Communications-Efficient Multithreading on Wide-Area Networks

Michael S. Bernstein and Bradley C. Kuszmaul
Yale University Department of Computer Science
bradley@cs.yale.edu

Abstract

This paper shows how to run multithreaded programs on a DRAM (Distributed Random Access Memory) parallel computer and demonstrates that such programs can run efficiently on a collection of machines distributed across thousands of miles over the internet. Suppose we have a fully strict multithreaded program has work T_1 and critical-path length T_∞, and we have a P processor DRAM machine with λ an upper bound to the cost of routing any permutation. This paper presents a deterministic conservative DRAM scheduling algorithm that runs in time $O((\lambda + \log P)(T_1/P + T_\infty))$ and a randomized conservative DRAM scheduling algorithm that runs in time $T_1/P + O((\lambda + \log P)T_\infty)$. We have modified the Cilk multithreaded runtime system to use our randomized conservative DRAM scheduler. Surprisingly the modified system, called TreeCilk, often achieves a performance improvement when one 2000-mile-away machine is added to a tightly-bound cluster of machines.

1 Introduction

Cilk is a multithreaded language for parallel programming that generalizes the semantics of C by introducing linguistic constructs for parallel control. Versions of Cilk previously released by MIT use a provably efficient, randomized, work-stealing scheduler suitable for machines with plenty of bandwidth, such as supercomputers and contemporary symetric multiprocessors (SMP's) [BJK$^+$96, FRL]. It consumes subtantial bandwidth, since when running that algorithm, virtually every processor in the machine can send a message to a randomly chosen other processor frequently. This paper shows how to run a multithreaded scheduler on networks with limited bandwidth.

To understand why we need a new scheduling algorithm for limited bandwidth machines let us first examine the algorithm used by Cilk on a high-bandwidth machine, and then consider what happens on machines with limited bandwidth. High-bandwidth Cilk uses a *randomized work-stealing* strategy, in which an idle processor sends a message to randomly chosen processor to try to find work to steal. Each processor has its own pool of threads to work on. If a processor's pool becomes empty, it tries to find a thread in another processor that it can steal. It searches for such a thread by choosing another processor at random. The first processor is known as a thief, and the second processor is known as a victim. The thief sends the victim a message. If the victim has any extra threads in its thread pool, it replies by sending one of the threads back to the thief, which starts working on the thread. Otherwise, if the victim has no extra threads, then it sends back a message indicating that the attempt to steal has failed, at which point the thief picks another victim at random and tries again. This algorithm is both provably good and demonstrates good performance empirically [BJK$^+$96].

Consider what happens if we use randomized work stealing on the internet, a network with very limited bandwidth compared to a supercomputer. Let us suppose that we are using a million processors spread across the internet. A problem occurs when there are only a few processors which have work to do. This actually occurs regularly in some Cilk programs [1] In fact, those few working processors need to communicate with each other in order to get past a nearly serial part of the search so that more parallelism can be found again. If one of the processors happens to be on the west coast and one happens to be on the east coast, and there are a million idle processors, then the internet backbone would be flooded with steal/fail message pairs crossing the Mississippi river. The program has trouble getting its work done, because the two active processors cannot communicate with each other efficiently. Furthermore the internet's performance for other users might collapse under the resulting traffic. The internet authorities might be within their rights to kick us off.[2]

To first order, we will accept the network being completely clogged up if our program is getting some useful work done. Given the grab-all-the-bandwidth-you-can nature of the current internet, we will not try to solve the problem faced by the internet authorities, except to try to make sure we actually use the bandwidth to do something useful. Here we focus on using bandwidth productively, rather on the problems of governing the internet.

In this paper we show how to achieve algorithmic work-scheduling while using the network as efficiently as possible. The internet is quite difficult to understand, so we attack this problem for dynamic random access memory (DRAM) machines [LM88]. In a DRAM, communications performance can be predicted by cutting the network and measuring the cut. One measures the bandwidth provided by the network across the cut, and the amount of information that must be sent across that cut. The ratio of information to bandwidth is the load of the cut. Each such load is a lower-bound to the amount of time it will take to run the program. The key to a DRAM is that the worst-case cut (the one with the highest load) actually predicts the performance of the communications operation.

[1]For example, the $\star$Socrates massively parallel chess program gets into this situation [JK97], and the Cilk `knary` multithreading benchmark both contain into "mostly serial" portions of the code.

[2]For a discussion of techniques to avoid swamping high-bandwidth networks with work-stealing requests, see [Kus94].

In the context of a DRAM machine, our goal is to design a *conservative* DRAM work-scheduling algorithm. A conservative DRAM algorithm is one which induces a load on the network not much greater than is absolutely necessary. It is clear what is meant by the "absolutely necessary load" for the for the conservative DRAM algorithms presented in [LM88]. But what do we mean by "the absolutely necessary" load for work stealing? The thieves and victims are changing constantly. At one instant in time, the only available thieves may be far away from the only available victims. Perhaps if we wait a little while, there will be some thieves and victims nearer each other so we won't have to use as much bandwidth for them to communicate. We define this problem away by considering a snapshot at some moment in time. In this situation the processors are either thieves, or they are busy, and if they are busy they may have work available to steal. Now we wish to match up the thieves and the victims in such way to allow us to prove a good performance bound and without using too much bandwidth.

The underlying goal of Cilk's randomized work-stealing is to match up victim and thief processors. This kind of matching problem is known as the *rendezvous* problem.[3] In the rendezvous problem there are two sets of processors victims, and thieves, The goal is to find a maximal bipartite degree-one graph beteen the victims and the thieves. That is, we wish to give every thief the name of at most one victim so that no victim is named by more than one thief. Also, we want either all the thieves to receive a name, or all the victims to be named, (or both if there are the same number of thieves and victims.) A rendezvous algorithm can be used to implement a work-stealing scheduler, as follows. Run the rendezvous algorithm, and then every thief that received a victim's name steals from that victim.

Our strategy is to superimpose a tree data structure onto the network, so that we can perform operations such as reduction and prefix summation. Internal nodes of the tree are implemented by processors at the leaves of the tree. We place the internal nodes of the tree to try to maximize the ability to communicate with processors below it in the tree. A left-to-right numbering scheme identifies the processors. (See Figure 1 for an example tree.)

In this paper we describe and analyze four rendezvous algorithms. The first, Algorithm *DPR*, is a deterministic algorithm that uses parallel prefix, and is based on Christman's rendezvous algorithm [Chr83]. The second, Algorithm *RPR* is a randomized rendezvous algorithm that achieves tighter theoretical bounds. The third and fourth algorithms are conservative DRAM algorithms. The third, Algorithm *DDR* is a deterministic DRAM rendezvous. The fourth, Algorithm *RDR* is a randomized DRAM rendezvous. Our bounds for the conservative DRAM algorithms are the same as the parallel prefix rendezvous algorithms, but we argue both theoretically and empirically that the conservative DRAM have significantly better preformance.

