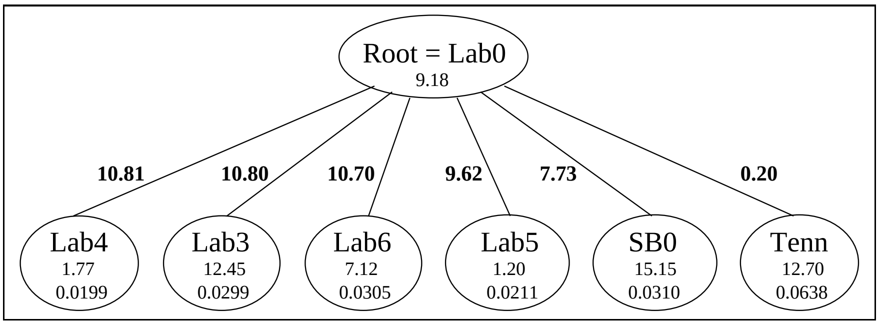
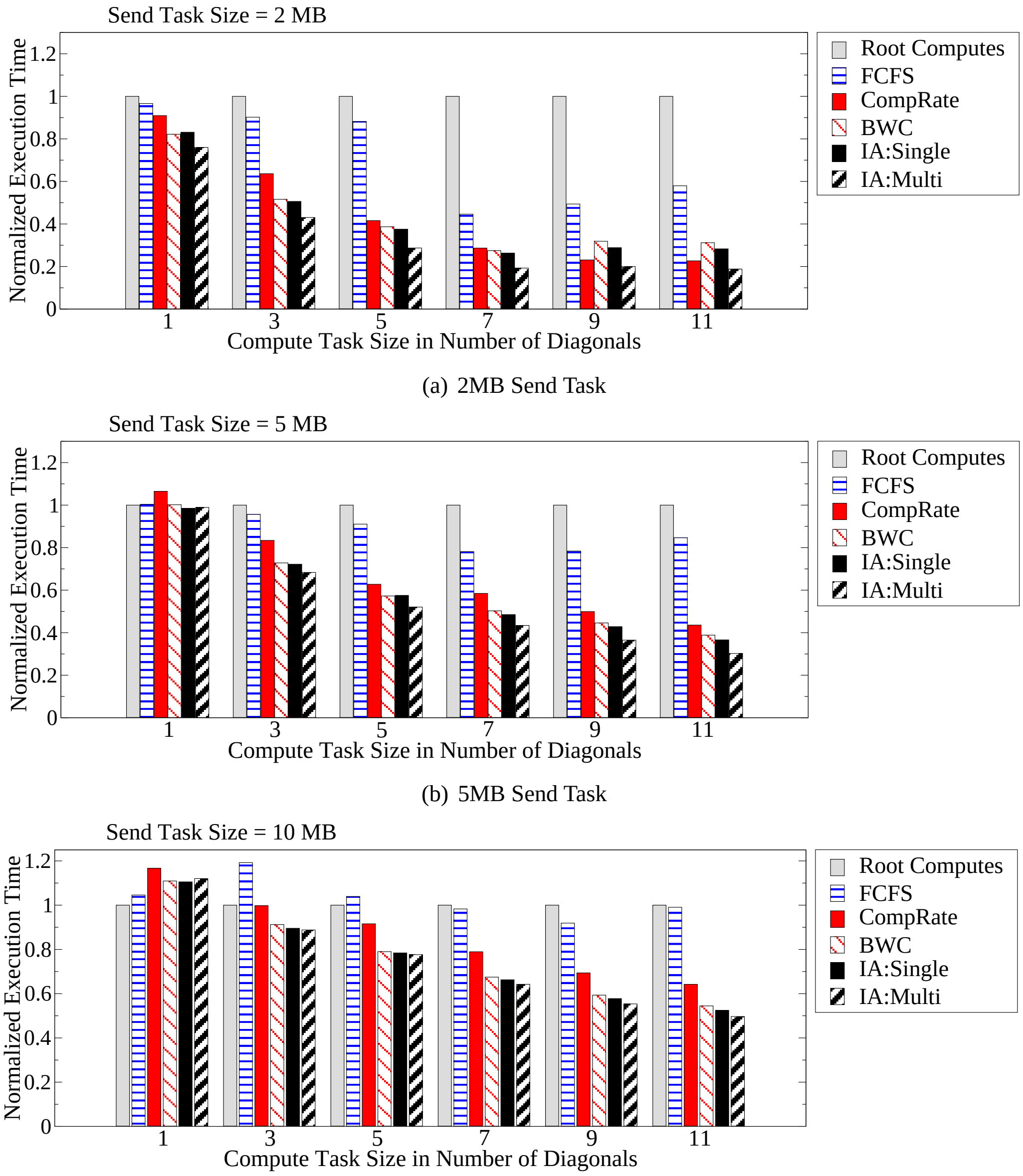


Figure 4. Example Testbed Fork Graph: Maximum Transfer Rate Order. Here the leaf nodes are arranged in non-increasing order of maximum transfer rate, as used by the BWC scheduling algorithm.



(a) 2MB Send Task

(b) 5MB Send Task

(c) 10MB Send Task

Figure 5. Comparison of scheduling algorithms. To provide a wide range of computation-to-communication ratios, we use task send sizes of (a) 2MB, (b) 5MB, and (c) 10MB. Additionally, compute task sizes varied from 1 to 11 diagonals. The normalized execution time is shown on the y-axis, where each bar represents an execution time normalized to the time of RootComputes.

application types. This result is also interesting because of the possible sub-optimality of our multi-ported scheduling heuristic, due to the lack of direct control over the subdivision of the sending bandwidth when sending to multiple child nodes. Among the single-ported algorithms, the IA:single was the best performer, especially when the computation-to-communication ratio is small. IA:single is followed closely by bandwidth-centric (BWC), which is not surprising since the two algorithms differ only in the priority assigned to Lab5. Both algorithms consistently outperformed the FCFS scheduling algorithm, which confirms the simulation results in [5].