We have implemented Algorithm *DDR* for the Cilk multithreaded programming system. (As of writing this paper we have not implemented Algorithm *RDR*, but we hope to have it running in time for the final paper. The required changes to our Cilk implementation appear to be straightforward.) Figure 1 shows an example of 25 processors spread across the USA, and along each edge of the tree, it shows the bandwidth used (in bytes per second.) The accounting of bandwidth works as follows. When a message is sent directly from one processor to another, we record that bandwidth is consumed along every edge of the tree. We record it this way, even though, for example, messages from Cleveland to Cambridge take a shortcut and never actually enter the Moffett Field cluster (where the root of the tree is situated.) This method of accounting shows more bandwidth being used along some links than are actually necessary. Leiserson and Maggs showed [LM88] that such a shortcut does not change a conservative DRAM algorithm into a nonconservative algorithm, however.

The remainder of this paper is organized as follows: Section 2 explains and analyzes the parallel prefix rendezvous algorithms (Algorithms *DPR* and *RPR*.) Section 3 presents and analyzes our conservative DRAM rendezvous algorithms (Algorithms *DDR* and *RDR*.) Section 4 provides an empirical performance

[3]The earliest version of the rendezvous algorithm that we know of was described by Christman [Chr83] in the context of parallel memory allocation.

analysis of the TreeCilk system using Algorithm *DDR*. Section 5 concludes with a discussion of the unsolved problems of wide-area Cilk.

2 Parallel-Prefix Rendezvous

This section shows how to implement a rendezvous without concern for network load induced by the algorithm. The algorithms presented in this section do not try to produce a local matching, and they are use substantial bandwidth while computing the matching. We shall describe and analyze two algorithms: Algorithm *DPR* (which stands for "Deterministic Prefix Rendezvous") and Algorithm *RPR* (which stands for "Randomized Prefix Rendezvous".) In the next section we shall describe algorithms that are more parsimonious with the network bandwidth.

The prefix rendezvous algorithms presented here use a parallel prefix enumeration primitive. The enumeration problem is as follows: Given a machine in which a subset S of the processors have a certain property, give each such processor a unique number between 0 (inclusive) and $|S|$ (exclusive.) Enumeration can be implemented in logarithmic time using a parallel-prefix operation, even on machines with relatively little bandwidth. For more details on how to implement parallel prefix and how it can be used, see [CLR90, Ble90].
Algorithm *DPR* (Deterministic Prefix Rendezvous):[4]

1. Enumerate the thieves.

2. Enumerate the victims.

3. The ith victim sends a message to Processor i, and the ith thief sends a message to Processor i (both messages contain the identity of their sender). Processor i is called the rendezvous processor for Victim i and Thief i.

4. The rendezvous processor then sends a message to the thief that contains the victim's identity.

5. When a thief recieves a message from the rendezvous processor, it knows the identity of a victim from which it should steal.

Algorithm *DPR*'s performance at multithreaded computations can be analyzed as follows. See [BL94] for a complete technical definition of the term *fully strict multithreaded computation*, and the performance measures *critical-path length* and *work*. Informally, a fully strict multithreaded computation is well-formed. The work is the time it takes one processor to execute the computation. The critical-path length is the time it would take an infinite-processor machine to execute the program with no scheduling or communications consts.

Definition 1 *The value λ is the worst-case cost of routing a permuation on the parallel computer being used.*

Theorem 2 *Consider the execution of any fully strict multithreaded computation with critical-path length T_∞ and work T_1 using the (nonrandomized) prefix rendezvous. The running time on P processors, including scheduling overhead, is $T_P = O((\lambda + \log P)(T_1/P + T_\infty)$.*

Proof: Every successful steal can be amortized against either work or the critical-path length in the manner of Brent's theorem [Bre74]. The overhead of a steal is $O(\lambda + \log P)$ since the enumerations cost time $\log P$

[4]Algorithm *DPR* is due to Christman [Chr83].

and the data that is sent for the rendezvous consists of three subpermutations, each of which costs no more than λ time. ∎

Algorithm *DPR*can be improved by modifying it so that every victim has an equal chance of being stolen. (If there are more thieves than victims, then there is no trouble, but if there are more victims than thieves, we want to make sure they all get a chance.)

Algorithm *RPR* (Randomized Prefix Rendezvous):

1. Choose a random number R and broadcast it to all the processors.
2. Enumerate the thieves, and also broadcast the total count of the thieves, T.
3. Enumerate the victims, and also broadcast the total count of the victims V.
4. Renumber the victims by adding their R to enumeration index (modulo V.)
5. The ith victim (using the new enumeration index) sends a message to Processor i, but only if $i < T$. The ith thief sends a message to Processor i. (both messages contain the identity of their sender).
6. The rendezvous processor then sends a message to the thief that contains the victim's identity.
7. When a thief recieves a message from the rendezvous processor, it knows the identity of a victim from which it should steal.

Thus, each victim has an equal chance of being matched with a thief. In this algorithm the probabilities of adjacent victims being stolen are not independent, but that independence is not needed for the Cilk results to go through if there is enough bandwidth.

Theorem 3 *Consider the execution of any fully strict multithreaded computation with critical-path length T_∞ and work T_1 using the randomized prefix rendezvous. The expected running time on P processors, including scheduling overhead, is $T_P = T_1/P + O((\lambda + \log P)T_\infty)$. Moreover, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time is $T_P = T_1/P + O((\lambda + \log P)(T_\infty + \lg(1/\epsilon)))$.*

Proof: The proofs of Blumofe and Leiserson [Blu95, BL94] work with some simplifications and minor modifications. In those proofs, an accounting scheme is used where processor time is accounted for by putting a dollar in one of three buckets on every time unit. The first bucket, the *work* bucket gets a dollar whenever a processor is actually working. The second bucket, the *steal* bucket gets a dollar every time a processor initiates a steal. The third bucket, the *waiting* bucket, gets a dollar for every time unit that a processor is waiting for a response to a steal request. On every time unit P dollars are distributed among the 3 buckets, since every processor is either working, stealing, or waiting. As with Blumofe and Leiserson's proof, the work bucket ends up with exactly T_1 dollars. The steal bucket will have an expected $O(PT_\infty)$ dollars in it, since on every global rendezvous, we have a good chance of stealing the critical threads. The wait bucket has an expected $O(PT_\infty(\log P + \lambda))$ since each rendezvous takes time $O(\log P + \lambda)$: It takes time $O(\log P)$ to perform the enumerates, and it takes time $O(\lambda)$ to send the messages, since they are all subpermutations. ∎

Thus the randomized prefix-rendezvous algorithm has large costs associated with critical-path length, but the costs associated with the work are small, which is an improvement over the deterministic prefix-rendezvous algorithm.

There are several problems with the prefix-rendezvous algorithms:

- These algorithms do not even try to produce a matching that preserves locality. It may be possible to find a matching that incurs no stealing across the root of tree, but these algorithms make no effort to find such a matching.
- Even if the matching happens to have a high degree of locality, the prefix rendezvous algorithms themselves use a great deal of bandwidth to find the matching. In particular, sending data to and from the rendezvous processors may incur much higher loads than the matching itself incurs. (This can happen, for example, if all the victims and the thieves are in the rightmost part of the tree, but the rendezvous processors are always at the leftmost part of the tree.)

On bandwidth-limited networks, Algorithm *RPR* has a big advantage over randomized work stealing, in that the slowdowns are limited by the cost, λ, of routing a worst-case permutation. Beyond the small load induced by computing the enumerations, every message sent is associated with actually getting some work stolen. In the randomized work-stealing algorithm, messages can saturate the network with no useful work getting done. For networks with plenty of bandwidth, however, randomized work-stealing is much faster than the randomized prefix-rendezvous algorithm.

3 Conservative Rendezvous

We have now explained and analyzed two prefix-rendezvous schedulers that do not take care to minimize network load. This section shows a conservative DRAM algorithm for rendezvous, which gives qualitatively better performance for the rendezvous. We have not been able to relate this improved rendezvous performance to an improvement on our bounds for work-scheduling, since as far as we know, in the worst case, it may actually require time λ to match thieves with victims. Empirically, we have found that the bandwidth used by our conservative rendezvous algorithm is substantially less than that used by the standard rendezvous algorithms, however. First we will more carefully define the goals we are trying to achieve, and then we will present and analyze two conservative DRAM algorithms, one deterministic and one randomized, which have the same bounds respectively as the deterministic and randomized prefix-rendezvous algorithms.

As stated earlier, we superimpose a tree onto the network. We designate the internal vertices as *switches* and the leaves are designated *processors*. When they do not need to be distinguished, switches and processors are called *nodes*. A switch may have an arbitrary number of children. All computational work is done by processors, and the scheduling work is distributed among the switches. Processors in the tree are in a one-to-one correspondence with the physical processors involved in the DRAM. Switches are virtual objects whose function is fulfilled by one of the physical processors in the subtree rooted in the switch. The computational requirements of scheduling are small compared to the requirements of the actual computation, so the selection of the host processor is somewhat arbitrary.

The notion of *ownership* is the foundation of Algorithm *DDR*. Each thief and each victim is logically *owned* by a node. When a processor becomes a thief or a victim, it creates a logical ownership, which it can transfer to its parent. Switches transfer ownership of thieves and victims upwards until a switch owns both a set of thieves and a set of victims; this switch is called the *matching switch*. The matching switch arranges a *match* between the thieves and victims, which signifies that each member of the thief set will steal from one of the members of the victim set. The matching switch does not actually make the one-to-one mapping; once a match is made, the switches in the subtree beneath the matching switch are responsible for constructing mappings, splitting the block of thief and victim ownerships into smaller blocks and distributing those blocks to their children. Match messages are sent only down subtrees which include thieves; since thieves steal from victims, the match messages are ultimately destined for thief processors. When a thief processor receives a single match pair containing its own ownership and the ownership of a victim, it steals from that victim and then destroys the pair of ownerships, since the thief is no longer a thief and the victim is no longer a victim. If the thief becomes idle once again, or the victim has more stealable work, they re-create ownership rights and the process begins anew.

Algorithm *DDR* maintains locality by performing matches in subtrees before performing matches between subtree. A switch S performs all possible matches between thieves and victims in the subtree whose root is S, and then sends the residue up to its parent. This process assures that matches between relatively local processors are made before matches between relatively remote processors, assuming that the tree is designed so that subtrees correctly reflect locality. The idea of organizing processors into a tree is not new; Keith Randall's distributed Cilk effectively uses a tree of height one with one switch and P leaves where P is the number of processors. The difference is that Algorithm *DR* does the matching conservatively, assuring that all intra-subtree matches are made before inter-subtree matches are attempted.

To help explain Algorithm *DDR*, we have illustrated several stages. The initial state of a hypothetical tree is illustrated in Figure 2. Switches are represented by rectangles and named with Greek letters. Processors are represented by rounded rectangles; thief processors are denoted Tx, while victim processors are denoted Vx. The notation $M = (t, v)$ signifies that the node owns t thieves and v victims. (The letter M stands for "Mine.") Thus, the thief processors own one thief each and the victim processors own one victim each.

3.1 Indirect Ownership Passing

The goal of making work scheduling a conservative algorithm requires that the scheduling algorithm pass no more information than necessary between nodes, particularly if that information is of variable size. This rules out an implementation which passes the ownership rights of thieves and victims by name, because the size of the messages containing those rights would grow linearly in the number of ownerships being passed. In the worst case, the size of a message could be proportional to the total number of thieves or the total number of victims in the tree. This is not a problem for small trees, and a scheduler using a name-passing scheme might even be better for small trees, but it has bad asymptotic behavior.

To make the passing of ownership a constant-size message, we pass ownerships anonymously. When a node owns T thieves, that ownership consists only in the node's thief ownership count being set to T. To pass ownership, a node creates a message which signifies the transfer of t thieves, sends the message, and subtracts t from its thief ownership count. When a node receives a transfer message, it adds the transfer count to its ownership count. The same process is used to transfer victims. The transfer message is therefore a constant-size message. This covers the messages which travel up from the processors to the root, carrying ownerships until a match is made. Figure 3 illustrates the tree after all ownerships have been transferred to the root switch α.

Match messages, which travel downward from the matching switch to the thief processors, are similar to transfer messages except that they include a *match pointer*. Match messages are ultimately destined for thief processors, and those thieves cannot steal without the address of a victim processor. Nevertheless, we established that the matching message must not include a full list of victim processors, since this could conceivably include all the victim processors in the tree. To solve this dilemma, we introduce the concept of a *match pointer*, which is either the name of a victim processor or a pointer to a switch which can provide a set of match pointers. When a switch or a processor requires actual victim names, it "dereferences" the match pointer by sending a *match query* to the referenced node, which responds with either a set of victim processor names or a set of match pointers. This allows us to pass names only when required, which keeps the algorithm conservative. Note that a match message always transfers the same number of thieves and victims; this is logical, since it would be impossible to match unequal numbers of thieves and victims.