5. Related Work

Most works in the area of application scheduling and mapping on distributed computing platforms make the simplifying assumption that communication and computation may overlap without cost (see for instance [1, 3, 4, 5, 6, 13, 15, 16, 9, 10, 18, 17]). While this assumption may seem reasonable, for instance due to DMA capabilities, this work provides empirical evidence that in fact it may not hold in practice. In our experiments conducted on a computing platform that aggregates individual hosts over various networks, both in Java and C, we have observed a significant interference of communication on computation. This work provides a model for this interference, and demonstrates how this model can be used when designing more practical and effective scheduling strategies.

One possible model is to just assume that some portion of each communication is never overlappable with computation, as in [11] for instance. Our results provide a more evolved model of interference that accounts for the specific data transfer and computation rates. Another possibility is to use more detailed communication models. For instance, low-level models have been proposed that include notions of overhead of communication on computation, such as the LogP [7] model and the family of models derived from it. LogP abstracts the communication of fixed-size short messages through the use of a number of parameters. A follow-on model, LogGP [2] introduces another parameter (G) to capture the behavior of long messages. By contrast our work focuses on a higher-level, more abstract model that can be easily instantiated with simple experiments, as demonstrated in Section 2, and was specifically developed to be used when reasoning about application scheduling.

6. Conclusion

We presented an empirical study of the interference of communication on computation for both multi-threaded Java and C. We define the *interference rate of communication on computation* (IR) to be the negative slope of the linear least-squares fit representing the relationship between the rate of computation on a processor and the rate(s) of communication between processors. We deployed an experimental measurement system on a select group of both near and distant heterogeneous processors. Within the framework of mostly Intel processors and Java with natural threads, we found that the computation rate is reduced by over 50% when communication reaches its maximum transfer rates. This reduction is roughly linear with the amount of data transferred per second, and is independent of the number of communicating threads. Furthermore, the IR of receiving (IR_r) from a processor is generally larger than the IR of sending (IR_s) to the same processor. Among other results, an intriguing one was that there is a synergy between sending and receiving. Our results with C confirm our results in Java (with marginally lower interference rates).

We further proposed a simple model based on measuring a small number of computation rates at a node: when a node was not communicating; receiving but not sending; and both sending and receiving at each child. The simple model is a linear interpolation of these points, and arises from our observation of greater accuracy in these small set of measurements vs. a more extended set. We then developed an interference-aware scheduling strategy that extends bandwidth centric scheduling. In the case that there are multi-port sends, we show that the throughput at each node is maximized by prioritizing the children of the node by interference rates, with lowest interference given the highest priority. In the single-port send case we determined a priority order based on the interference rate, task size, maximum bandwidth of receiving, and the compute-only rate. For both cases, we also determined an upper bound on the steady-state throughput.

We also performed real-world experiments of a collection of autonomous scheduling algorithms (including bandwidth-centric and interference-aware) on simple fork trees. In almost all cases, interference-aware scheduling outperforms the others (the exception being very small task size, when it was more beneficial to simply compute all tasks at the root, rather than communicate any tasks). Our results suggest that interference should be considered when developing algorithms that attempt to improve performance by overlapping computation and communication.

References

[1] T. Abdelrahman and G. Liu. Overlap of computation and communications on shared-memory networks-of-workstations. *Journal of Parallel and Distributed Computing Practices*, 2(2):145–153, June 1999.

[2] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[3] S. B. Baden and S. J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proceedings of SC'98, Orlando Florida*, November 1998.

[4] F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Optimizing metacomputing with communication-computation overlap. *Lecture Notes in Computer Science*, 2127:190–204, 2001.

[5] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric Allocation of Independent Task on Heterogeneous Platforms. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[6] V. Bouchitte, P. Boulet, A. Darte, and Y. Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap. *International Journal of Supercomputer Applications and High Performance Computing*, 9(3):205–219, 1995.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[8] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Proceedings of Euro-Par, Vol. I*, pages 165–172, 1996.

[9] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing completion time for loop tiling with computation and communication overlapping. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*, page 39, Los Alamitos, CA, Apr. 23–27 2001. IEEE Computer Society.

[10] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A technique for overlapping computation and communication for block recursive algorithms. *Concurrency: Practice and Experience*, 10(2):73–90, Feb. 1998.

[11] K. Högstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):307–321, March 2003.

[12] B. Kreaseck. *Dynamic Autonomous Scheduling on Heterogeneous Systems*. PhD thesis, University of California at San Diego, 2003.

[13] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[14] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. On the Interference of Communication on Computation in Java. In *Proceedings of the Third International Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS)*, 2004.

[15] A. Lain and P. Banerjee. Techniques to overlap computation and communication in irregular iterative applications. In *Proceedings of the International Conference on Supercomputing, Mancester, England*, pages 236–245, 1994.

[16] M. J. Quinn and P. J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(2):197–204, 1996.

[17] A. Sohn and R. Biswas. Communication studies of dmp and smp machines. Technical Report NAS-97-005, NASA Ames Research Center, March 1997.

[18] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Identifying the capability of overlapping computation with communication. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 133–138, Boston, Massachusetts, Oct. 20–23, 1996. IEEE Computer Society Press.