3.2 Reservations

We introduce the concept of a *reservation* to make match queries work properly. There is a problem with the system described above: when a switch is asked to deference a match pointer, it is not clear how the switch can avoid returning the same victim processors to multiple queriers. To handle this problem, we use a system of *reservations*. When a node transfers ownership of victims to its parent, it keeps track of how many ownerships were passed as a reservation. The node no longer owns those processors, but it still must reserve them until the owner claims them. The state of the tree with reservation counts is shown in Figure 4. The notation "R=(t,v)" indicates that the node is reserving t thieves and v victims. Note that the sum of the "mine" and the "reserved" pairs for a given node is the count of thieves and victims which the node has been informed of.

Thief reservations are redeemed when a parent sends a match message to its child. Since it is thief processors which ultimately steal, the match messages for a given set of matches cascade down the thief subtree to the thief processors named in the set. Match messages transfer ownership of both thieves and victims; the owned victims are in an adjacent subtree, but since a node N originally transferred ownership of t thieves to its parent, it must eventually receive a match message for t thieves, redeeming its own reservation. Upon receipt of the match message, the ownership rights of t thieves are transferred from the reserved pool to the "mine" pool, and N then proceeds to set up matches as described below.

Victim reservations are redeemed by match queries. When the matching switch sends a match message to a switch, it sends the rights to a number of thieves and victims, as described above; it also sends a set of *match pairs*. A match pair consists of a match pointer and a victim count; together they describe the right to a reservation of a given size on the named node. A thief switch holding a match pair can send a match query to the node named by the match pair. A node N receiving a match query redeems the reservation by returning a set of match pairs whose match pointers point to a subset of N's children and whose victim counts sum to the victim count in the original reservation. In other words, N swaps a match pair on itself for a set of match pairs on its children. N then decrements its reservation count and adjusts information about its children so that it does not allocate the same victim processor to two queriers.

To continue our example, we show the state of the tree after the root switch α has sent a match message for two thieves and two victims to γ and while γ is in the midst of querying β in Figure 5. Switch α has decremented its thief and victim ownership counts by two, since it transferred ownership to γ via a match message, along with a match pair for two victims from β. Switch γ has decided to redeem this reservation, so it has sent a match query to β, which has responded with two match pairs naming its two children, $V1$ and $V2$. Notice that the sum of the victim counts in β's response is equal to the victim count in γ's query, and that β has decremented its reservation count, since it no longer holds the reservations on its children. Once γ receives the response, it holds the reservations on $V1$ and $V2$.

3.3 The Knife Algorithm

The Knife algorithm is used to actually match up victims and thieves. When a switch prepares to send match messages to its children, it must make sure that the match messages transfer equal numbers of thieves and victims. To minimize communication costs, a switch should send only one match message to a given child for a given matching cycle. To satisfy these goals, we use the Knife algorithm. It executes on a node in the thief subtree which owns a number of thieves and victims and holds a set of match pairs. The algorithm, running on a node N, is the following:

1. Attempt to find a subset of N's owned match pairs, the sum of whose sizes is exactly equal to the number of thieves in one of N's children C.

2. If this fails, it fails because one of the match pairs is too large. Send a match query to the switch

named by the overly large match pair to exchange it for a set of smaller match pairs.

3. If the Step 1 attempt succeeds, compose a match message containing a transfer of thief and victim ownership rights and the subset of match pairs and send it to C.

This algorithm is repeated until we finish matching all the thieves and victims.

An example of a simple execution of the Knife algorithm is illustrated in Figure 5. Switch γ is unable to send any match messages before it queries β because it holds a match pair of size 2, whereas its children are all thieves of size 1. Switch γ needs a match pair of size 1. To obtain such a match pair, it sends a match query to β. If, hypothetically, β returned a match pair of size 2, γ would then query on that pair; it continues this process into it obtains a match pair of the correct size. The querying switch essentially attempts to "cut" off a slice of the tree of the correct size, and continues "cutting" until it succeeds. For a balanced tree, no more than $logP$ queries are necessary where P is the number of processors.

While it seems as if an alternative algorithm could avoid this loop, such an algorithm causes a message explosion. Consider the alternative algorithm where a switch performs only one match query, querying its sibling or cousin. If it still holds an overly large match pair (V, a), the switch trivially divides it into two match pairs (V, b) and (V, c) where $a = b + c$. This is a legitimate operation, since the node V can fulfill two reservations just as easily as one, and appears to allow a switch to send match messages to its children while doing only one match query. The problem occurs in trees like Figure 6. Since Switch A is not willing to look ahead into B's children, it just splits the match pair for C into four pieces and sends them on to its children; C therefore gets queried four times, creating a communication hotspot in the network. The Knife algorithm does one extra query to C and thus avoids the message explosion and the hotspot.

Once a thief processor receives a match pair naming a real victim processor, it is able to steal. The Knife algorithm does not guarantee that a thief processor will receive a match pair with a real victim name; it may be given a match pointer, in which case the thief processor uses match queries to follow the chain of match pointers until it reaches the real victim processor. The thief processor steals work from the victim processor, and then destroys the ownership rights to both the thief and the victim. At this point, since the thief processor held and destroyed the ownership rights and reservations for both the thief and the victim, no node owns or reserves either one.

This set of procedures is the essence of the algorithm. Within a given subtree, ownership rights rise to the subtree root, which composes and sends match messages to its children. These match messages cascade down the thief subtree, arranging the appropriate work steals. In the meantime, the subtree root has sent residual thieves or victims to its parent, so that this same process is taking place on the next level.

The use of anonymous ownership and match pointers is basic to the distributed nature of the scheduling algorithm. If we were to use a name-passing algorithm, then the matching switch would be doing most of the scheduling work for a given set of matches. In our system, the scheduling work is distributed among all the switches in the thief and victim subtrees. A given switch in the thief subtree is only responsible for executing the Knife algorithm. A given switch in the victim subtree simply has to fulfill match queries with match pairs. The apparent inconvenience of having to do match queries and deal with match pointers is balanced by better distributed behavior for the scheduling algorithm.

3.4 Analysis

Algorithm *DDR* has the same performance bounds as Algorithm *DPR*, but Algorithm *DDR* is conservative.

Lemma 4 *If the processor-tree is balanced of depth at most* $\log P$*, then the DRAM rendezvous algorithm is conservative with respect to the tree and the permutation induced by the matching.*

Proof: The messages that go up the tree induce only a constant load on the tree. The downward-going messages are of size contain at most $O(\log P)$ node identifiers, since each matching names a contiguous region of processors by naming the maximal roots of the subtrees containing those processors. The node names and counts require at most $O(\log P)$ bits since there are only P processors to identify or count. Each subtree may receive at most one match message from each of its ancestors. The knife algorithm is used to split any particular node at most once per match message, and are only handled by the ancestors of the victims that will eventually be stolen.

Theorem 5 *Consider the execution of any fully strict multithreaded computation with critical-path length T_∞ and work T_1 using the (nonrandomized) DRAM rendezvous. The running time on P processors, including scheduling overhead, is $T_P = O((\lambda + \log P)(T_1/P + T_\infty)$.*

Proof: The same as for Theorem 2.

3.5 Randomized DRAM Rendezvous

Algorithm *RDR* uses a slight twist on the nonrandomized DRAM algorithm. In Algorithm *DDR*, a matching switch tries to match as many of its children as possible to maximize locality, without regard to whether far-away victims will ever be stolen. To make sure that each victim has a fair chance of being stolen requires global knowledge of how many thieves and victims there are. To do this, we first calculate the total number of thieves and victims by summing up the tree and broadcasting down. At this point every switch knows how many thieves and victims are below it in the tree, and how many are in the entire tree. Let T be the total number of thieves in the entire tree, and V be the total number of victims. If $T \geq V$ then each switch matches locally as before. Otherwise, in a completely fair scheme, each victim should have chance $R = T/V$ of being stolen.

We will define a "fairness coefficient", f, which tells us how fair we want to be. If $f = 1$ then the algorithm will be completely fair. If $f = 1/2$ then the algorithm will guarantee each victim a chance fR of being stolen. If $f = 0$ the algorithm reverts to Algorithm *DDR*. Each switch uses f to determine how much matching it may perform locally. If a switch has T_s thieves and V_s victims in its subtree, then it knows that there are $T - T_s$ thieves and $V - V_s$ victims in the rest of the world. The switch must guarantee that the rest of the world will have $fR(V - V_s)$ thieves available for its victims. Thus it must contribute at least $fR(V - V_s) - (T - T_s)$ thieves, and it can use up to $T - fR(V - V_s)$ of its thieves for local matching. We can thus adjust f to make a tradeoff between provably good performance and improved locality.

Theorem 6 *Consider the execution of any fully strict multithreaded computation with critical-path length T_∞ and work T_1 using the randomized DRAM rendezvous with f a positive constant. The expected running time on P processors, including scheduling overhead, is $T_P = T_1/P + O((\lambda + \log P)T_\infty)$. Moreover, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time is $T_P = T_1/P + O((\lambda + \log P)(T_\infty + \lg(1/\epsilon)))$.*

Proof: The same as for Theorem 6.

4 Empirical Results

We implemented Algorithm *DDR* in the Cilk multithreaded runtime system. We call the DRAM version of Cilk *TreeCilk*. We found that the system is not very predictable, whether we run on a controlled environment, such as the NASA Ames Whitney cluster, or on an uncontrolled environment such as the internet. Surprisingly the internet is not much worse than the local-area network.

To measure the performance of TreeCilk, we ran applications, and measured the critical path, work, and runtime. Currently TreeCilk can only run a few Cilk applications. Our version does not support dynamic memory allocation or Cilk's abort protocol. We plan to address both of these shortcomings in the near future. Because of these limitations, we only have a few programs running, including the doubly recursive fibonacci program and the synthetic `knary` benchmark [BJK+96].

We ran the `knary` program with a variety of parameters and on a variety of machine configurations. In order to plot different computations on the same graph, we normalized the machine size and the speedup by dividing these values by the average parallelism $\overline{P} = T_1/T_\infty$, as was done in [BJK+96]. Figures 7 and 8 show the outcomes of these experiments. Figure 7 includes only data from running in the Whitney ethernet-connected Pentium-Pro cluster. Figure 8 shows data from running on the internet using several whitney machines plus one Yale machine, communicating from Connecticut to California.

The linear-speedup plus critical-path model of [BJK+96] does not appear to work very well for our implementation, even when we run in the controlled environment of the Whitney cluster. The data from Figure 9, which includes only runs for which the work is at least 18 seconds is a little better, but it still appears that we do not fully understand this system. We tried a curve fit of the performance against a model of the form

$$T_P = c_1 T_1/P + c_\infty T_\infty + c_l \log P T_\infty$$

and fits that included the $\log P T_\infty$ basis function are a little better than fits that include only the T_∞ basis function, but we do not consider the curves to fit well enough to be predictive.

We suspect that the part of the system we do not understand is the synchronization. Perhaps our algorithm is "too synchronous". We plan to try some modifications to the algorithm that make it less synchronous.

We found that taking a cluster computation and adding a far-away processor often improved the speed. One would not be surprised if the overhead of adding a far-away processor overcame any additional processing power added by the processor. Sometimes adding a processor helps, and sometimes it does not. Surprisingly, we found that in situations where the addition of a local processor improves performance, the addition of a processor located across the internet also improves performance.

Link Loads

We can measure link loads on our tree structure very easily. Since our abstraction of the internet is that it is a tree-structured DRAM, there is a unique path between each source and destination. If we record the number of bytes sent to each source and each destination, we can reconstruct the bandwidth actually used. Figure 1 shows the link loads for one run of TreeCilk on 25 processors distributed across the USA. The loads across the root of the tree are an order of magnitude less than what a ordinary modem provides.

5 The Future

In time for the final paper, we hope to get the following done:

- Implement Algorithm *RDR*, the randomized DRAM rendezvous, for TreeCilk. Perhaps the improved theoretical behavior over Algorithm *DDR* will translate to improved empirical behavior.
- Loosen the synchronization requirements of the algorithm to allow different parts of the machine to perform their work more independently.
- Support the full Cilk-5 language. For dynamic memory allocation Keith Randall has already made many of the needed changes in his Distributed Cilk, and we simply need to patch our implementation with his improvements. We also hope to get the abort protocol running eventually. With these two changes we will be able to run much more interesting applications, such as blocked matrix multiplication, LU decomposition, Barnes-Hut, and the CilkChess parallel chess program.

In the medium-term we hope to improve TreeCilk. Our strategies include the following:

- Use the higher-performance networks available on the Whitney cluster. (We are using UDP on the ethernet. We hope to try these experiments on a machine with a high-performance system-area network.
- We hope to merge the fault-tolerant Cilk of Blumofe and Lisiecki [BL97] with our system so that TreeCilk will be able to tolerate processor failures.

In the longer term, internet applications such as TreeCilk will become more common, and the internet will need to provide a way to control the bandwidth consumption of these applications. It is becoming clear how one controls the bandwidth consumed by a point-to-point connection, but is still not clear how to control the bandwidth consumed by a collection of processors working on one application without slowing them down unnecessarily. Perhaps one day, we will see applications such as CilkChess running on millions of processors over the internet.

Acknowledgments

This work was supported by an NSF CAREER Grant, contract CCR-9702980 (6/1/97–5/31/99.) Mike Bernstein was supported during the summer of 1997 by a grant from the Hughes Foundation to the Perspectives on Science program at Yale College.

The NASA Ames Research Center provided access to their Whitney distributed Linux computing cluster. Whitney is supported by NASA contract NAS 2-14303 administered by MRJ Corporation at NASA Ames Research Center, Moffett Field, California. (Christopher Kuszmaul of MRJ served as the principal investigator at Ames for this collaboration.) The NASA Lewis Research Center provided access to their network of SMP processors. (Thanks to Tad Kollar and Isaac Lopez for handling our needs at Lewis Research Center.) Additional high-performance computing facilities were provided by the MIT Laboratory for Computer Science, including equipment donated by Sun Microsystems and Intel Corporation.

Thanks to Keith Randall of MIT for releasing his preliminary version of distributed Cilk to us so we could implement our algorithms. Thanks to Robert Blumofe and Charles Leiserson for helpful discussions.

References

[BJK$^+$96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pp. 207–216, Santa Barbara, California, July 1995.) (`ftp://theory.lcs.mit.edu/pub/cilk/cilkjpdc96.ps.gz`).

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, November 1994. To appear.

[BL97] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Symposium*, Anaheim, California, 6–10 January 1997. (`ftp://theory.lcs.mit.edu/pub/cilk/USENIX97.ps.gz`).

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[Blu95] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1995. (`http://www.cs.utexas.edu/users/rdb/papers/PhDthesis.ps.gz`).

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[Chr83] David Park Christman. Programming the connection machine. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, January 1983. Also available as Xerox PARC Technical Report ISL-84-3, April, 1984.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1990.

[FRL] Matteo Frigo, Keith H. Randall, and Charles E. Leiserson. The implementation of the Cilk-5 multithreaded language. Extended abstract submitted for publication. (`ftp://theory.lcs.mit.edu/pub/cilk/cilk5.ps.gz`).

[JK97] Christopher F. Joerg and Bradley C. Kuszmaul. The ⋆socrates massively parallel chess program. In Sandeep N. Bhatt, editor, *Parallel Algorithms: Third DIMACS Implementation Challenge*, pages 117–140. American Mathematical Society, 1997. Published in the DIMACS Series in Discrete Mathematics, Volume 30, as a result of the Third DIMACS Implementation Challenge held at Rutgers University, October 17–19, 1994. (An early version is at `ftp://theory.lcs.mit.edu/pub/cilk/dimacs94.ps.Z`).

[Kus94] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1994. Available as Technical Report MIT/LCS/TR-645 and as `ftp://theory.lcs.mit.edu/pub/bradley/phd.ps.Z`.

[LM88] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

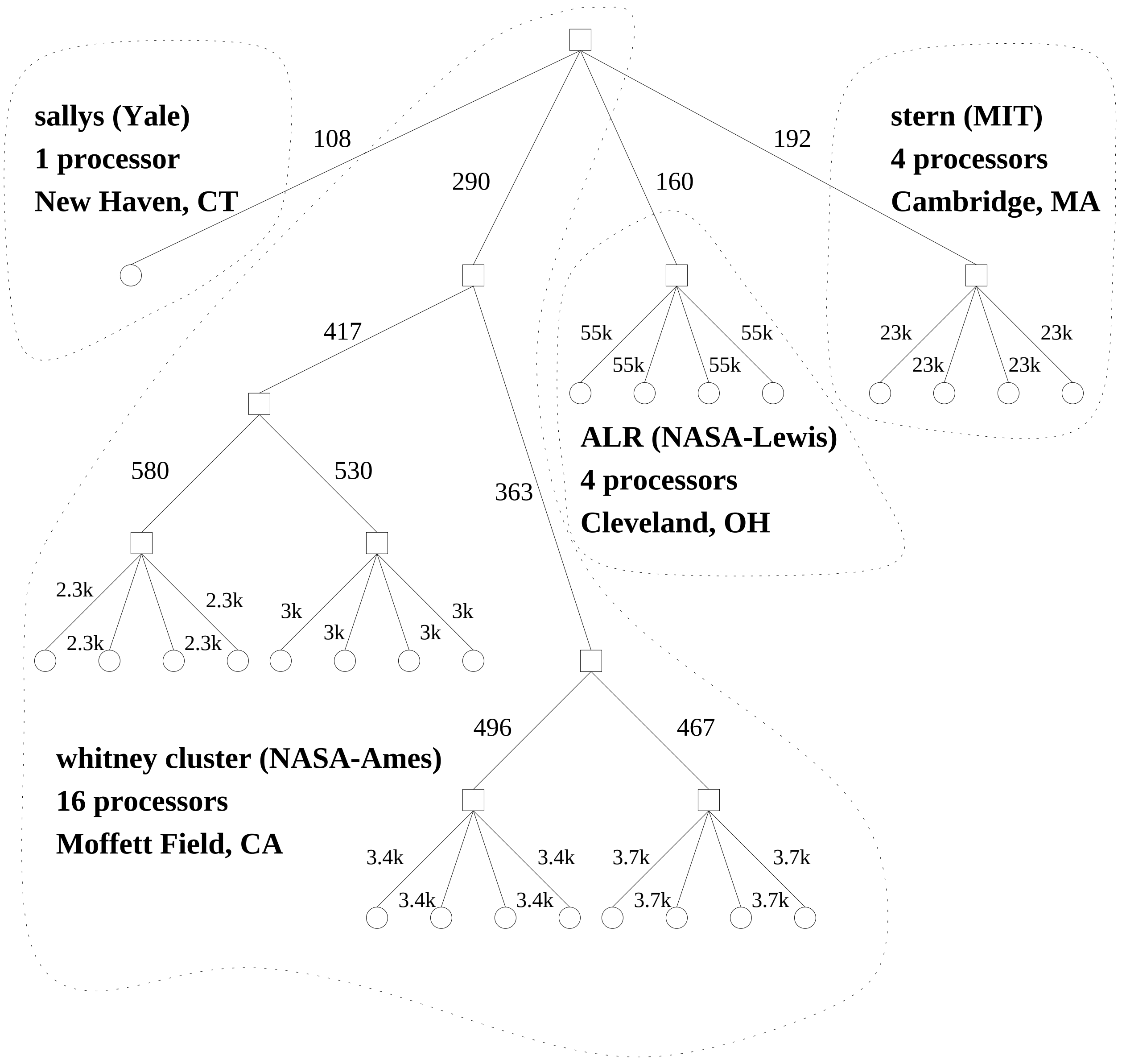

Figure 1: The bandwidths used in a big TreeCilk computation. A total of 25 processors spread across the USA were used. These data come from running the `knary` with parameters `(1,5,30000,8)`. (See [BJK+96] for an explanation of the parameters to `knary`). The dotted lines show where the internal tree nodes were physically placed. The numbers on the tree edges show the bandwidth consumed over the computation (measured in bytes per second.) The MIT and ALR machines are both 4-processor Pentium Pro 200Mhz SMP computers (represented as a subtree with four leaves). The Whitney site provided 16 one-processor Pentium Pro 200Mhz computers (represented as a mixed binary/4-ary tree of depth 3.) The Yale machine is a Pentium Pro 200Mhz computer. This computation ran for 112 seconds and achieved a speedup of about 9 on 25 processors.

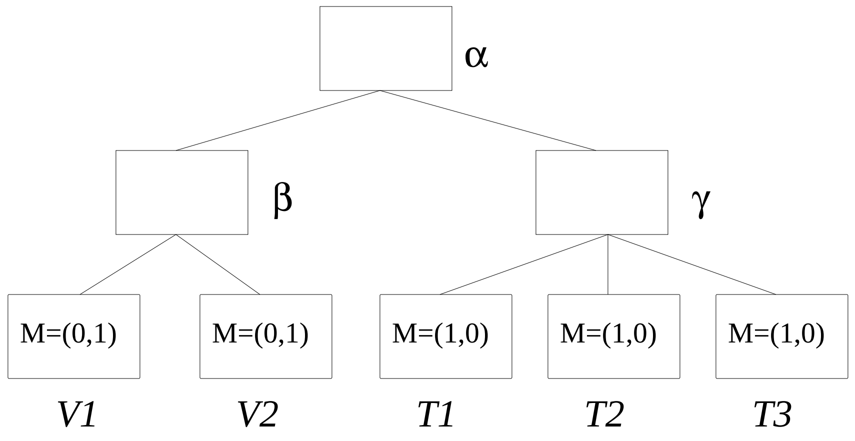

Figure 2: A tree of switches and processors before the matching process has commenced. Rectangles represent switches and rounded rectangles represent thieves. The notation $M = (t, v)$ signifies that the node owns t thieves and v victims.

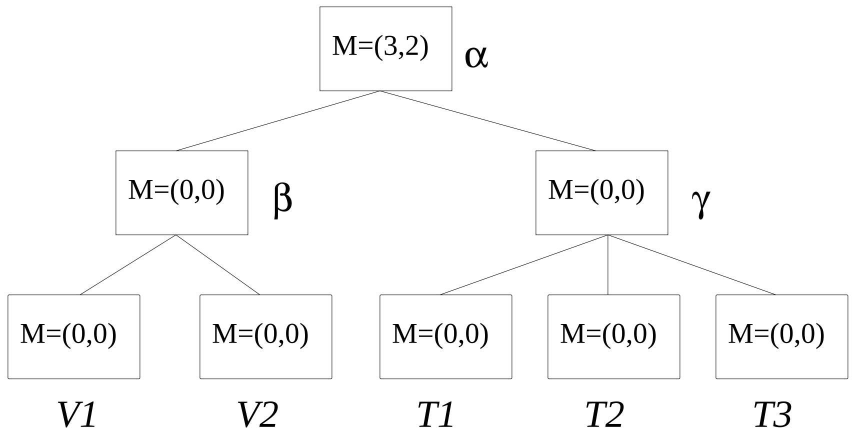

Figure 3: The state of the tree after all ownerships have been transferred up to the root.

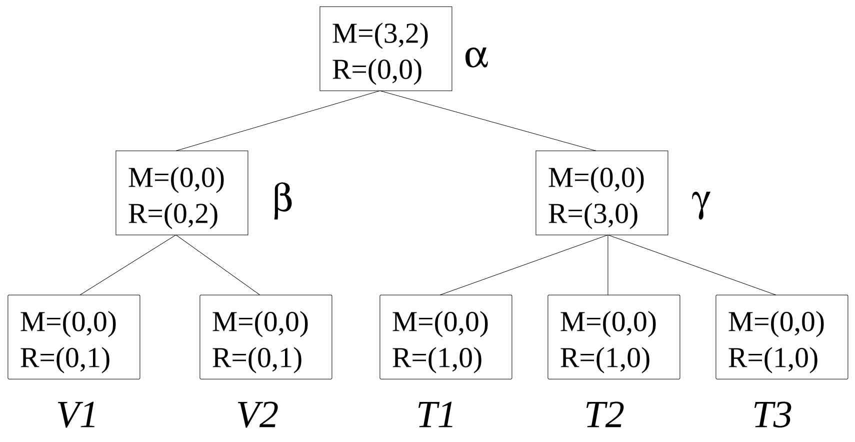

Figure 4: The state of the tree after ownership transfer with reservations illustrated. The notation “R=(t,v)” indicates that the node is reserving t thieves and v victims.

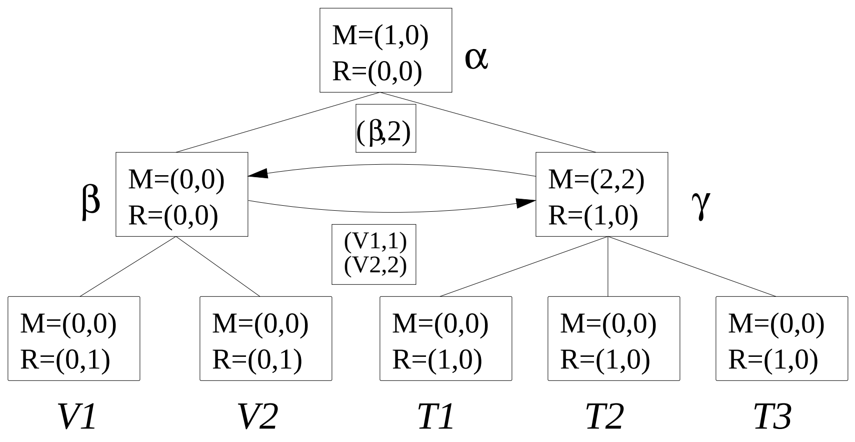

Figure 5: The state of the tree after α sent a match message to γ and while γ is in the midst of querying β. A single match pair is being exchanged for two inferior match pairs.

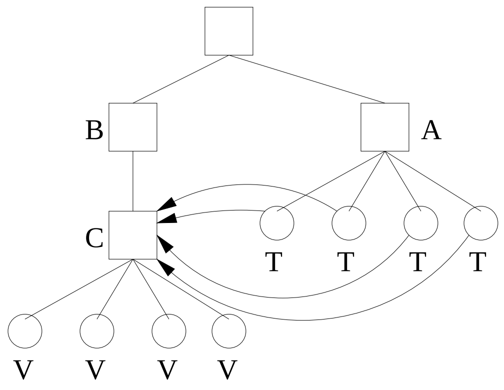

Figure 6: The message explosion which occurs if the Knife algorithm is not used. Switch A does not realize that by doing one extra query it could save C from being queried four times.

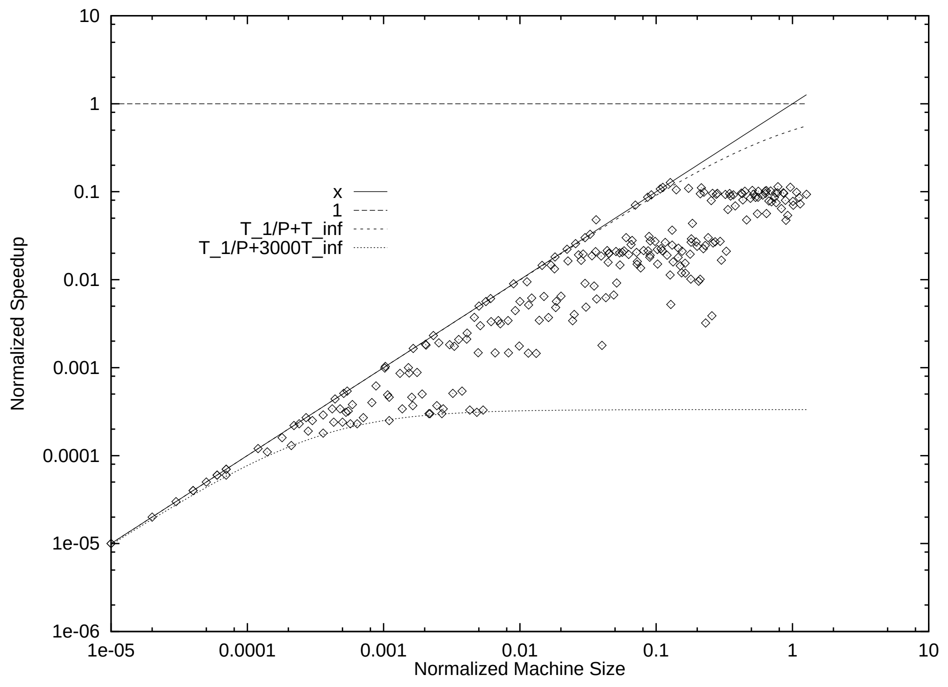

Figure 7: Various `knary` data running on the Whitney cluster.

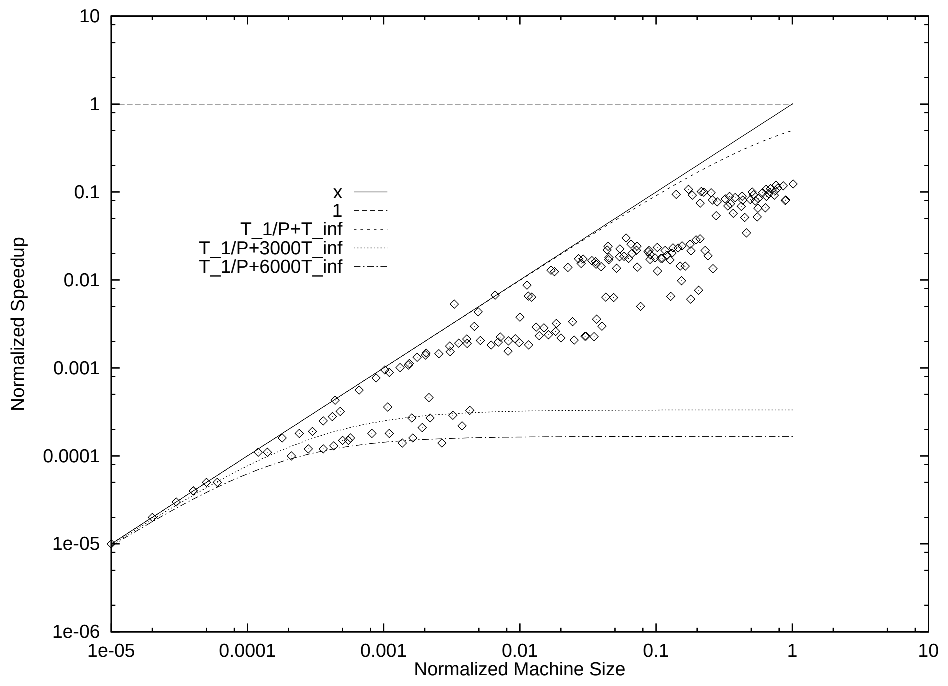

Figure 8: Various `knary` data running on the internet (Whitney plus one Yale machine.)

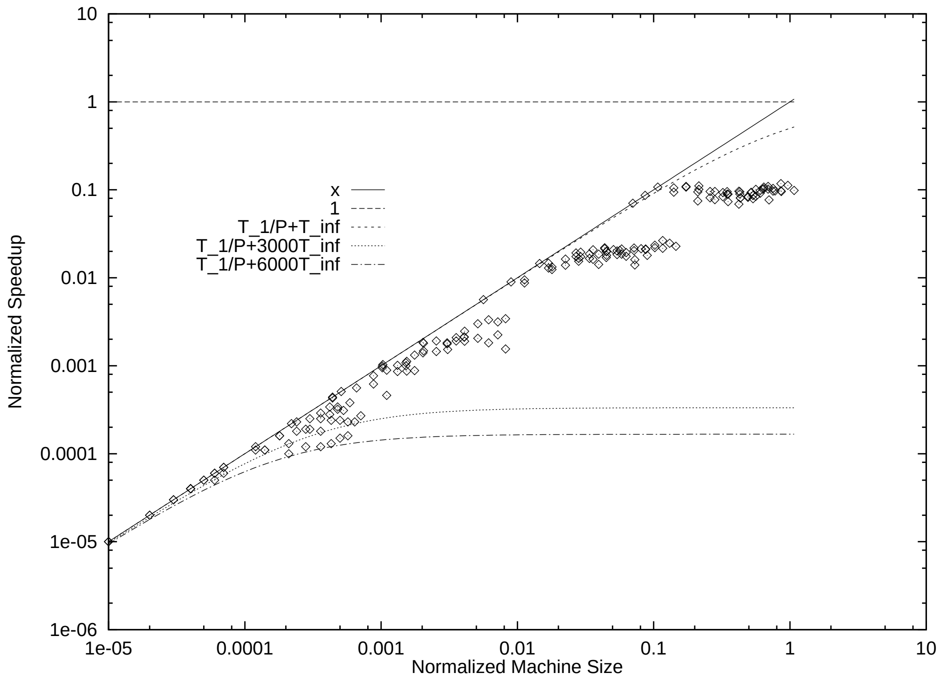

Figure 9: The Whitney and internet data with only data points for which $T_1 \geq 18$.